Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

PROGRAMAÇÃO PARALELA EFICIENTE E DE ALTO NÍVEL SOBRE ARQUITETURAS DISTRIBUÍDAS

Francisco Heron de Carvalho Junior

TESE DE DOUTORADO

Recife
9 de dezembro de 2003

Universidade Federal de Pernambuco
Centro de Informática

Francisco Heron de Carvalho Junior

PROGRAMAÇÃO PARALELA EFICIENTE E DE ALTO NÍVEL SOBRE ARQUITETURAS DISTRIBUÍDAS

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: *Prof. Rafael Dueire Lins, Ph.D.*

Recife
9 de dezembro de 2003

Dedico esta tese a todos os que dedicaram suas vidas ao progresso científico.

AGRADECIMENTOS

- A Rafael Lins, pelo competente trabalho na orientação desta tese e por ter se constituído em referencial para orientar minha formação científica;
- A CAPES, Centro de Informática e Departamento de Eletrônica e Sistemas da Universidade Federal de Pernambuco, os quais forneceram o suporte financeiro e material necessário à realização da pesquisa que originou esta tese;
- Aos meus pais, Heron e Hildene, e irmãos, Maria Helena e Hilano, pelo suporte e incentivo na busca deste objetivo;
- A Rhanna, pelo carinho, compreensão e apoio prestados nestes últimos dois anos, sendo para mim um importante fator de equilíbrio e inspiração;
- Aos alunos Fernando Lins, Nivia Quental, Marcelo Pita, Marcelo Costa, Rangner Ferraz, Rodrigo Menezes e Humberto Menezes pelo apoio técnico prestado na realização da pesquisa que originou esta tese;
- A Rogério, Raquel, Júnior, Lillian, Jerfesson, Ricardo, Alexandre e Stael, pela amizade e apoio que têm prestado desde o começo desta caminhada; e
- A Deus, por ter nos propiciado a oportunidade de desvendar o mundo que nos cerca através da habilidade de fazer ciência.

A imaginação é mais poderosa que o conhecimento. Ela amplia a visão, dilata a mente, desafia o impossível. Sem a imaginação o pensamento estagna.

—ALBERT EINSTEIN (1875-1955)

RESUMO

Mudanças paradigmáticas têm sido observadas no contexto da computação de alto desempenho a partir da última década. A consolidação das arquiteturas distribuídas, bem como o avanço no estado-da-arte das tecnologias de processadores e interconexão em redes, culminou no aparecimento dos *clusters*, redes de convencionais de computadores capazes de rivalizar com supercomputadores em seu nicho de aplicações a um custo inferior em ordens de magnitude. Recentemente, com o avanço no estado-da-arte das tecnologias de interconexão de redes de longa distância, vislumbrou-se o uso da infra-estrutura destas para construção de supercomputadores de escala virtualmente infinita. Este conceito ficou conhecido como *grid computing*. Pesquisas em todo o mundo visam viabilizar o uso destas arquiteturas para supercomputação, com resultados promissores.

Atualmente, *Clusters* e *Grids* são tecnologias de grande influência para o futuro da computação de alto desempenho. Sua maior implicação reside na miríade de novas aplicações para supercomputação, extrapolando os limites da computação meramente científica. Entretanto, estas tem se caracterizado por um maior nível de complexidade estrutural e escala, exigindo ferramentas de mais alto nível para o seu desenvolvimento.

O paradigma de programação paralela permite a implementação de aplicações sobre *clusters* e *grids*. Entretanto, a dificuldade inerente à construção de programas paralelos, assim como a carência de ferramentas de alto nível com esse propósito, levaram ao consenso sobre a necessidade em investir no desenvolvimento de modelos avançados voltados a programação paralela eficiente e de larga escala.

O modelo # (*hash*) de programação paralela, produto desta tese de doutorado, surge como uma alternativa aos mecanismos eficientes convencionais de desenvolvimento de programas paralelos sobre arquiteturas distribuídas. Foi desenvolvido segundo um conjunto de premissas induzidas pelo contexto que se criou com o surgimento e disseminação das tecnologias associadas a *cluster* e *grid computing*. O modelo surge com sólidas fundações em modelos formais baseados em redes de Petri, permitindo a análise de propriedades e avaliação de desempenho de programas usando ferramentas pré-existentes adaptadas a esse disseminado formalismo. Implementa-se a linguagem Haskell$_{\#}$, a qual adere ao modelo #, usando Haskell para descrever computações. O uso de Haskell permite a ortogonalização transparente entre os meios de coordenação e computação de um programa #. Complementa ainda o arcabouço de análise formal de programas #, extendendo-o no nível de computação, devido a existência de ferramental adequado ao tratamento formal de linguagens funcionais puras e não-estritas.

Palavras-chave: Processamento de Alto Desempenho, Programação Paralela, Programação Funcional, Engenharia de Programas, redes de Petri

ABSTRACT

New technological paradigms have emerged in high-performance computing since last decade. The consolidation of distributed architectures and the recent advances in the state-of-the-art performance of processor and interconnection networks gave birth to *clusters*, conventional computer networks that can compete with supercomputers in its niche of applications, but with costs of an order of maginitude smaller. Recently, the advances in the state-of-the-art of interconnection of wide area networks interconnection technologies have suggested the use of them as virtually infinite scalable supercomputers, called *grids*. In the last years, great research efforts have been spent world wide to make possible the use of internet for super-computing applications, with promising results.

Cluster and *Grid computing* are now key concepts in high-performance computing, dictating its further directions. Besides widening the number of potential users, a myriad of new applications have been induced in this area, extrapolating merely scientific computing, traditionally the most important niche of applications for high-performance computing. However, these new applications have been characterised by higher structural complexity and scale, requiring higher level tools to support their development.

The parallel programming paradigm is the essential tool for the development of applications that take advantage of performance of the *clusters* and *grids*. However, its inherent complexity and the lack of efficient higher level tools for the development of applications claim for efforts to be conducted providing more advanced models for developing efficient parallel programs, support their validation by formal methods and adapted to the modern software engineering techniques.

The # parallel programming model, a product of this thesis, emerges as an alternative to the conventional mechanisms to support development of parallel programs over distributed architectures. Targeted at emerging cluster and grid technologies. The # model has solid foundations in formal models based on Petri nets, allowing property analysis and performance evaluation of parallel programs, using existing tools based on this formalism. The Haskell$_{\#}$ language has been implemented, adhering the # model, but using Haskell to describe sequential computations. The use of Haskell allows to make transparent the separation between coordination and computation media of # programs.

Keywords: Parallel Programming, High Performance Computing, Functional Programming, Software Engineering, Petri Nets

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

Neste capítulo introdutório, são contextualizadas e discutidas as motivações aos problemas que constituem o objeto de estudo desta tese, constituindo o substrato para o delineamento de seus objetivos.

1.1 ANTECEDENTES

A presente tese surge no contexto das pesquisas que vem sendo desenvolvidas desde mais de uma década pelo grupo de pesquisa liderado pelo orientador desta, com o objetivo do desenvolvimento de uma linguagem funcional paralela integrada a um ambiente de desenvolvimento e prova de propriedades formais de programas, possibilitando a prototipação rápida e simulação de aplicações, sendo dessa maneira fiel aos princípios básicos da programação funcional. Em contraponto às correntes então vigentes, advogava Lins que "as aplicações onde a corretude é crítica, também o é o desempenho", tendo o mesmo sido um dos que buscaram a eficiência das linguagens funcionais com o desenvolvimento de várias otimizações em máquinas interpretadas baseadas em combinadores categóricos [150, 155, 212, 151]. Sob a inspiração da máquina G de T. Jonhsson [129], Lins chegou a modelos compilados, tendo sido, na máquina ΓCMC [154], um dos pioneiros no desenvolvimento de técnicas que vieram a ser mais tarde largamente adotadas e conhecidas como *unboxing*. Também foi Lins um dos pioneiros a advogar a incorporação de bibliotecas e códigos, inclusive nativo, aos programas funcionais, desde que convenientemente encapsuladas de forma a não comprometer a corretude de programas. Dessa forma, haveria um refinamento dos programas em busca de eficiência.

Não é por acaso que o modelo # apresentado nesta tese possui combinadores baseados nos construtores de OCCAM. As primitivas de comunicação de OCCAM nada mais são do que a implementação de CSP (*Calculus of Communication Processes*) de Hoare, um formalismo consagrado e com semântica bem definida. Quando da fundação do grupo de usuários de OCCAM e Transputers para a América Latina em 1998, em Florianópolis, Lins defendia a separação entre coordenação (combinadores ao estilo de OCCAM) e código sequencial na construção de programas paralelos, uma vez que a hierarquização tonaria mais simples o propósito de prova de programas.

Nessa mesma direção, uma grande dificuldade necessita ser superada: os formalismos de álgebra de processos (CSP, CCS, π-calculus), embora oferecessem semântica bem definida, mostravam-se de pouco intuitivas para programadores comuns. Qualquer aplicação, por menor que fosse, quando traduzida para um desses formalismos ganhavam dimensões e complexidade que a tornava impossível de ser gerenciada por seres humanos. Por outro lado, os provadores de teoremas para tais formalismos haviam alcançado pouco progresso. Foram essas mesmas razões que ressuscitaram as redes de Petri no início da

década de noventa, como uma alternativa onde o apelo gráfico trazia alguma "intuição" a respeito do sistema modelado. Por essa razão, em 1995, Lins começou a fazer a ponte entre tais formalismos, o que serviu como base para o trabalho aqui descrito.

Do ponto de vista de implementação, a estrada percorrida foi igualmente longa e tortuosa. Seu início deu-se com a doação pela INMOS de placas de Transputers T800 a Universidade Federal de Pernambuco. Um *cross-compiler* gerava o código sequencial para o interpretador GMC[176]. OCCAM era a "cola" que conectava os processos funcionais. As limitações do *hardware* e a ineficiência do interpretador ficaram patentes. A concretização do projeto de tal linguagem funcional paralela só veio a ocorrer com a doação do IBM SP2[1] à UFPE, já com a participação do autor desta tese como aluno de mestrado. A autor desta tese já toma como plataforma de implementação um ambiente de compilação distribuída visando *clusters*, numa evolução natural do modelo anterior, sob suposições advindas do panorama atual da computação de alto desempenho advinda do emergimento destas arquiteturas, discutidas adiante, sendo ainda responsável pela concretização e apresentação de soluções para diversos aspectos surgidos durante a evolução desse projeto de longa data.

As seções seguintes contextualizam o panorama atual da computação de alto desempenho, delineando-se os problemas contemporâneos para os quais esta tese apresenta soluções.

1.2 COMPUTAÇÃO DE ALTO DESEMPENHO: PANORAMA ATUAL

Nas últimas décadas, a rápida evolução da tecnologia de computadores pessoais tem sido motivada pelo crescente investimento das grandes indústrias de computação. Tradicionalmente, o desenvolvimento de processadores mais velozes tem sido visado. Com a recente disseminação do uso de computadores pessoais ligados em rede, em especial motivada pela popularização da *Internet*, o desenvolvimento de interfaces de comunicação mais eficientes tem sido focalizado. Motivam as indústrias a grande concorrência entre estas visando o domínio de um mercado com escala extraordinária, com faturamentos de grande magnitude e sem estimativas para estagnação em seu crescimento. Esta grande escala de mercado torna possível a amortização dos investimentos da indústria em pesquisa e desenvolvimento, sem que isto influencie de forma perceptível o preço dos equipamentos aos consumidores.

Ao longo da história, tem-se observado que a velocidade com a qual a tecnologia empregada em uma certa classe de computadores evolui é proporcional à sua escala de mercado. Por exemplo, durante a década de 80, *estações de trabalho* ocuparam o lugar de minicomputadores e *main-frames* em seu nicho de aplicação. A maior escala de mercado das estações de trabalho, em especial devido ao seu menor custo em relação à arquiteturas de classes superiores, permitiu uma maior quantidade de investimentos que permitiram sua evolução.

Fenômeno semelhante observou-se no mercado de supercomputação. No final década de 80, o crescimento no desempenho das estações de trabalho, acompanhado do barateamento de seu custo de aquisição, motivou o uso desta tecnologia para construção de supercomputadores [4]. Surgiram então as arquiteturas de *processamento massivamente*

paralelo (MPP), supercomputadores de memória distribuída que empregam processadores originalmente desenvolvidos para estações de trabalho, porém conectados por meio de uma interface de comunicação proprietária de alto desempenho. Tais arquiteturas contrapõem-se às arquiteturas vetoriais e multi-processadas, as quais dominavam o mercado de supercomputação até então [31]. MPP's diferenciam-se de redes de estações de trabalho ligadas em rede devido ao tipo de interface de comunicação adotada, a qual assume uma série de restrições com a finalidade de tornar mínima a latência de comunicação entre os processadores. Por exemplo, em MPP's, é comum organizarem-se estaticamente os processadores segundo topologias regulares características, como *hipercubos* e *malhas*, de forma a adequar-se a certos tipos de aplicações relevantes e obter melhor desempenho a um custo inferior. Exemplos desta classe de arquiteturas são: Intel Paragon, Cray's T3D e T3E, *Thinking Machines*'s CM2 e CM5, IBM SP2, ASCI Red e Sun HPC.

Entretanto, em MPP's, os custos de tempo e capital associados ao desenvolvimento de uma nova tecnologia de interconexão de unidades de processamento pertencentes a uma nova família de processadores mais velozes mostram-se muito altos. Além disso, a rapidez da evolução do desempenho associado às tecnologias de interconexão não tem acompanhado proporcionalmente a rapidez da evolução do desempenho dos processadores. Por estes motivos, para um certo fabricante, a tecnologia aplicada nas MPP's topo de linha tende a ser inferior, em termos de desempenho, à tecnologia usada nas estações de trabalho de última geração. Isso é causado pelo tempo necessário, após o aparecimento de uma nova geração de processadores, para o desenvolvimento de um supercomputador baseado na nova tecnologia, o chamado *lag time*. Com isso, à época do lançamento de uma nova MPP, a tecnologia do processador utilizada encontra-se defasada. Soma-se a estes fatos, a pressa dos fabricantes em lançar novas gerações de processadores para fazer frente à concorrência, atraindo novos usuários e aplicações. Assim, tornava-se mais comum que estações de trabalho topo de linha, conectadas em uma rede de comunicação suficientemente eficiente, fossem mais atrativas do que os supercomputadores, em nichos de aplicação antes exclusivos a esta classe de arquiteturas [4].

A partir de meados da década de 90, computadores pessoais viriam a rivalizar com estações de trabalho. Empresas como Intel, AMD, Cyrix e Apple, voltadas ao desenvolvimento de processadores para computadores pessoais, rapidamente conquistaram espaço no mercado antes dominado pela SUN, DEC-ALPHA, etc, fabricantes de estações de trabalho de arquitetura RISC, o que levou ao quase desaparecimento destas em vários nichos de mercado, principalmente na engenharia. Também neste caso, o maior investimento das indústrias de PC's em pesquisa e desenvolvimento, possível devido a maior escala do mercado de computadores pessoais, foi fator determinante.

Não demorou muito para que computadores pessoais também rivalizassem com supercomputadores em nichos de aplicação antes restritos a esta tecnologia. Em 1992, o projeto *Beowulf* [29], desenvolvido na NASA sob a coordenação de Donald Becker, pioneiramente demonstrou como computadores pessoais ligados em redes convencionais e equipados com *software* adequado podiam equiparar-se em desempenho a supercomputadores, porém a uma fração de seu custo. Surgem assim os primeiros *clusters* de computadores pessoais, os quais podiam ser montados com componentes disponíveis em estabelecimentos comerciais comuns ("*hardware* de prateleira") [19]. Vale ressaltar a importância do surgimento

de sistemas operacionais gratuitos e de código aberto, como o Linux, na evolução dos *clusters*, reduzindo ainda mais os custos associados à aplicação desta tecnologia. Deve-se ainda ao conceito de código aberto, adotado pelos desenvolvedores do sistema Linux, a possibilidade do desenvolvimento de protocolos de comunicação e *software* de suporte adaptado ao processamento de alto desempenho em *clusters*, o que constituiu importante aspecto explorado no trabalho de Donald Becker e sua equipe. Demonstrou-se que o *software* tem importância fundamental no uso eficiente de *clusters* para computação de alto desempenho.

Desde então, muitos experimentos com a aplicação de *clusters* têm alcançado resultados similares aos alcançados pelo projeto *Beowulf*, motivando o surgimento no meio acadêmico de forte interesse no desenvolvimento e aplicação desta tecnologia emergente [35]. Esse fato culminou com o reconhecimento de *cluster computing* como umas das áreas de pesquisa de interesse da IEEE, com a criação de um grupo de interesse específico[1], de forma a unir esforços multi-disciplinares de pesquisadores provenientes das áreas de redes de computadores, sistemas distribuídos, arquiteturas de computadores, processamento de alto desempenho, linguagens de programação, etc [20].

É comum argumentar-se que os componentes de *software* têm papel preponderante na evolução dos *clusters*, e não os componentes de *hardware*[31]. Contribuiu para isso, os muitos anos de pesquisa e desenvolvimento, sobretudo nos EUA, voltados ao desenvolvimento de ferramentas de programação para arquiteturas distribuídas, as quais resultaram no aparecimento de bibliotecas de passagem de mensagens de uso geral, como MPI [73] e PVM [97], e bibliotecas científicas pré-paralelizadas de alto desempenho, como PetSC, Linpack, IMSL, Reduce, etc. Deve-se isso às dificuldades tradicionalmente impostas pelo governo norte-americano para a aquisição de supercomputadores vetoriais japoneses.

O uso de *clusters* para fins de computação de alto desempenho não ficou restrito ao meio acadêmico/científico, onde proliferavam os grupos de pesquisa em diversas áreas do conhecimento interessados no uso desta tecnologia, principalmente para simulação computacional. Seu baixo custo e potencial para reuso de parques computacionais pré-existentes, levou a sua aplicação em diversas áreas da indústria, comércio, governo, etc. Em um contexto inimaginável há poucas décadas, usuários com orçamentos reduzidos para aquisição de supercomputadores, tornaram-se capazes de fazer uso efetivo do potencial desta tecnologia. Para países em desenvolvimento como o Brasil, este é um fator importante para alavancar seu desenvolvimento científico e tecnológico. Há algum tempo, por exemplo, a PETROBRÁS, repetindo com sucesso o que outras multinacionais da indústria do petróleo vinham realizando a algum tempo [203], tem empregado *clusters* para execução de aplicações de seu interesse, em especial a simulação de reservatórios de petróleo [165], aplicação de grande interesse econômico e com fortes requisitos computacionais [54].

A Figura 1.1 apresenta a evolução dos 500 supercomputadores mais poderosos em operação, classificados segundo sua arquitetura, desde junho de 1993. Esta estatística, atualizada semestralmente, pode ser acompanhada através do *site Top 500 Supercomputer Sites* [167], mantido por grupos de pesquisa em universidades americanas e indústrias,

[1] www.ieeetfcc.org

interessadas em pesquisa e desenvolvimento na área de processamento de alto desempenho. Observa-se que, em especial a partir do ano 2000, a participação dos *clusters* e NOW's (*networks of workstations*) tem crescido exponencialmente. Segundo o levantamento publicado em novembro de 2002, esta participação atualmente alcança a proporção de 18,6%. Isso evidencia que *clusters* se tornaram uma realidade em supercomputação e dentro de alguns anos devem dominar esse mercado, em especial devido ao forte interesse da indústria nesta tecnologia.

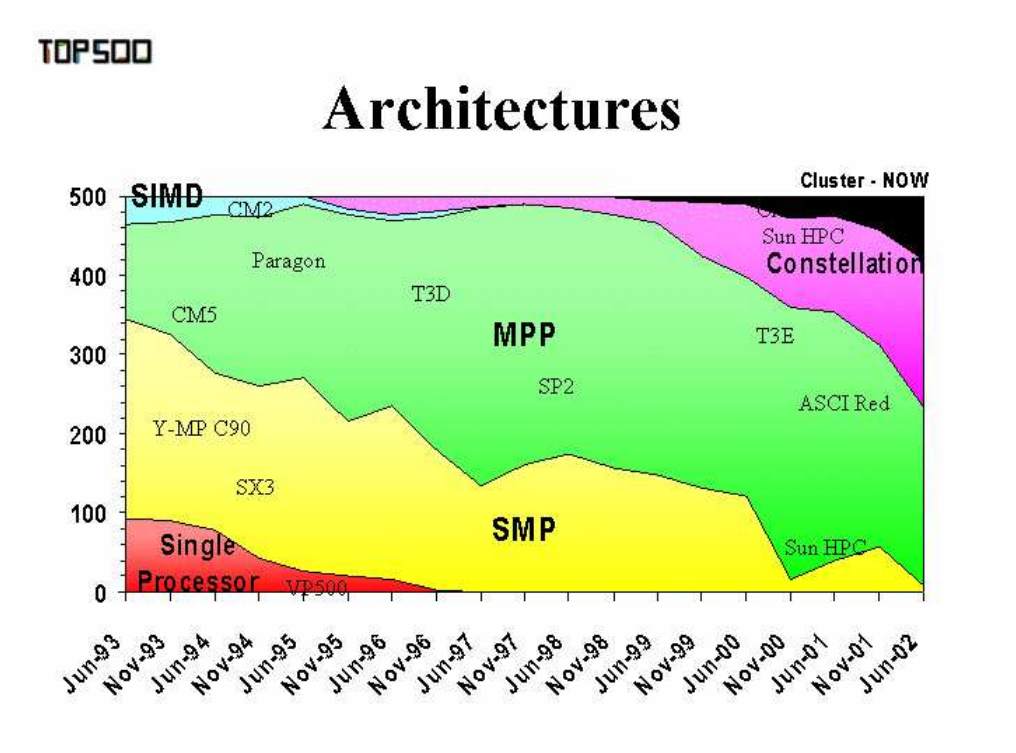


Figura 1.1. Classificação Arquitetural dos 500 supercomputadores de maior desempenho (http://www.top500.org)

Além da disseminação da tecnologia de *cluster computing*, outra importante tendência em computação de alto desempenho advém da aplicação da tecnologia de *grid computing* para a construção de supercomputadores de escala virtualmente infinita, onde os nós podem eventualmente estar separados geograficamente[89]. O surgimento da tecnologia de *grid computing* tem suas origens na disseminação das tecnologias relacionadas à *Internet*, a qual motivou o surgimento de tecnologias de alto desempenho para interconexão de computadores em redes de longa distância. Essa tecnologia emergente tem sugerido uma miríade de novas possibilidades de aplicações para a computação de alto desempenho [89]. Nos EUA, esforço tem sido conduzido nos últimos anos para interconexão dos grandes centros de supercomputação espalhados pelas instituições acadêmicas nacionais e indústrias de computação, formando *grids*. Tecnologias e conceitos comuns em sistemas distribuídos têm uma importância fundamental para tornar possível o uso

deste novo tipo de arquitetura, permitindo aos usuários e programas abstrairem-se do caráter disperso dos nós de processamento que compõem essas arquiteturas. Inúmeros são os grupos de pesquisa que têm se dedicado ao estudo de novas tecnologias de suporte ao desenvolvimento e gerenciamento de aplicações em arquiteturas de *grid*. Destacam-se as contribuições das pesquisas desenvolvidas no ANL (*Argonne National Laboratory*), notadamente pelo grupo de pesquisa liderado pelo professor Ian Foster, e no ICL (*Innovative Computing Laboratory*), na Universidade do Tennessee, EUA, sob a coordenação do professor Jack Dongarra [231]. Dentre as consequências advindas da aplicação de *grids* em supercomputação, destaca-se o aumento em escala e complexidade das aplicações, exigindo ferramentas de programação de mais alto nível para lidar com o seu desenvolvimento de forma eficiente. Essencialmente, este constitui o problema endereçado nesta tese com respeito ao uso desta tecnologia emergente.

1.3 PROGRAMAÇÃO PARALELA PARA TODOS

Apesar de seu baixo custo e aplicação em diversas áreas, o uso de *clusters*, sobretudo em países em desenvolvimento como o Brasil, encontra empecilhos. O mais relevante destes é a carência de massa crítica especializada na instalação e operacionalização de arquiteturas de alto desempenho, e, principalmente, na implementação eficiente de aplicações sobre estas arquiteturas. Além da escassa formação nessa área oferecida pelas universidades brasileiras, contribui para este fato o maior grau complexidade inerente à tarefa de programação sob *clusters*, quando comparada à programação convencional.

Em *clusters*, a latência de comunicação é alta quando comparada à MPP's. O uso eficiente deste tipo de arquitetura é possível com o uso de ferramentas explícitas de programação paralela, onde o programador é responsável pela configuração e gerenciamento dos processos paralelos que compõem a aplicação. Isso se deve ao fato das dificuldades computacionais inerentes a paralelização automática de programas e seu gerenciamento dinâmico eficiente [76], cujas melhores soluções conhecidas aplicam-se a arquiteturas apropriadas ao paralelismo de fina granularidade, onde a latência de comunicação é comparavelmente insignificante. Mesmo nestas arquiteturas, os resultados satisfatórios não se aplicam a programas paralelos em geral, mas somente para aqueles que satisfazem certas restrições impostas pelo modelo de programação, como topologias regulares e pré-definidas de processos, em geral associadas a características intrínsecas da arquitetura paralela em questão.

Exemplos de ferramentas comuns para programação paralela sobre *clusters* são as bibliotecas de passagem de mensagens PVM (*Parallel Virtual Machine*) [97] e MPI (*Message Passing Interface*) [73], tendo esta última se tornado um padrão nos últimos anos, sobretudo devido a proliferação dos *clusters*. Ambas possuem implementações para C e Fortran. A primeira volta-se a ambientes heterogêneos, implementando o conceito de *máquina virtual* para abstrair as diferentes característcas das máquinas que compõem a rede, sobretudo no que diz respeito ao formato de tipos de dados, enquanto a última volta-se a arquiteturas homogêneas, com suposições que lhe asseguram melhor desempenho em relação à PVM.

Entretanto, a tarefa de programação paralela é inerentemente mais difícil que a tradi-

cional programação sequencial [88]. Além de lidar com a implementação das computações que caracterizam as funcionalidades do programa, o programador deve coordenar a execução de um conjunto, possivelmente grande, de tarefas paralelas, organizadas segundo uma topologia particular configurada explicitamente, sobre um conjunto de nós de processamento. Na programação paralela, são preocupações comuns a instanciação e sincronização de processos, a adequação da implementação às características do *cluster*, o balanceamento de carga, o controle de localidade de dados e o controle de granularidade [204].

Além de inerentemente mais complicada que a programação sequencial, a programação paralela carece de um modelo consensual [204]. Em geral, as linguagens existentes foram desenvolvidas de forma a atender às restrições de arquiteturas específicas e necessidades de certas classes de aplicações, possuindo cada uma suas peculiaridades. O projeto de modelos ao mesmo tempo portáveis e eficientes de programação paralela tem sido portanto dificultado devido às diferentes características das arquiteturas de computação de alto desempenho [83, 123]. Entretanto, na última década tem-se observado a consolidação das arquiteturas paralelas distribuídas, notadamente *clusters*, constelações e MPP's, em contraposição às máquinas de memória compartilhada e vetoriais, devido a sua maior escalabilidade e ao surgimento de interfaces de comunicação de maior desempenho para esta classe de arquiteturas [80, 167]. Isso pode ser comprovado na Figura 1.1. O surgimento de bibliotecas de passagem de mensagens padronizadas para estes ambientes também tem contribuído para sua consolidação. Entretanto, o modelo de programação destas bibliotecas, baseado no emprego explícito de primitivas de instanciação de processos, troca de mensagens entre estes, e sua organização em topologias, é bastante complexo para aplicação no desenvolvimento de aplicações paralelas de grande escala. A partir de um certo grau de complexidade, aplicações programadas por meio de bibliotecas de passagem de mensagem são pouco estruturadas, dificultando sua manutenção, compreensão e depuração. Exigem assim programadores altamente especializados para compensar os efeitos advindos das dificuldades inerentes a esse estilo de programação.

Vários modelos de linguagens de programação paralela distribuída têm sido propostos [204]. Entretanto, aqueles que se propõem a oferecer um maior nível de abstração e portabilidade, pagam o preço da menor eficiência e escalabilidade limitada. Além disso, poucas são as linguagens que oferecem meios simples para análise formal de propriedades de programas paralelos, fator decisivo para construção de programas confiáveis e cujo desempenho pode ser assegurado formalmente, apesar da existência de muitos formalismos dedicados a prova de propriedades, como CSP [115], CCS [169], redes de Petri [188] e π-calculus [171] e avaliação de desempenho, como PRAM [85], BSP [222] e LogP [66] e suas variantes [3, 175, 125].

Os fatores discutidos acima com respeito à tarefa de programação paralela sobre *clusters* dificultam a utilização destas arquiteturas, uma vez que significativa parcela dos usuários de fato dessa tecnologia são leigos em programação, como matemáticos, físicos, biólogos, químicos, engenheiros, etc. Os recursos financeiros para a contratação de pessoal qualificado para a tarefa de implementação eficiente de suas aplicações em *clusters* são escassos, principalmente em instituições acadêmicas de pesquisa. Dificulta ainda o alto custo de contratação desse tipo de profissional, atualmente escasso. Além

disso, vale ressaltar que mesmo programadores profissionais necessitam de treinamento especial para programação paralela eficiente em arquiteturas distribuídas, tendo em vista ser esta uma tecnologia que até poucos anos atrás estava restrita a centros de pesquisa e indústrias de grande porte, como petrolífera e aeroespacial, em países desenvolvidos.

Sob o ponto de vista defendido nesta tese, os modelos de programação paralela distribuída mais promissores têm sido aqueles que se enquadram no paradigma de coordenação [99]. Este emergiu a partir do início da década de 90 como um arcabouço que inspirou o surgimento de novas classes de linguagens de programação concorrentes [98, 9]. O paradigma de coordenação pressupõe a divisão de um programa concorrente em dois níveis: *computação* e *coordenação*. As entidades computacionais (processos) cooperam através de um meio de coordenação, o qual pode ser implementado sob diversas formas, podendo ser orientado a dados ou a processos [8]. Uma linguagem de programação concorrente é dita de coordenação quando torna transparente os meios de computação e coordenação, ortogonalizando as tarefas de configuração do paralelismo e implementação das computações realizadas pelo programa paralelo. A tarefa de programação paralela é facilitada, por oferecer aos programadores a possibilidade de raciocinar independentemente sobre as partes paralelas e sequenciais de programas.

Muitas linguagens têm sido desenvolvidas ou enquadradas segundo as suposições do paradigma de coordenação [9, 10, 182, 12, 40, 161, 230, 46, 196, 173, 140, 82, 23, 213, 62, 100]. Aquelas que influenciaram no trabalho desenvolvido nesta tese pertencem a duas classes. A primeira inclui aquelas linguagens de coordenação que empregam linguagens funcionais, em nível de computação ou coordenação, tomando vantagem de certas características importantes que distinguem essas linguagens, como o suporte a funções de alta ordem e avaliação *lazy*, as quais tornam possível abstrair fluxo de controle e operações de entrada e saída da especificação das computações. Exemplo de linguagems de coordenação baseadas em linguagens funcionais são Eden [41] e Caliban [136, 208], as quais usam Haskell em nível de computação, e Delirium [157] e SCL [69], as quais empregam linguagens funcionais como meio de coordenação de processos. A segunda classe inclui aquelas linguagens de coordenação que empregam o conceito de linguagem de configuração no nível de coordenação, como PCL [205], Darwin [161], CONIC [163], Olan [32], Durra [25], Polilyth [193], dentre outras, em geral aplicadas para a construção de sistemas distribuídos. Na literatura, estas são incluídas dentre as linguagens de coordenação orientadas a processos. Para essas linguagens, muitos trabalhos demonstram sua adequabilidade dentro de um contexto de engenharia de *software* distribuído, oferecendo um alto nível de modularidade [71, 143] e abstração. Este trabalho introduz o uso de configuração dentro de um contexto de programação paralela de alto desempenho, a exemplo de OCCAM [124], a partir de meados da década de 80. Entretanto, OCCAM não pode ser considerada uma linguagem de coordenação. Em uma abordagem totalmente oposta, OCCAM não fazia distinção entre processos e computações, os tratando indistintamente [124].

Um outro importante paradigma, usado largamente no contexto de programação paralela, é a programação baseada em *esqueletos de algoritmos* [64]. Estes foram introduzidos por Cole em 1989 com a finalidade de capturar padrões de algoritmos que pudessem ser reusados em várias aplicações, com implementações otimizadas pré-definidas para difer-

entes arquiteturas, de forma a torná-los portáveis. Esqueletos de algoritmos tornaram-se bastante úteis para programação concorrente e, em especial, paralela, oferecendo uma forma elegante de abstrair características intrínsecas de uma arquitetura alvo [67, 107, 43]. Originalmente, esqueletos de algoritmos foram introduzidos como HOF's (*higher order functions*), de forma que linguagens funcionais tornaram-se bastante comuns para expressar esqueletos [69, 17, 95]. Vale ressaltar a existência de linguagens que empregam conjuntamente esqueletos e linguagens de configuração, como TPascal [95]. Neste trabalho, nos referiremos a esqueletos aplicados em linguagens de configuração como *esqueletos topológicos*.

Observando o que foi discutido nos parágrafos anteriores, tendo em vista as limitações atuais da tecnologia de *cluster computing* e o aumento da complexidade das aplicações paralelas motivada pelo emergimento da tecnologia de *grid computing* em supercomputação, o surgimento de mecanismos de programação paralela distribuída que conciliem mecanismos de abstração de alto nível, simplicidade, alta expressividade e fácil verificabilidade com alta eficiência, alta portabilidade e escalabilidade é um dos grandes desafios para os pesquisadores desta área.

Em outras palavras: **assim como foi possível tornar arquiteturas de processamento de alto desempenho acessíveis para todos, um dos desafios subseqüentes é fazer o mesmo com relação aos mecanismos de programação (paralela) sobre essas arquiteturas, tornando-as mais simples e abstratas sem grande comprometimento na eficiência, portabilidade ou escalabilidade de programas paralelos de qualquer escala**.

1.4 O MODELO # (HASH)

O modelo #, produto desta tese de doutorado, surge como uma alternativa aos meios convencionais voltados ao desenvolvimento de programas paralelos. Foi concebido segundo as tendências atuais em computação de alto desempenho advindas da disseminação das tecnologias de *cluster computing* e *grid computing*, discutidas na seção anterior, no que diz respeito às novas classes de usuários e aplicações que têm emergido. O modelo # oferece um mecanismo de programação de alto nível, adaptado às modernas metodologias de desenvolvimento de *software* de grande escala. Por ter sido desenvolvido de forma acoplada a modelos formais baseados em redes de Petri, oferece a possibilidade de análise de propriedades e avaliação de desempenho de programas usando ferramentas pré-existentes e disseminadas. Reuso e modularidade constituem preocupações onipresentes no projeto e implementação deste modelo.

Essencialmente, uma aplicação desenvolvida segundo o modelo # pode ser observada sob dois mundos em dimensões ortogonais: o *mundo das computações* e o *mundo dos processos*. O primeiro diz respeito às funcionalidades oferecidas pela aplicação, descrevendo as computações que as implementam, enquanto o último diz respeito a sua execução paralela, empregada para que esta seja realizada de maneira mais eficiente aproveitando ao máximo os recursos disponíveis em uma arquitetura distribuída. A configuração do paralelismo é explícito, sob total controle do programador, como convém a um modelo eficiente para programação paralela sobre arquiteturas onde a latência de comunicação é

fator determinante ao desempenho da aplicação. *Processos* e *canais de comunicação*, ao modo de OCCAM [124], são entidades primitivas. Assume-se a implementação no topo de uma biblioteca de passagem de mensagens, com vistas a portabilidade e eficiência. Vários mecanismos de alto nível são acrescentados ao modelo com vistas a possibilitar a geração de código eficiente, apesar do alto nível de abstração oferecido pela linguagem. Dentre estes, destaca-se o suporte a esqueletos topológicos parciais. O mundo dos processos no modelo # busca capturar a essência do comportamento conjunto dos processos que compõem um programa paralelo escrito por meio de uma biblioteca de passagem de mensagens. Virtualmente, qualquer linguagem poderia ser usada para o desenvolvimento dos módulos computacionais sob o ponto de vista do mundo das computações, embora a versão do modelo # implementada nesta tese suporte somente o uso da linguagem Haskell. O motivo para isso é que Haskell permite a ortogonalização entre os mundos de computação e de processos sem a necessidade de qualquer extensão, devido ao suporte a funções de alta ordem e *lazy evaluation*, favorecendo ainda a análise de propriedades formais de programas sob o nível de computação. Haskell pode ser vista como a linguagem ideal para a prototipação de módulos computacionais dentro do modelo #, fase anterior à implementação dos módulos propriamente ditos, escritos em linguagens apropriadas à funcionalidade provida pelo módulo.

A presente tese de doutorado pode ser vista como uma evolução e extensão ao produto da dissertação de mestrado desenvolvido pelo seu autor [47] e da tese de doutorado de Ricardo Lima [146]. Nestes trabalhos, embora a versão de Haskell$_{\#}$ apresentada atendesse aos objetivos iniciais a que se propunha, possuía limitações para uso geral em computação científica que não podiam ser ignoradas, principalmente no que diz respeito ao seu poder expressivo para especificar muitos dos padrões de interação comuns entre processos que compõem uma aplicação paralela, além de oferecer um limitado mecanismo de suporte a modularidade. Embora sua primeira versão de fato atendesse aos requisitos impostos a linguagem àquela época [148], era necessário evoluir seu modelo de paralelismo, suportando um nível de expressividade maior, no mínimo equivalente a expressividade de redes de Petri, para especificação de concorrência, mas mantendo a obediência a premissa da hierarquia de processos, com todas as consequências advindas do suporte a essa característica, incluindo a facilidade de implementação, portabilidade, eficiência e modularidade suportada pela linguagem. Esses aspectos foram tratados nesta tese, produzindo um modelo de programação a que se denominou #, abrindo caminho para que, em um futuro breve, extensões a implementação desta linguagem suportem módulos computacionais descritos em outras linguagens, alternativas a Haskell, como C, Fortran e Java, ou capazes de encapsular as funcionalidades de bibliotecas científicas de alto desempenho pré-existentes.

1.5 OBJETIVO GERAL

O objetivo da pesquisa que originou esta tese consiste no desenvolvimento de um arcabouço de alto nível, baseado na experiência advinda da evolução da linguagem Haskell$_{\#}$, para programação paralela distribuída, sob premissas induzidas pelo novo contexto surgido em computação de alto desempenho como consequência da disseminação

das tecnologias de *cluster computing* e *grid computing*, em especial o aumento do grau de complexidade e escala das aplicações e o maior número de potenciais usuários da tecnologia de programação paralela.

1.6 OBJETIVOS ESPECÍFICOS

O modelo # foi projetado segundo um conjunto de premissas, detalhadas no Capítulo 3, motivadas pelo novo contexto que tem se configurado em computação de alto desempenho desde a última década. A partir destas, delimitaram-se os objetivos específicos a serem alcançados na pesquisa que deu origem a esta tese, relatados a seguir:

- **Concepção da linguagem de configuração #**: a linguagem de configuração # materializa as abstrações primitivas introduzidas pelo modelo # para captura dos aspectos essenciais que caracterizam programas paralelos quando observados sob o ponto de vista do *mundo dos processos*. A equivalência com redes de Petri para descrição de padrões de interação concorrente deve ser considerada. Deve-se ainda oferecer suporte a técnicas avançadas de modularização de programas, como composição hierárquica e suporte a esqueletos, visando a construção de programas de grande escala e adptação às modernas técnicas de engenharia de grandes projetos de *software*;

- **Tratamento formal de programas # por meio de redes de Petri**: Um esquema de tradução automático de programas # para redes de Petri, visando permitir o uso de ferramentas de análise de propriedades e avaliação de desempenho que suportem este formalismo para análise formal de programas #. A modelagem de programas # por meio de redes de Petri deve suportar os aspectos essenciais que caracterizem a sua semântica. Exemplos devem ser considerados para demonstrar a eficácia do arcabouço proposto;

- **Implementação de Haskell$_{\#}$**: Deve ser implementada uma nova versão para a linguagem Haskell$_{\#}$, com suporte ao modelo # e empregando Haskell para descrição de computações. Supõe-se implementação no topo de MPI, voltada a *clusters*, tendo em vista portabilidade, desempenho e facilidade de implementação. Demonstra-se como o modelo # pode ser implementado usando ferramentas pré-existentes e consagradas, sem significativo comprometimento de desempenho;

- ***Bechmarks* para avaliação de desempenho**: o desempenho da implementação de Haskell$_{\#}$ devenvolvida deve ser avaliado, por meio de programas *benchmarks*, comparando a escalabilidade de desempenho obtida com a escalabilidade de versões puramente MPI dos programas considerados na avaliação;

- **Implementação de aplicações**: Estudos de caso de implementações de aplicações, reais e simuladas, devem ser consideradas para demonstrar diversos aspectos relevantes a cerca do estilo # de programação paralela, evidenciando as vantagens advindas de sua aplicação quando comparada a abordagens de mais baixo nível, em especial bibliotecas de passagens de mensagens, como MPI;

- **Implementação de um protótipo para VHT**: VHT (Visual # Tool) constitui a ferramenta de suporte ao desenvolvimento de programas paralelos, baseada no modelo #, onde deverão estar implementados os resultados obtidos pela pesquisa que resultou nesta tese. Neste trabalho, é desenvolvida uma versão protótipo para esta ferramenta, com a maioria de suas funcionalidades incluídas, em especial relativas a implementação visual das abstrações suportadas na construção de programas #;

1.7 ESTRUTURA DESTA TESE

Além desta introdução, este trabalho inclui mais sete (7) capítulos, cujos respectivos conteúdos são resumidos nos items que se seguem:

- **Capítulo 2:** Revisão de literatura, envolvendo os aspectos relacionados à área de processamento de alto desempenho relevantes para o desenvolvimento da pesquisa que deu origem a esta tese de doutorado;

- **Capítulo 3:** Descrição do modelo # de programação, onde são introduzidas as abstrações suportadas por este modelo visando a captura dos aspectos essenciais à configuração da execução paralela de processos, os quais constituem um programa paralelo, assumindo-se que estes comunicam-se por meio de passagem de mensagens em um ambiente distribuído. A linguagem de configuração # é uma materização para as citadas abstrações, sendo usada efetivamente para construção de programas #;

- **Capítulo 4:** Discussão sobre os artefatos e técnicas de programação usadas no desenvolvimento de programas #. A estrutura do capítulo busca explicar como o modelo # é capaz de suportar, de forma natural, as três principais técnicas de modularização de programas (composicional, esqueletos e aspectos), em um arcabouço único de programação. Discute-se ainda como descrições # de programas MPI podem ser derivadas a partir de seu código fonte. Diversos exemplos de programas # são apresentados com a finalidade de demonstrar a expressividade e habilidade do modelo de lidar com certas classes representativas de aplicações simples e complexas;

- **Capítulo 5:** Apresentação da tradução de programas # para redes de Petri [188], visando o suporte a análise de propriedades formais e avaliação de desempenho utilizando este formalismo. Esse capítulo pode ser ainda enxergado sob o ponto de vista de uma descrição formal para a semântica da linguagem de configuração #; Exemplos de análise de propriedades de programas com redes de Petri, a partir do esquema de tradução especificado, são apresentados, utilizando as ferramentas INA e PEP;

- **Capítulo 6:** Descrição da implementação da linguagem Haskell$_{\#}$, a versão do modelo # onde Haskell é usada para descrição de computações. Figuras de desempenho

usando um sub-conjunto do *NAS Parallel Benchmarks* [16] são ainda apresentadas com a finalidade de analisar a eficiência da implementação proposta;

- **Capítulo 7:** Conclusões do trabalho e propostas de novos trabalhos que devem dar continuidade pesquisa que deu origem a esta tese de doutorado.

CAPÍTULO 2

TÓPICOS EM COMPUTAÇÃO DE ALTO DESEMPENHO E PROGRAMAÇÃO PARALELA

Este capítulo introduz a base conceitual que inspirou e orientou as premissas que guiaram a concepção e evolução do modelo #, produto desta tese. Incialmente, são discutidos aspectos relevantes concernentes a aplicação e evolução da tecnologia de processamento paralelo, motivando as discussões subseqüentes sobre o paradigma de programação paralela e seu uso integrado ao emergente paradigma de coodenação, o qual inspira muitos dos conceitos desenvolvidos neste trabalho.

2.1 PROCESSAMENTO PARALELO

No âmbito da matemática, a tentativa em formalizar-se a noção de procedimento, conjunto de passos discretos orientado a realização de uma tarefa específica, motivou o surgimento de vários formalismos, dentre as quais destacam-se, em ordem cronológica, as funções recursivas de Gödel [104], o λ-calculus de Church [60] e a máquina de Turing [218, 219]. Os resultados obtidos decorrentes das pesquisas sobre estes formalismos levaram ao emergimento da teoria da computação como área de forte interesse na matemática. Entretanto, como ferramenta prática, a computação moderna recebeu seu impulso decisivo devido aos resultados obtidos pelo matemático húngaro John von-Neumman, radicado nos EUA, os quais levaram à concepção da primeira arquitetura de computador digital, a qual herdou seu nome [11]. Esta influencia até os dias atuais a tecnologia de construção dos computadores digitais.

Na arquitetura von-Neumann, os passos de um procedimento são descrito por meio de instruções realizáveis mecanicamente. Em máquinas de uso prático, a execução de cada instrução é realizada em um tempo finito e não nulo. Podemos assim caracterizar o *tempo de execução* de uma sequência de instruções como a soma dos tempos gastos para executar cada uma destas. Um programa de computador, segundo a arquitetura von-Neumman, pode ser então definido como uma sequência de instruções, ordenadas totalmente segundo um fluxo de controle para realização uma tarefa para a qual foi especificado.

Nas últimas seis décadas, o emprego de computadores disseminou-se com sucesso em diversas áreas de aplicação. O surgimento constante de novas aplicações e o aumento da complexidade das aplicações existentes, induzida pela sempre presente necessidade de cobrirem-se instâncias maiores de um mesmo problema, motiva a onipresente demanda por computadores cada vez mais poderosos. Essa demanda tem motivado as políticas agressivas de investimento tecnológico e científico das grandes indústrias de computação, as quais só são possíveis devido ao aumento da escala de mercado para os computadores resultante do surgimento de novos usuários e aplicações. Maior escala de mercado implica em maiores faturamentos, permitindo maior diluição e aumento dos investimentos em

ciência e tecnologia, os quais resultam em computadores mais poderosos e na conseqüente inspiração para o surgimento de novas aplicações. Esse processo retro-alimentativo justifica o crescimento vertiginoso do mercado de computação e sua tecnologia nas últimas décadas.

Dentre as aplicações que emergiram para os computadores, um subconjunto especial destas caracteriza-se por uma demanda crítica por maior velocidade de processamento, sobretudo voltadas à computação científica e simulações em indústrias de grande porte, envolvendo modelos matemáticos complexos. Essas aplicações motivaram o surgimento e evolução das tecnologias de processamento de alto desempenho, a partir de onde emergiram os conceitos de supercomputação e processamento paralelo.

Relembrando a arquitetura original proposta por von-Neumman, esta pressupõe a existência de uma linha sequencial de instruções, onde cada uma destas é executada por vez. Uma maneira óbvia de tornar um computador mais veloz é torná-lo então capaz de executar cada instrução individual de maneira mais rápida, empregando novas tecnologias e materiais na construção de processadores sequenciais. Entretanto, limitações físicas têm tornado proibitiva a relação custo/benefício associada ao uso desta abordagem [112, 174, 181, 123].

Uma outra forma, atualmente economicamente e tecnologicamente viável, para o incremento da velocidade de processamento em arquiteturas computacionais baseadas na arquitetura von-Neumann consiste no emprego da tecnologia de processamento paralelo, o qual consiste em enfraquecer a suposição do modelo von-Neumann de que as instruções devem estar *totalmente* ordenadas segundo um fluxo de controle. Segundo a abordagem paralela, estas podem estar ordenadas de maneira *parcial*, o que implica na possibilidade de execução simultânea de instruções em várias unidades independentes de processamento. Essa tecnologia pode ser aplicada em diferentes níveis de abstrações em uma arquitetura computacional, utilizando abordagens bastante peculiares, geralmente assumindo suposições intimamente relacionadas à organização física da arquitetura. As primeiras arquiteturas paralelas foram propostas ainda no final da década de cinqüenta. As próximas seções elucidam aspectos relevantes concernentes a tecnologia de processamento paralelo, seguindo-se a motivação do foco principal deste trabalho, referente à paradigmas de alto nível voltados à programação paralela explícita sobre *clusters*.

2.1.1 Hierarquias de Paralelismo

Em arquiteturas de computador, o processamento paralelo pode ser empregado sob três níveis, os quais diferenciam-se pelo nível de abstração em relação a arquitetura física e o grau de transparência da tarefa de configuração da execução paralela em relação ao programador:

- **Nível Físico**. O paralelismo é explorado ao nível da organização interna dos processadores, os quais podem ser enxergados como caixas-pretas que executam computações de forma aparentemente sequencial. Técnicas como o uso de *pipe-lines*, mútiplas linhas de instruções e execução simultânea especulativa de instruções são exemplos de tecnologias comumente usadas para este fim em processadores superescalares[123]. O emprego do processamento paralelo sob este nível tem impulsion-

ado o rápido crescimento, verificado desde a última década, no desempenho dos computadores pessoais e estações de trabalho;

- **Nível de Suporte**. Adimite-se a existência de várias unidades físicas sequenciais de processamento. A configuração da execução paralela é realizada automaticamente por ferramentas de suporte às aplicações. Exemplos de uso do paralelismo a este nível incluem os compiladores capazes de gerar código paralelo para um programa escrito em uma linguagem sequencial (paralelismo implícito), como GpH [215], sistemas operacionais capazes de explorar as funcionalidades de sistemas multi-processados, como Linux, Windows NT e variantes, e camadas intermediárias de *software*, conceito conhecido como *middleware*, em sistemas de *cluster computing*, capazes de abstrair a rede de comunicação das aplicações, distribuindo a computação implicitamente entre os nós de processamento [20];

- **Nível de Aplicação**. O paralelismo é explicitamente configurado como parte da tarefa de programação das aplicações. Linguagens de programação paralela, como HPF (*High Performance Fortran*) [86], ou bibliotecas de passagem de mensagens que extendem as funcionalidades de linguagens sequenciais de uso disseminado, como PVM [97] e MPI [73], são geralmente empregadas para esse fim. O programador é responsável pelas tarefas de particionamento da aplicação em processos paralelos, sincronização destes e sua alocação nas unidades de processamento da arquitetura paralela, com preocupações relacionadas ao balanceamento da carga de trabalho nestas unidades.

Facilmente conclui-se que os três níveis de paralelismo podem coexistir em um mesmo sistema, definindo *hierarquias* de suporte ao processamento paralelo. Além disso, os níveis de aplicação e suporte podem sobrepôr-se, especialmente no uso de linguagens paralelas semi-explícitas, onde certas tarefas de configuração do paralelismo são delegadas ao compilador e outras ao programador. O foco desta pesquisa concentra-se nas técnicas de exploração do paralelismo em nível de *aplicação*.

2.1.2 Arquiteturas de Processamento Paralelo

Apesar de sua conceituação simples, o processamento paralelo tem motivado, nas últimas décadas, grande número de pesquisas visando o desenvolvimento de tecnologias para o seu suporte. Surgiram então uma rica variedade de arquiteturas paralelas, com características peculiares [123]. Essa diversidade motivou o aparecimento de várias caracterizações e classificações para arquiteturas paralelas. A mais aceita e difundida foi proposta por M. J. Flynn ainda na década de sessenta [84], segundo a qual arquiteturas paralelas podem pertencer a uma das seguintes classes:

- **SIMD** (*Single Instruction Multiple Data*): Nesta categoria estão incluídos os processadores vetorais, os quais durante muitos anos dominaram o *estado-da-arte* da computação paralela. Estes suportavam um conjunto de instruções específicas capaz de operar simultaneamente sobre um conjunto de dados. Embora seu uso tenha

se tornado menos comum a cada ano, arquiteturas vetoriais de supercomputação ainda são utilizados em muitos centros de pesquisa e indústrias. Recentemente, tecnologias SIMD tem sido empregadas em processadores empregados em computadores pessoais (Intel Pentium e variantes e IBM Power PC G4), com a finalidade de acelerar a execução de aplicações multimídia, como jogos e edição de vídeo [209, 37];

- **MIMD** (*Multiple Instruction Multiple Data*): Supõe-se a execução simultânea de multiplas linhas de instrução, as quais podem operar sobre vários conjuntos de dados. Atualmente, é comum a utilização de um conjunto de processadores sequenciais, cada um com sua linha de instrução independente, os quais podem compartilhar o mesmo espaço de endereçamento (arquiteturas de *memória compartilhada*) ou não (arquiteturas de *memória distribuída* ou *multicomputadores*). No primeiro caso, a comunicação entre os processos paralelos é realizada através de *variáveis compartilhadas*, enquanto no segundo, o mecanismo mais adotado tem sido os de *passagem de mensagens* e *chamada de procedimento remoto*.

Além desta, a qual tornou-se clássica, há outros tipos de classificações para arquiteturas paralelas que merecem atenção. Cita-se [77] como referência para taxonomias de arquiteturas paralelas.

Observando a lista dos 500 supercomputadores mais velozes da atualidade, observa-se que estes enquadram-se em quatro grandes classes:

- **Máquinas Massivamente Paralelas (MPP's)**: correspondem aos supercomputadores contemporâneos, constituídas por uma grande quantidade de unidades de processamento distribuídas (normalmente em quantidade variando entre 128 e 1024), as quais comunicam-se por meio de uma rede dedicada de alto desempenho. MPP's são arquiteturas proprietárias, que em geral utilizam a tecnologia de processador utilizado em estações de trabalho do fabricante. Exemplos de MPP's atualmente no mercado incluem HP SPP, IBM SP, SGI Origin, Cray T3E/T3D, Fujitsu VPP e Cray X1;

- **Clusters (NOW's)**: redes de estações de trabalho ou computadores pessoais interligadas por meio de interfaces de rede convencionais, como Ethernet, Myrinet ou SCI (*Scalable Coherent Interface*), possivelmente equipadas com protocolos de comunicação comuns, como TCP/IP, ou voltados a computação de alto desempenho, como VIA [210]. Clusters oferecem um excelente relação custo/benefício em relação aos supercomputadores baseados na tecnologia das MPP's, tendo se tornado populares em um grande número de aplicações na indústria e pesquisa científica, onde antes somente os custosos supercomputadores eram aparentemente capazes de fazer frente a suas demandas computacionais. Esse contexto configurou-se a partir de meados da década de 90 com o sucesso do projeto Bewoulf [30], desenvolvido na NASA por Donald Becker;

- **Computadores com memória compartilhada (multi-processadores)**: nestes, as unidades de processamento, independentes, sincronizam por meio de um espaço

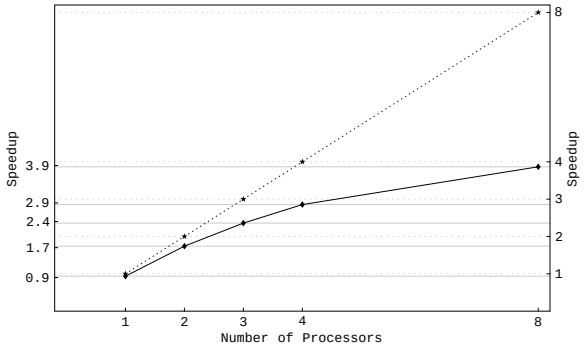


Figura 2.1. Exemplo de curva de *speedup para o programa MCP-Haskell*$_{\#}$

de endereçamento comum. Assim como *clusters*, possuem uma boa relação custo/benefício, podendo ser construídos com *hardware* de prateleira. Entretanto, sua pouca escalabilidade, resultante da limitação ao número de processadores que podem acessar um mesmo espaço de endereçamento de forma eficiente, devido especialmente ao conhecido efeito de contenção de memória, limita seu uso a uma classe restrita de aplicações;

- **Constelações**: podem ser entendidas como *clusters* formados por computadores multi-processados, onde cada nó de processamento possui pelo menos 16 processadores. O paralelismo é explorado de forma hierárquica, tomando-se proveito das vantagens advindas do uso de *clusters* e multi-processadores, adequando-se portanto a uma classe mais rica de aplicações, embora a um custo relativamente mais alto;

É importante ressaltar a ausência de arquiteturas vetoriais (SIMD) de supercomputação na lista dos 500 mais poderosos computadores comtenporâneos. Este fato tem sido observado desde meados da última década, sugerindo que a popularização dos processadores super-escalares e a evolução acelerada do desempenho destes tem tornado obsoleta esta outrora tão difundida tecnologia. Entretanto, tecnologias vetoriais vêm sendo empregadas com sucesso na arquitetura interna de processadores super-escalares disponíveis no mercado, em especial visando melhorar o desempenho destas arquiteturas para aplicações multimídia, jogos e tratamento de imagens [209, 37].

2.1.3 Medidas de Desempenho

Face a necessidade de meios de analisar-se e comparar-se o desempenho de programas e arquiteturas paralelas, surgiram algumas medidas padrões para esta finalidade. A seguir, definiremos as duas mais comuns: *speedup* e *eficiência*, as quais serão usadas posteriormente.

2.1.3.1 Speedup O *speedup* mede a relação entre o tempo de execução de uma execução sequencial de um programa e o tempo de execução da versão paralela em n pro-

cessadores. Seja T_n o tempo de execução de um programa paralelo sobre n processadores. Algumas definições comuns para *speedup* (S_n) são:

i) $S_n = \frac{T_1}{T_n}$, onde T_1 é o tempo de execução do programa paralelo executado sobre 1 processador, medido em um dos processadores do computador paralelo;

ii) $S_n = \frac{T_s}{T_n}$, onde T_s é o tempo de execução do algoritmo sequencial mais eficiente para o problema, medido em um processador sequencial;

iii) $S_n = \frac{T'_1}{T_n}$, onde T'_1 é o tempo de execução do algoritmo sequencial mais eficiente para o problema, medido em um dos processadores do computador paralelo.

Espera-se que o valor do *speedup* assuma um valor em um intervalo aberto de 1 a n, onde n é o número de processadores. Se o valor do *speedup* é menor que 1, então a versão sequencial do programa é mais eficiente que sua versão paralela. Embora seja esta uma situação não esperada para um programa paralelo, ocorre quando a quantidade de processos paralelos é suficientemente grande para que o tempo gasto em seu gerenciamento (instanciação e sincronização), seja significativo em relação ao tempo de computação propriamente dito. Tal comportamento também é comumente observado quando o número de processadores é inadequadamente pequeno para explorar o paralelismo. Em geral, o algoritmo paralelo é menos eficiente que o sequencial quando executado em 1 ou 2 processadores. Um *speedup* ideal (linear) é alcançado quando o valor do speedup é igual a n. Neste caso, o tempo de execução do programa sequencial é igualmente distribuído entre os processadores, sem que hajam perdas relativas ao gerenciamento do paralelismo. Embora teoricamente o *speedup ideal* seja hipotético, na prática certas circustâncias especiais podem permitir que este seja alcançado ou até mesmo superado (*speedup super linear*). Isso é possível quando utilizamos a Definição 2. Imagine que o particionamento da aplicação permite que cada processo paralelo (dados e código) caiba dentro de uma cache de altíssima velocidade em um processador. O tempo ganho com um gerenciamento de memória mais eficiente pode compensar o tempo gasto com o gerenciamento do paralelismo, gerando um ganho em desempenho em relação a versão sequencial. Esta é uma situação atípica, onde a sobrecarga sobre a computação gerada pelo gerenciamento da memória na execução da versão sequencial é maior que a sobrecarga na computação gerada pelo gerenciamento do paralelismo e pelo gerenciamento de memória da versão paralela.

2.1.3.2 Eficiência Outra medida comum é a *eficiência* (E_n), informalmente definida como a fração de tempo que os processadores usam durante a execução para computação efetiva.

$$E_n = \frac{S_n}{n} \times 100\%$$

A título ilustrativo, um programa com speedup linear possui eficiência de 100%, o que equivale a dizer que os processadores estiveram ocupados com computação efetiva

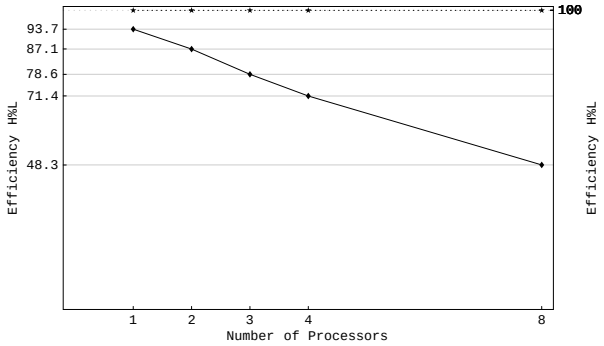


Figura 2.2. Exemplo de curva de eficiência para o programa MCP-Haskell$_{\#}$

durante todo o tempo de execução. Ou seja, não houve tempo gasto com sincronização e controle de processos.

Nas Figuras 2.1 e 2.2, apresentamos as curvas de speedup e eficiência de uma aplicação real construída com Haskell$_{\#}$, linguagem que será tratada nos capítulos posteriores. Na Figura 2.1, compare a curva de *speedup* obtida com a curva de *speedup* linear (linha pontilhada). O programa MCP-Haskell$_{\#}$ foi executado repectivamente em 1, 2, 4, e 8 processadores, sendo os tempos comparados aos obtidos por uma versão sequencial do mesmo.

2.1.3.3 A Lei de Amdahl: Limites Teóricos do Speed-up Em 1967, Gene Amdahl propôs que, em um programa paralelo, o *speedup* estaria limitado pela existência de uma porção não-paralelizável de código na versão sequencial do programa, chamada porção (inerentemente) *sequencial*. Assim, Seja P um programa paralelo. Podemos definir s e p como, respectivamente, as frações *ineremente sequencial* e paralelizável de P. Obviamente $s+p=1$. Segundo Amdahl, o *speedup máximo* que poderia ser obtido nessas circunstâncias seria definido pela seguinte fórmula:

$$S_n^{MAX} = \frac{1}{s+\frac{p}{n}}, \text{ onde n é o número de processadores;}$$

Note que quando o número de processadores tende ao infinito, o *speedup máximo* tende ao valor $\frac{1}{s}$, o qual é considerado como o limite teórico do *speedup*, segundo Amhdal.

$$lim_{n\to\infty} S_n^{MAX} = \frac{1}{s}$$

Fundamentado em dados experimentais, no final da década de 80, Gustafson [106, 105] propôs uma reformulação na Lei de Amhdal, baseado na idéia de que a proporção entre as porções sequencial e paralelizável de um programa em geral varia de acordo com o tamanho do problema a ser resolvido. Segundo Gustafson, em situações reais, o programador não fixa o tamanho do problema para execução em quantidades diferentes de processadores, o que seria uma suposição válida em uma pesquisa acadêmica. Na prática, o tamanho do problema aumenta de acordo com os recursos disponíveis no sistema (número

de processadores), visando obter, por exemplo, resultados mais acurados em uma simulação numérica de equações diferenciais parciais não-lineares. Neste caso específico, o tamanho do problema poderia ser a resolução do *grid* (número de pontos no espaço e/ou tempo) em uma aproximação por diferenças finitas. Ainda segundo Gustafson, uma formulação mais realista para o *speedup*, assumiria fixo o tempo de execução. Surgiram então duas definições alternativas de *speedup*: *scaled speedup* e *fixed-time speedup* [106], os quais têm a característica de não impôr limites ao *speedup*.

2.2 PROGRAMAÇÃO PARALELA

Nesta seção, são discutidos aspectos relevantes com respeito à exploração do paralelismo em nível de aplicação, onde o programador é responsável pela configuração explícita da execução paralela de um programa, como parte inerente à tarefa de programação. Para isso, são empregadas linguagens paralelas ou bibliotecas de suporte, as quais extendem as funcionalidades de linguagens sequenciais conhecidas com o suporte ao paralelismo. Em [204] é apresentado um *survey* bastante completo a respeito da caracterização das diversas linguagens e bibliotecas de paralelismo disponíveis na atualidade.

Em geral, modelos de programação paralela têm surgido em íntima associação com arquiteturas paralelas particulares, de forma a suportar suas características peculiares e obter o máximo de seu desempenho. Enquanto podemos afirmar, com pequena margem de erro, que em um computador sequencial a velocidade de execução de um programa é diretamente proporcional a capacidade de processamento do processador, em arquiteturas paralelas tal suposição nem sempre é verdadeira, mesmo quando são comparadas arquiteturas com capacidades de processamento semelhantes. Isto se deve ao fato de que outros fatores devem ser considerados ao analisar-se o desempenho de uma aplicação sobre arquiteturas paralelas, além da velocidade de suas unidades de processamento. Dentre estes, destacam-se grau de heterogeneidade entre as unidades de processamento, sua organização topológica e as características de desempenho da rede de comunicação que as interliga. As seções que se seguem discutem aspectos relevantes a respeito da tarefa de programação paralela, notadamente em sua abordagem explícita.

2.2.1 Transparência e Dinâmica do Paralelismo

Em relação à transparência do paralelismo exposto ao programador, modelos de programação paralela podem assumir abordagens que variam desde aquelas totalmente *implícitas*, onde a execução paralela é resolvida, em tempo de compilação e/ou execução, sem interferência do programador, até aquelas totalmente *explícitas*, onde o programador é responsável pela *configuração do programa paralelo*, tarefa que envolve o *particionamento* do problema, agregação de tarefas (controle de granularidade), *sincronização* das tarefas paralelas, e o *balanceamento de carga* dos processadores, com vistas a minimizar o tempo de ociosidade destes durante a execução. Entre essas duas abordagens extremas existem abordagens intermediárias, como as que permitem ao programador definir *anotações* que auxiliam o compilador na tarefa de configuração. Abordagens desta natureza são conhecidas como *semi-explícitas* ou *semi-implícitas* [204].

Modelos de linguagens paralelas podem ainda ser classificados como *estáticos* ou *dinâmicos*. O paralelismo dinâmico envolve mecanismos que modificam a *configuração do programa paralelo* em tempo de execução, como forma de adaptação a novas condições de carga em uma rede de computadores, por exemplo, enquanto o paralelismo estático envolve mecanismos que supõem a configuração do paralelismo em tempo de compilação, permanecendo esta inalterada durante a execução do programa.

Implementações de modelos implícitos de programação paralela são, em geral, ineficientes quando comparadas a implementações de modelos *explícitos*. Esse fato é consequência das dificuldades computacionais conhecidas para os problemas envolvidos na configuração automática de programas para execução paralela ótima, em suas instâncias gerais. Dentre esses problemas, incluem-se o particionamento da aplicação, o balanceamento dinâmico de carga e o controle de granularidade [96, 76]. Na prática, assumem-se certas condições que permitem o uso de algoritmos e heurísticas aplicados a instâncias específicas do problema, sem que haja garantia de obtenção de resultados ótimos em condições gerais.

Em certas classes de arquiteturas, em especial quando o grau de acoplamento entre as unidades de processamento é pequeno, o uso de mecanismos dinâmicos de gerenciamento do paralelismo pode causar uma sobrecarga capaz de tornar proibitivo seu uso. Particularmente, está associado aos mecanismos dinâmicos o controle do balanceamento de carga entre os processadores, possivelmente com suporte à migração de processos e a reconfiguração dinâmica da topologia da rede de comunicação que descreve a interação entre os processos. A reconfiguração dinâmica torna árduo, ou virtualmente impossível, o tratamento formal de programas paralelos, visando análise de propriedades estruturais de programas e análise formal de custos, uma vez que não é possível prever as alterações na topologia da rede de processos que constituem o programa, durante sua execução.

2.2.2 Balanceamento de Carga

Em um modelo ótimo de execução paralela, a capacidade máxima das unidades de processamento é utilizada durante todo o tempo de execução de um programa paralelo, não havendo portanto momentos de ociosidade para alguma destas. Na prática, entretanto, a estrutura irregular de algumas classes de programas paralelos, a heterogeneidade ou irregularidade da arquitetura paralela, ou mesmo a dificuldade em alocar-se uma certa aplicação, mesmo regular, em uma certa arquitetura, mesmo homogênea, tornam difícil a obtenção de um balanceamento ótimo. Surge assim a necessidade de mecanismos de balanceamento capazes de minimizar a ociosidade dos processadores, alocando tarefas aos mesmos.

O balanceamento de carga pode ser realizado *explicitamente*, pelo programador, ou *implicitamente*, pelo compilador ou sistema em tempo de execução. Mecanismos implícitos de balanceamento são de difícil trato computacional. As soluções existentes na literatura são simplificações aplicadas a instâncias restritas do problema geral, normalmente definidas a partir da suposição de certas características do programa paralelo ou arquitetura alvo, nem sempre aplicadas a maioria das situações. Os melhores resultados são obtidos quando a tarefa de balanceamento é delegada ao especialista humano, principal-

mente quando este é conhecedor das características que afetam o desempenho tanto da aplicação quanto do ambiente de execução.

Podemos ainda classificar os mecanismos de balanceamento como *estáticos* ou *dinâmicos*. Técnicas estáticas atuam satisfatoriamente em aplicações que possuem um padrão regular de interação entre processos e assumem a execução em arquiteturas homogêneas. Por outro lado, quando quaisquer dessas suposições não é válida, empregam-se mecanismos dinâmicos. Entretanto, estes últimos são capazes de gerar uma considerável sobrecarga na execução de um programa paralelo, principalmente devido a possibilidade da migração de processos entre processadores, de forma a equilibrar suas cargas de trabalho. Em arquiteturas onde a latência de comunicação é alta, este recurso pode gerar uma sobrecarga proibitiva na execução do programa, anulando qualquer ganho de desempenho obtido por uma melhor distribuição da carga de trabalho. A migração de tarefas pode ser realizada através de dois mecanismos básicos:

- **Processos Ativos.** O processo é responsável por requisitar trabalho de um processo sobrecarregado, sempre que torna-se ocioso;
- **Processos Passivos.** Um sistema gerenciador controla o balanceamento da carga de trabalho das unidades de processamento, distribuindo ou redistribuindo as tarefas paralelas entre os processadores sempre que necessário. Outra possibilidade é o próprio processo sobrecarregado encarregar-se de dividir seu excedente de tarefas entre processadores ociosos.

2.2.3 Particionamento, Granularidade e Escalabilidade

Além do reconhecimento das sequências de código sequencial que podem ser executadas em paralelo (particionamento), a paralelização de um programa envolve a definição da granularidade de paralelismo, ou seja, o tamanho relativo destas sequências, processo conhecido na literatura como *agregação* [88]. Podemos então encontrar diversos níveis de granularidade em programas paralelos, de forma que quanto mais *grossa* a granularidade de um programa maior é a relação entre o tempo de execução das unidades sequenciais individualmente e o tempo de execução da aplicação paralela como um todo. A granularidade tem um papel preponderante no desempenho de uma aplicação paralela, sendo influenciada pela arquitetura alvo considerada. Quanto mais *fina*, maior o número de processos paralelos e consequentemente maior será a quantidade necessária de comunicação para sincronizá-los. Além de possuir uma quantidade maior de unidades de processamento para suportar maior número de tarefas paralelas, uma arquitetura que suporta paralelismo de fina granularidade de forma eficiente deve prover mecanismos bastantes eficazes de comunicação entre tais unidades, evitando assim sobrecarga de comunicação excessiva, resultante da interação entre os processos, sobre o custo global de execução do programa. O grau de granularidade potencial de um programa paralelo é inerente às suas características intrínsecas. Entretanto, a escolha do grau ótimo depende de dois fatores: a técnica de particionamento utilizada e as características da arquitetura alvo, como descrito anteriormente. Existem basicamente duas técnicas de particionamento:

- **Decomposição funcional** (*paralelismo funcional*): As unidades paralelizáveis são escolhidas distingüindo-se, no programa, unidades funcionais que podem ser agrupadas em tarefas para execução paralela. Esse é um tipo de paralelismo comum em arquiteturas MIMD;

- **Decomposição de domínio** (*paralelismo de dados*): O domínio de dados sobre o qual o programa ou algumas de suas funções atuam é sub-dividido em várias partes para que sejam processadas em paralelo. Esse é o tipo de paralelismo típico em arquiteturas SIMD, particularmente as chamadas arquiteturas vetoriais. Em arquiteturas MIMD, esse tipo de paralelismo pode ser implementado pelo uso do modelo de programação SPMD (*Single Program, Multiple Data*), onde uma cópia de um único programa é disparada em cada unidade de processamento, atuando cooperativamente no processamento de um conjunto de dados em comum. O paralelismo de dados é uma forma bastante eficiente e simples de explorar o paralelismo em uma classe rica de aplicações de alto desempenho, em geral com *speedup* próximo ao linear.

Utilizando decomposição de domínio, o grau de granularidade potencial varia dependentemente do tamanho do domínio a ser paralelizado. Assim, considerando-se um certo domínio, quanto maior a quantidade de processadores disponíveis mais fina será a granularidade. Por outro lado, admitindo um certo grau de granularidade, quando aumentamos o número de processadores podemos aumentar também o tamanho do domínio, de forma a utilizar de maneira mais efetiva o maior poder computacional disponível para obtenção de soluções para instâncias mais complexas do problema ou aumentar a acurácia dos resultados. Isso mostra a maior *escalabilidade*, ou capacidade de melhor explorar os recursos disponíveis no ambiente, suportada com a adoção do paralelismo de dados. Esta qualidade mostra-se particularmente útil em aplicações onde a qualidade ou precisão do resultado é diretamente proporcional ao tamanho do domínio processado, como por exemplo na simulação de campos petrolíferos através de métodos de discretização numérica (*diferenças finitas*, *elementos finitos* [56] ou *volumes finitos* [224]). Por outro lado, a decomposição funcional é bem mais restritiva. O conjunto de tarefas paralelas varia de acordo com a semântica da aplicação, a qual é invariante. O paralelismo funcional em geral está associado ao paralelismo de granularidade grossa ou média.

Em abordagens implícitas de paralelismo, em nível de suporte, vale ressaltar a existência das técnicas de particionamento de controle, como aquelas empregadas na paralelização implícita de programas funcionais, pela avaliação paralela de expressões [44], ou paralelização de *loops* em programas imperativos[22]. Estas estão associadas ao paralelismo de muito fina granularidade, em geral pouco eficiente em arquiteturas fracamente acopladas. Particularmente no caso de paralelização de *loops* em linguagens imperativas, técnica bastante difundida em arquiteturas vetoriais, algoritmos complexos são empregados pelo compilador para sua obtenção. Entretanto, embora implícito, requer habilidade do programador na construção de loops "paralelizáveis", muitas vezes necessitando do uso de anotações no código.

A busca pelo grau ótimo de granularidade é uma tarefa considerada difícil de ser realizada através de um compilador (*estaticamente*) ou um sistema em tempo de execução

(*dinamicamente*). Esta última abordagem, útil em um ambiente onde o desempenho dos nós pode variar temporalmente, como em redes de computadores, pode gerar uma sobrecarga considerável sobre a computação do programa, devido a execução do mecanismo para controle dinâmico da granularidade em paralelo com a aplicação em si. Além disso, não é garantido obter-se os resultados esperados, devido a dificuldade inerente aos problemas de difícil trato computacional envolvidos, fato que motiva simplificações com perda de generalidade com o objetivo de que o mecanismo seja implementado de forma eficiente.

2.3 QUAL O MODELO IDEAL PARA PROGRAMAÇÃO PARALELA ?

Apesar de ineficiente, o paralelismo implícito tem a vantagem de tornar transparente para o programador a tarefa de configuração do programa paralelo, fazendo com que a tarefa de programação paralela iguale-se em nível de dificuldade à programação sequencial. Na programação paralela explícita, o programador, além de lidar com a programação sequencial das tarefas paralelas, deve sincronizá-los e alocá-los a processadores de forma conveniente. A programação paralela é portanto uma tarefa inerentemente mais difícil que a programação sequencial.

Abordagens *explícitas*, *estáticas* e com granularidade média ou grossa têm conseguido os melhores resultados em relação ao desempenho de programas paralelos. Além disso, tem se mostrado mais viável facilitar a tarefa de programação paralela explícita, do que tornar mais eficiente o paralelismo explorado em nível implícito. O grande desafio tem sido encontrar um modelo padrão de programação para esta abordagem. Em geral, é difícil associar mecanismos de *abstração* e *modularidade* de alto nível com *eficiência* de maneira ótima e sob um arcabouço comum. Este trabalho fundamenta-se na idéia de que um modelo ideal de programação paralela suportaria intrinsecamente as seguintes características:

- *abrangência* e *expressividade*, com o intuito de capturar-se a essência dos padrões conhecidos de paralelismo;
- *eficiência* e *portabilidade*, permitindo que um mesmo programa seja executado em várias arquiteturas paralelas com o aproveitamento máximo de seus respectivos potenciais de desempenho;
- existência de mecanismo de alto nível que suportem os conceitos de *abstração e modularidade* requeridos para construção de aplicações paralelas de larga escala, motivando fortemente o reuso de componentes;
- *simplicidade*, de forma a tornar a prática da programação paralela acessível ao programador comum. A ortogonalização entre as tarefas de especificação das computações e configuração do paralelismo é uma forma efetiva para alcançar-se tal propósito;
- possibilidade de *análise formal de programas*, utilizando formalismos pré-existentes, como CCS (*Calculus of Communicating Systems*) [169], redes de Petri [188], CSP (*Communicating Sequential Processes*) [115], dentre outros;

- existência de um *modelo de custos*, o qual possibilite analisar ou simular os custos da execução de programas paralelos sobre diferentes arquiteturas, bem como os custos inerentes a utilização de um certo padrão de paralelismo no particionamento de uma aplicação, evitando-se medições diretas;

2.4 PROCESSAMENTO PARALELO EM LINGUAGENS FUNCIONAIS

Linguagens funcionais constituem um importante paradigma de programação, a partir do qual muitos dos desenvolvimentos hoje aplicados em outras classes de linguagens tem buscado inspiração. A pesquisa desenvolvida nesta tese tem forte influência neste paradigma, sendo a linguagem Haskell$_{\#}$ uma extensão paralela para uma linguagem funcional. Tendo isso em vista, discutimos nesta seção a aplicação da tecnologia de processamento paralelo no contexto do paradigma funcional.

2.4.1 O Paradigma de Programação Funcional

O paradigma de programação funcional emergiu, a partir da década de sessenta, devido ao surgimento de linguagens cujo modelo computacional baseava-se no λ-calculus [60], um formalismo matemático, desenvolvido por Alonzo Church na década de trinta, capaz de capturar a noção de função computável através de passos mecânicos e discretos. Outros formalismos conhecidos para o mesmo propósito são as *máquinas de Turing* [218, 219] e as *funções recursivas* de Gödel [104]. Embora hajam controvérsias a respeito, LISP [166] é considerada a primeira linguagem funcional. Contudo, LISP não é considerada uma linguagem funcional pura, por suportar *atribuições destrutivas*.

Desde seu surgimento, o paradigma funcional tem motivado o desenvolvimento de várias linguagens [118, 153]. Entretanto, somente a partir do final da década de setenta, devido a *Turing Award Lecture* de John Backus [15], em 1978, estas linguagens passaram a ser vistas com maior notoriedade. Naquela época, Backus alertava a comunidade da computação sobre a necessidade de substituir o paradigma imperativo, derivado da arquitetura von-Neumann, pelo paradigma funcional, com o objetivo de extingüir o contexto que formou-se com a *crise do software* [192][1]. Defendeu então idéias que posicionavam o paradigma funcional em um patamar tecnológico superior ao paradigma imperativo, arguindo então serem linguagens funcionais alternativas reais às linguagens imperativas. Dentre as características positivas destas linguagens, podemos citar: o alto nível de abstração, a ausência de *atribuições destrutivas*, a garantia de *transparência referencial*, a transparência do fluxo de controle nos programas, a independência de arquitetura (portabilidade), a melhor interpretação do código e os novos mecanismos para facilitar a construção de programas modulares, como *funções de alta ordem* e *avaliação procrastinada* (*lazy evaluation*) [122]. Dentre outras vantagens, essas características permitem o raciocínio sobre programas visando construção de *software* confiáveis, a transformação de programas, visando otimização de código [198], e a exploração de um tipo de paralelismo implícito inerente à própria linguagem.

[1] O custo do *software* tornara-se substancialmente mais alto que o do *hardware*, com projeções desastrosas.

Entretanto, a implementação eficiente de linguagens funcionais, tanto no que diz respeito a tecnologia de compilação quanto ao projeto de sistemas em tempo de execução, tem se constituído em um desafio difícil de ser superado. Este fato decorre do alto nível de abstração característico dessas linguagens, sobretudo em relação à arquiteturas específicas de computação. Linguagens imperativas, por exemplo, foram desenvolvidas inspiradas nas peculiaridades da arquitetura *von-Neumann*, o padrão *de facto* vigente nos dias atuais para computadores, sendo implementadas de forma extremamente eficiente sobre esta plataforma. Essa realidade gerou durante certo tempo um desapontamento da comunidade científica com os resultados obtidos com a aplicação de linguagens funcionais. Entretanto, desde a última década, as pesquisas voltadas para implementação eficiente destas linguagens têm evoluído bastante. Resultados animadores, em relação ao desempenho de programas funcionais comparado ao desempenho de programas correspondentes escritos em C e FORTRAN, já foram obtidos [111]. Simultaneamente, novos mecanismos de estruturação e construção de programas têm surgido, com vistas a tornar estas linguagens mais práticas e poderosas.

2.4.2 Haskell

Haskell [121] é uma linguagem funcional pura, não estrita e de alta ordem que vem sendo desenvolvida por um comitê científico desde 1987. Nesta época, estabeleceu-se na comunidade científica o consenso de que a evolução das linguagens funcionais puras e não-estritas estava sendo emperrada pela ausência de uma linguagem de uso comum, sobre a qual as novas pesquisas deveriam concentrar-se. Desde então, *Haskell* tem se tornado um padrão para pesquisa e desenvolvimento em linguagens funcionais e para construção de programas em diversas áreas de aplicação. Vários compiladores seqüenciais e paralelos encontram-se disponíveis para essa linguagem [102, 195, 14].

Haskell tem um papel preponderante no processo de popularização e no desenvolvimento de pesquisas que visam a implementação eficiente de linguagens funcionais, tendo incorporado as mais recentes inovações oriundas da pesquisa sobre estas linguagens, incluindo *funções de alta ordem*, *semântica não-estrita*, *tipos algébricos* definidos pelo usuário, suporte estático aos polimorfismos paramétrico e *ad hoc*, *casamento de padrões*, *compreensão de listas* (*list comprehesions*), sistema de *módulos*, *mônadas* e um rico conjunto de tipos primitivos, incluindo *arrays*, inteiros de precisão fixa e arbitrária e números de ponto flutuante. Hoje, pode-se afirmar seguramente que a evolução da linguagem *Haskell* coincide com a evolução da tecnologia de programação funcional, o que posiciona esta linguagem, por definição, como o *estado-da-arte* desta tecnologia.

2.4.3 Linguagens Funcionais e Processamento Paralelo

A história da programação funcional paralela compreende dois períodos [152]. No primeiro, o qual durou até meados da década de oitenta, o paralelismo era visto como a saída para executar programas funcionais tão eficientemente quanto programas imperativos. No segundo, o qual perdura até os dias atuais, *a programação funcional passou a ser enxergada como uma alternativa real para o processamento paralelo.* Essa mudança de perspectiva é consequência da discutida evolução, ocorrida nos últimos anos, na tec-

nologia de compilação de programas funcionais em arquiteturas seqüenciais.

Em linguagens funcionais puras, devido a ausência de efeitos colaterais, a ordem de avaliação das expressões é irrelevante para o resultado de uma computação, de forma que é possível avaliá-las em paralelo. Burge [44], em 1975, utilizou este fato para sugerir uma técnica onde os argumentos das funções eram avaliados em paralelo (*paralelismo horizontal*), com a possibilidade de serem absorvidos não-avaliados pela função e com a exploração do paralelismo especulativo. Berkling [34] estudou mecanismos semelhantes a estes. Outro tipo de paralelismo explorado era o *paralelismo vertical*, onde um *pipe-line* era usado para passagens de valores entre funções.

Embora linguagens funcionais exponham melhor o paralelismo inerente dos programas, de forma a que a tarefa de paralelização seja simplificada, o uso destas em abordagens implícitas resultava em um paralelismo de muito fina granularidade, o que implicava na criação de uma grande quantidade de tarefas e, consequentemente, no aumento da sobrecarga de comunicação. Como o custo da criação e comunicação de processos é alto em arquiteturas paralelas convencionais, sobretudo em arquiteturas de memória distribuída, paralelismo de fina granularidade tende a gerar programas paralelos ineficientes. Chegou-se então a um consenso de que era necessário a criação de novas arquiteturas de *hardware* específicas, orientadas à execução de programas funcionais paralelos. Exemplos de arquiteturas desenvolvidas sob este contexto são ALICE (*Applicative Language Idealised Computing Engine*) [110, 70], ICL Flagship [227, 228], EDS/Goldrush [226] e GRIP (*Graph Reduction in Parallel*) [189, 2]. Entretanto, os experimentos mostraram que a construção de *hardware* de propósito especial é muito custoso, além de ter-se apresentado muito difícil que um dia tais arquiteturas apresentassem as mesmas vantagens práticas de máquinas de propósito geral disponíveis no mercado. Além disso, a rápida evolução da tecnologia dos microprocessadores sequenciais tornava improvável que máquinas de propósito especial pudessem superá-las no modo seqüencial de execução, onde é gasto a maior parte do tempo na execução de programas paralelos de granularidade média e grossa. Os resultados obtidos motivaram então o consenso de que o problema não se encontrava na arquitetura dos processadores, mas na tecnologia de comunicação entre estes.

Atualmente, implementações seqüenciais de linguagens funcionais apresentam desempenho próximo, em certos casos, a implementações de linguagens imperativas importantes, como C e FORTRAN [111]. Nesse novo cenário, o processamento paralelo encontrou um novo parceiro na programação funcional, de forma que algumas implementações de linguagens paralelas funcionais já se encontram a um bom tempo disponíveis publicamente [135, 215]. Estas implementações possuem algumas características em comum:

- Com o objetivo de prover uma alta carga de trabalho, utilizam técnicas especulativas dinâmicas para criação de tarefas;

- Implementações que utilizam paralelismo explícito empregam um ambiente paralelo de baixo nível (como PVM [97] ou MPI [73]), responsável pelo gerenciamento do balanceamento de carga de acordo com a carga de trabalho do sistema. O programador deve apenas fornecer indicativos de tarefas paralelas em potencial (anotações);

- A redução paralela do grafo procede em um grafo compartilhado de programas e dados. Dessa forma, uma função primordial do sistema em tempo de execução dessas linguagens é gerenciar a memória virtual compartilhada onde o grafo reside.

Em linguagens paralelas explícitas, como Occam [124], os usuários são responsáveis por configurar as tarefas que devem executar em paralelo. Particionar um programa seqüencial em tarefas paralelas não é um trabalho simples e requer um conhecimento completo do programa bem como da arquitetura sobre a qual este vai executar. Anotações, utilizadas pelo programador para identificar oportunidades de paralelismo em programas, são comuns. Hope+ e Flagship [137] empregam anotações para controle preciso do grau de avaliação. O exemplo mais completo de linguagem baseada em anotações é talvez Concurrent Clean [179], a qual permite a definição de anotações para controle da cópia e compartilhamento de grafos, bem como para alocação e escalonamento de tarefas. O conceito de *programação para-funcional* foi introduzido inicialmente por Hudak [119], definindo um conjunto de anotações para programas funcionais, os quais mantém sua semântica funcional. Estas anotações indicam escalonamento de eventos, o qual inclui demanda explícita, criação, terminação, composição paralela e seqüencial de processos e o mapeamento de tarefas para processadores particulares [120]. O sistema em tempo de execução de um programa para-funcional executa testes dinâmicos para verificar a carga do sistema e então decidir se uma tarefa paralela deve ou não ser criada. Caliban [136, 208] provê uma linguagem funcional separada para a tarefa de alocação de processos, no qual um programa é composto por uma parte de processo e outra de ligação. Eden [41] parte do mesmo princípio de Caliban quanto a separação entre paralelismo e computação, provendo ainda construtores para facilitar a definição de sistemas *reativos*. Caliban e Eden são consideradas linguagens paralelas por coordenação, assim como SCL (*Structured Coordination Language*)[69] e Delirium[157]. *Algorithm Skeletons* foram introduzidos por Cole [63, 64] com o objetivo de capturar padrões de computação paralela, como dividir-para-conquistar e *pipeline*. A idéıa é que um mesmo esqueleto possa ser usado em várias arquiteturas, sendo suficiente que sua implementação seja modificada para obter bom desempenho. Essa abordagem tem sido muito utilizada em programação paralela funcional, utilizando o conceito de funções de alta ordem. SCL é um exemplo de linguagem paralela funcional baseada em esquetos[68]. Caliban, Eden também provêem um rico conjunto de *skeletons* [136, 95]. Eden, Caliban, SCL e Delirium serão discutidas com maiores detalhes adiante.

Muitas implementações e extensões paralelas de Haskell têm surgido nos últimos anos. *Concurrent Haskell* [190] provê um substrato mais expressivo para construir programas que executam I/O, notadamente aqueles que suportam interfaces gráficas de usuários, para os quais a utilidade do suporte a concorrência é bem-estabelecida. A meta dos desenvolvedores de *Concurrent Haskell* é obter um paralelismo implícito e semanticamente transparente. Entretanto a versão corrente utiliza paralelismo explícito. GUM [214] é o sistema em tempo de execução construído para suporte a linguagem GpH (*Glasgow Parallel Haskell*) [215], uma extensão do compilador GHC (*Glasgow Haskell Compiler*) [102] para suporte ao paralelismo. GUM foi implementada sobre a biblitoteca de passagem de mensagens PVM, para instanciação e comunicação de tarefas paralelas, e, por isso,

apresenta boa portabilidade, disponibilizando versões para computadores de memória compartilhada (multiprocessadores SPARCserver) e arquiteturas de memória distribuída (*clusters*). Os experimentos mostram um aumento do desempenho dos programas em relação a versões seqüenciais, nessa implementação, embora com resultados bastante inferiores quando comparados aqueles obtidos para a Eden [156], a qual adota um mecanismo explícito com o mesmo propósito. O paralelismo adotado por GpH é semi-explícito, de forma que os programadores apenas *expõem* o paralelismo, isto é, identificam partes de programas que podem ser avaliadas em paralelo, utilizando-se para isso de anotações as quais denominou-se *estratégias de avaliação* [216]. Estratégias de avaliação, por serem funções de alta ordem podem ser definidas de forma ortogonal ao código funcional que expressa a computação realizada pela função, possibilitando nível mais alto de modularidade e abstração.

Recomenda-se [217] como referencial completo e atualizado sobre versões paralelas e distribuídas da linguagem Haskell. Uma abordagem mais geral, onde são discutidos aspectos pragmáticos a respeito do uso de processamento paralelo em linguagens funcionais, pode ser encontrada em [109].

2.5 PARALELISMO VERSUS CONCORRÊNCIA

Faz-se útil distingüir-se neste trabalho os conceitos de paralelismo e concorrência. O conceito de paralelismo pode ser enxergado como um caso particular do conceito de *concorrência*. Ambos pressupõem o particionamento de programas em unidades que executam simultaneamente (múltiplas linhas de instrução independentes), entretanto enquanto o primeiro utiliza tal artifício como meio de fazer com que tais unidades sejam executadas em processadores distintos visando ganho de desempenho, o outro o faz visando a estruturação de programas inerentemente compostos de módulos que possuem fluxos de controle independentes, sem restrições sequenciais de dependência em tempo de execução, a menos do emprego de protocolos de sincronização com a finalidade de evitar-se interferência, cujos principais são semáforos e monitores [6]. Nem todo sistema concorrente pressupõe a utilização de unidades de processamento distintas para execução de cada processo. Técnicas são empregadas para permitir a execução simultânea de várias tarefas em um mesmo processador, como compartilhamento de tempo (*time sharing*) e multi-tarefa (*multi-threading*) preemptiva.

O diagrama na Figura 2.5 mostra a relação entre as classes de sistemas concorrentes aqui discutidas. Associada a cada uma destas, foram desenvolvidos ao longo dos anos modelos de programação adequados, descritos a seguir:

- *modelos de programação distribuída*, para estruturação de sistemas concorrentes distribuídos, fora do contexto de processamento de alto desempenho;
- *modelos de programação concorrente com memória compartilhada*, voltados a estruturação de programas concorrentes onde as unidades executáveis (processos), compartilham um espaço de endereçamento, através do qual podem comunicar-se; e

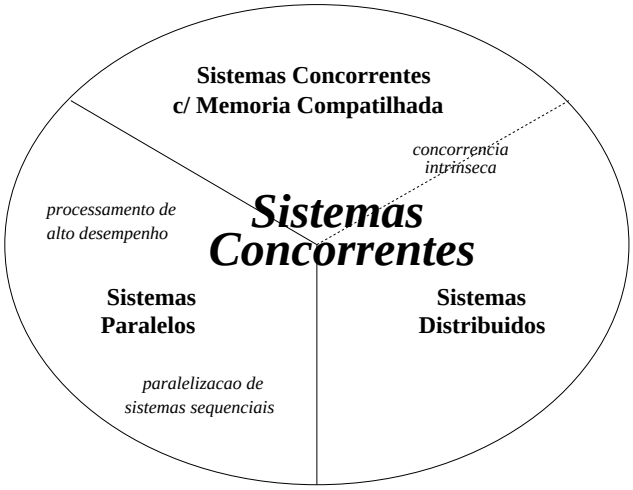


Figura 2.3. Taxonomia para Sistemas Concorrentes

- *modelos de programação paralela (explícita)*, voltados a busca de eficiência com a execução simultânea de programas em vários processadores, dentro de um contexto de *computação de alto desempenho*;

Tais modelos, estudados sob diferentes visões dentro da ciência da computação, são vistos como generalizações ao modelo convencional de programação sequencial de forma a suportarem as necessidades específicas das aplicações para as quais foram orientados. Além disso, em cada modelo, distinguem-se diferentes abordagens, geralmente voltadas a implementação sobre arquiteturas computacionais pré-determinadas. Embora potencialmente aplicáveis em contextos bastantes distintos, possuem em comum a exigência de algum tipo de interação entre componentes sequenciais.

Uma classificação conveniente para sistemas concorrentes deve-se a E. Shapiro [199]. Segundo ele, tais sistemas podem ser classificados em dois grupos: *transformacionais* e *reativos*. Sistemas transformacionais são aqueles que tem como objetivo a computação de um valor em função de outros valores recebidos como entrada, enquanto sistemas reativos tem como objetivo interagir com o ambiente onde estão executando, não possuindo em geral uma condição de terminação. Sistemas operacionais, sistemas de controle aplicados à automação industrial e sistemas de infusão hospitalar são exemplos clássicos de sistemas reativos. Sob esta ótica, podemos enxergar programas paralelos como casos particulares de sistemas transformacionais.

O mecanismo de particionamento natural para sistemas concorrentes é o *funcional*. O mecanismo de *particionamento de domínio* é intrínseco a sistemas de processamento paralelo, assim como certas preocupações, como *controle de granularidade* e *balanceamento de carga*, devido ao seu requisito primordial de desempenho. Ambientes para suporte a *programação concorrente* geral devem possuir mecanismos *explícitos*, expondo a noção de processo e o mecanismo sob o qual estes interagem. Devem ainda suportar a noção de *não-determinismo*, do ponto de vista comportamental.

2.6 O PARADIGMA DE COORDENAÇÃO

O *paradigma de coordenação* [99], proposto no início da década de noventa, consolidou-se como um arcabouço para orientar o projeto de novas linguagens e modelos de programação concorrente que buscam separar em níveis distintos as preocupações inerentes à especificação de entidades que realizam computações das entidades que realizam a coordenação entre estas. Provê-se dessa forma um grau de modularidade não suportado por linguagens que não aderem a esse paradigma.

Entende-se que o paradigma de coordenação não deve ser enxergado como mera extensão ao modelo tradicional de programação sequencial, mas como o modelo de programação por excelência. Sob esta ótica, modelos de programação sequencial não devem mais ser enxergados como auto-suficientes, mas sempre associados a um modelo de coordenação. Na prática, isso vem sendo observado na própria evolução das linguagens dentro do modelo tradicional de programação, as quais têm evoluído no sentido de admitir abordagens *degeneradas* de coordenação visando o suporte à *modularidade*, inicialmente com o surgimento do conceito de sub-programa (década de sessenta), posteriormente com o surgimento do conceito de *abstração* de dados (década de setenta), e mais recentemente com a emergência do *paradigma de programação orientado a objetos*.

Aproximando-se de uma abordagem mais formal, um modelo de coordenação pode ser representado através de uma tripla **(E,L,M)** [61, 230], onde:

- **E** representa um conjunto de entidades que devem ser coordenadas;
- **L** representa o meio utilizado para coordenar as entidades;
- **M** representa um arcabouço semântico ao qual o modelo adere.

Entidades são programas geralmente especificados em linguagens sequenciais convencionais e a coordenação destas define o comportamento computacional da aplicação. Em todas as linguagens de coordenação estudadas neste trabalho, à sintaxe da linguagem sequencial são acrescidas primitivas que servem como uma cola que une as entidades ao meio de coordenação. Um requisito básico de um modelo de coordenação é que tais primitivas não devem possuir nenhum efeito sobre a semântica da computação efetuada por cada entidade. Idealmente, entendemos que um modelo de coordenação deva prover uma separação total entre o componente de coordenação e o componente de computação (entidades) de programas, de forma que os mecanismos de coordenação sejam totalmente transparentes em relação às entidades coordenadas, tanto a *nível semântico* quanto a *nível sintático*. Dessa forma, do ponto de vista do nível de coordenação, *entidades* são enxergadas como *caixas pretas*, que podem ser acessadas por meio de interfaces bem definidas. Essa característica tem um impacto importante sobre a prática da engenharia de programas, por tornar ainda maior os graus de *modularidade* e *abstração* suportados pelo modelo de programação. Dentre as consequências bem vindas com esta prática, podemos citar como mais importantes:

i) O aumento do potencial para **reuso de componentes pré-existentes**, tanto a nível de computação quanto à nível de coordenação, este último dependendo do tipo de mídia utilizado;

ii) A **implementação e validação independente de cada componente**, seja ele de coordenação ou de computação, uma vez definidas suas interfaces e propriedades formais;

iii) A maior **facilidade para paralelização de programas sequenciais**, em virtude do potencial elevado de reutilização do código sequencial sem necessidade de modificações em sua estrutura lógica;

iv) A possibilidade de **tratamento formal hierarquizado**, utilizando formalismos apropriados em *nível de coordenação*, como redes de Petri [188], CSP [115], CCS [169] e π-calculus [170], e em *nível de computação*, principalmente quando uma linguagem funcional pura não-estrita é utilizada a esse nível;

v) O **suporte natural a uma abordagem multi-lingual**, onde várias linguagens sequenciais convencionais podem ser usadas para especificação da computação das entidades;

vi) A **redução da complexidade dos programas**, fato que deve ser observado sobretudo na construção de programas de larga escala, devido ao suporte a uma abordagem composicional [87] e, possivelmente, hierarquizada na construção de programas. O conceito de *programação em larga escala* é naturalmente suportado dentro deste contexto.

Os pontos positivos citados acima complementam-se de forma a oferecer um ambiente de programação que permite a construção de programas *confiáveis* e que facilita a especificação e implementação de programas *complexos* e de *larga escala*, duas das mais importantes preocupações no contexto da engenharia de programas [192].

Confiabilidade e Eficiência. O reuso de componentes (i) e de código sequencial pré-existente sem necessidade de modificações lógicas em sua estrutura (iii) permitem a reutilização de código previamente testado e validado. No caso da implementação de aplicações numéricas de larga escala com requisitos de alto desempenho e confiabilidade, como aquelas encontradas em *simulação de bacias petrolíferas* e *previsão climática*, por exemplo, existe um conjunto rico de bibliotecas numéricas disponíveis que podem ser reutilizadas. Estas, pelo fato de terem sido testadas, validadas e implementadas durante anos ou décadas, utilizando o estado-da-arte das técnicas numéricas existentes, são em geral eficientes e confiáveis. É portanto improvável a viabilidade da implementação de suas funcionalidades para aplicações específicas, considerando a necessidade de um bom balanceamento entre os custos associados ao desenvolvimento (tempo e recursos humanos) e benefícios em termos de desempenho e confiabilidade. A abordagem multi-lingual (v) torna ainda maior o potencial para reutilização de componentes e código, uma vez que estes podem estar especificados em diferentes linguagens. No exemplo citado anteriormente, não haveria então necessidade de utilizar-se implementações de bibliotecas específicas para uma certa linguagem, muitas vezes não disponíveis ou nem sempre as mais eficientes devido a limitações inerentes à própria linguagem. O tratamento formal

de programas (iii) permite a sua validação e análise de suas propriedades e custos de execução, de forma a que seus requisitos mínimos de desempenho e confiabilidade sejam garantidos. Além disso, ao permitir a construção independente de cada componente do programa (ii), torna-se possível que um ou vários programadores concentrem-se em certas partes críticas de uma aplicação, tornando mais fácil a sua implementação eficiente, confiável e rápida. Tal característica é também motivada pela redução da complexidade dos programas devido a natureza composicional de sua construção (vi). É natural que quanto maior a facilidade provida à tarefa de programação, maior a chance de que programas confiáveis sejam construídos.

Especificação Simples de Programas Larga Escala. O altos níveis de *modularidade* e *abstração* providos pelo modelo ideal de coordenação permitem a construção de programas utilizando uma abordagem composicional (6), onde um programa complexo ou de grande porte pode ser dividido em uma série de sub-problemas, os quais podem ser estudados, especificados, implementados e validados separadamente. Cada sub-problema poderia ser assim enxergado como uma entidade a ser coordenada e o meio de coordenação como a cola usada para uni-las com o fim de definir a solução do problema original. Em alguns casos, notadamente aplicações de grande porte e alta complexidade, a grande quantidade de sub-problemass identificados pode atrapalhar a compreensão da aplicação como um todo, devido a própria dificuldade inerente a inteligência humana em visualizar e entender problemas a partir de um certo grau de complexidade. Para estes casos, uma estrutura composicional hierarquizada, onde vários sub-problemas podem ser vistos de forma unificada sob a perspectiva de um sub-problema de maior nível, pode reduzir a complexidade aparente de uma aplicação. A linguagem de coordenação K2 [13] utiliza este conceito, permitindo que uma aplicação seja vista como uma entidade em outra aplicação. O suporte a reutilização de componentes sem alterações em diversos níveis (1, 3 e 5) reduz a necessidade de implementação de alguns sub-problemas ou entidades, diminuindo assim os custos associados a esta tarefa e naturalmente facilitando a implementação da aplicação como um todo. Além disso, é importante citar a facilidade oferecida para construção de programas paralelos a partir de versões sequenciais sem necessidade de alterações na lógica ou estrutura do código sequencial (3), o que torna naturalmente mais simples a tarefa de construção de programas paralelos.

2.6.1 Classificando os Modelos de Coordenação

Usualmente, os modelos de coordenação são classificados de acordo com o *tipo de mídia utilizada para coordenação das entidades computacionais*. Dessa forma, são reconhecidas dois tipos básicos de modelos [182]:

- **Orientados a Dados:** Modelos de coordenação *orientados a dados* tem como característica definir seu meio de coordenação em função dos conjuntos de dados trocados entre as entidades coordenadas. Em geral, são oferecidas às entidades primitivas que permitem o acesso a um meio comum onde dados produzidos ou necessários para as entidades estão localizados. Neste tipo de modelo, não há uma delimitação sintática entre os componentes de coordenação e computação. A

principal linguagem pertencente a essa categoria, considerada a primeira linguagem de coordenação ou pelo menos a que iniciou a discussão a cerca desse tipo de modelo, é Linda [98]. Na próxima seção descreveremos sucintamente esta linguagem, a partir da qual outras foram derivadas, como Bauhaus Linda [46], Bonita [196], Law-Governed Linda [173], Objective Linda [140], Ariadne [82], Sonia [23], Laura [213], and Jada [62];

- **Orientados a Processos:** Distintamente aos modelos orientados a dados, os *modelos orientados a processos* abstraem o meio de coordenação dos dados trocados e mantidos pelas entidades, separando assim de forma mais clara as entidades de *coordenação* das entidades de *computação*. Em geral, uma linguagem de configuração é utilizada para definir e coordenar entidades, de forma *estática* ou *dinâmica*. Tais linguagens utilizam um modelo inspirado em CSP (*Communicating Sequential Processes*), tendo sido influenciadas em alguns aspectos pela linguagem OCCAM, principalmente quanto aos mecanismos de comunicação, geralmente utilizando os conceitos de *portas* e *canais*. Exemplos de linguagens de coordenação dentro dessa classificação são Conic [163], Darwin/Regis [161], CL [131, 132], Caliban [136], Eden [41], Manifold [9], $Haskell_{\#}$ [50], K2 [13] e SCL [69], algumas das quais seão estudadas com mais profundidade adiante;

2.6.2 Exemplos de Linguagens de Coordenação

Nesta seção, descrevemos algumas linguagens de coordenação importantes que têm surgido deste o advento deste paradigma.

2.6.2.1 Linda [98] é reconhecida como a primeira linguagem de coordenação de fato, tendo motivado as discussões iniciais a cerca deste novo paradigma. Como veremos, seu elegante modelo de coordenação inpirou o surgimento de uma série de outras linguagens.

O meio de coordenação suportado por Linda baseia-se no conceito de *espaço de tuplas*, uma estrutura de dados compartilhada entre as entidades coordenadas (processos) composta por um conjunto de tuplas (registros). Estas podem ser *passivas*, representando dados, ou *ativas*, representando processos, quando um ou mais de seus componentes correspondem a chamadas de função. Tuplas são criadas e manipuladas pelas entidades que compõem um programa através de um conjunto restrito de primitivas, oferecidas como extensões a linguagem *host*, esta utilizada para especificação da computação realizada por cada entidade. As primitivas de Linda oferecem suporte ao acesso às tuplas passivas, utilizando o mecanismo de *casamento de padrões*. As principais primitivas são:

- **out(t):** adiciona a tupla passiva t ao espaço de tuplas;
- **in(a):** recupera, no espaço de tuplas, uma tupla passiva correspondente a *anti-tupla* a. A tupla recuperada é apagada do espaço de tupla;
- **rd(a):** possui o mesmo efeito de $in(a)$, com exceção de que a tupla recuperada não é apagada do espaço de tuplas;

- **eval(t):** adiciona uma tupla passiva ao espaço de tuplas, resultado da avaliação da tupla ativa recuperada do espaço de tuplas a partir da anti-tupla t. A avaliação ocorre com a criação de um novo processo, o qual executa em paralelo com o processo que o criou;

As primitivas **in** e **rd** são blocantes (síncronas), de forma que o processo pára até que uma tupla torne-se disponível no espaço de tuplas. As demais primitivas, ao contrários, são não-blocantes (assíncronas).

Anti-tupla Uma *anti-tupla* é definida como uma tupla que pode conter *variáveis* tipadas em um ou mais campos. São utilizadas para implementar o mecanismo de casamento de padrões. Para definir uma variável em um componente de uma anti-tupla, basta acrescentar o prefixo "?" antes do nome de uma variável. Por exemplo, utilizando o casamento de padrões, a anti-tupla ("*inteiro*", 3, ?a) casa com qualquer tupla que tenha como primeiro componente a cadeia "*inteiro*", como segundo componente o inteiro 3 e como terceiro componente qualquer valor do tipo da variável com nome a. Assim, se a é do tipo *ponto flutuante*, as tuplas ("*inteiro*", 3, 5.44) e ("*inteiro*", 3, 1.0) casam com a anti-tupla em questão, ao contrário de ("*inteiro*", 3, "$float$"), uma vez que "$float$" é do tipo *cadeia de caracteres*.

Como o conceito de tuplas está presente em virtualmente qualquer linguagem de programação convencional, é possível implementar as primitivas de Linda em qualquer linguagem. Dessa forma, virtualmente qualquer linguagem convencional pode ser usada como linguagem *host* em programas Linda. Atualmente, um rico conjunto de linguagens são suportadas por implementações de Linda incluindo-se C, Fortran, Java, Pascal, Ada, Lisp, Eiffel e Prolog.

Linda oferece um mecanismo elegante para separação dos mecanismos de coordenação e computação, permitindo que processos sejam desacoplados em tempo e espaço, ou seja não há dependências explícitas de dados entre processos. Entretanto, sua implementação em *arquiteturas de memória distribuída* é dificultada tendo em vista ser o espaço de tuplas compartilhado entre os processos. Isso contraria a tendência atual de consolidação desta classe de arquiteturas para processamento de alto desempenho. Algumas implementações de Linda sobre arquiteturas de memória distribuída tem surgido, entretanto com resultados ainda pouco expressivos de eficiência, como Piranha [100], um modelo de execução de programas Linda sobre *clusters*.

Linda inspirou o desenvolvimento de uma série de outras linguagens de coordenação, as quais buscavam suprir algumas de suas deficiências e limitações, principalmente em relação a eficiência e suporte a melhor estruturação de programas. Algumas dessas linguagens apenas extendem o modelo de coordenação de Linda, enquanto outras diferenciam-se desta linguagem em muitos aspectos. A seguir discutiremos as características dos principais representantes da classe de linguagens de coordenação derivadas de Linda.

Bauhaus Linda [46] extende o modelo original de coordenação de Linda para oferecer suporte a *espaços de tuplas múltiplos*, utilizando o conceito de *msets* (*multisets*) para sua implementação. Essa abordagem torna possível a definição de hierarquias entre espaços

de tuplas em programas, um método de estruturação bastante útil em muitos casos, o qual favorece a implementação eficiente de Bauhaus Linda sobre arquiteturas de memória distribuída e permite maior segurança de acesso aos dados em certos tipos de aplicações. O casamento de padrões associativo utilizado em Linda é substituído em Bauhaus Linda pelo conceito de *inclusão em conjunto não-ordenado*[2], de forma a permitir a que *msets* sejam adicionados, lidos, ou removidos de outros *msets*. O modelo de coordenação de Bauhaus Linda não faz distinção entre tuplas e espaços de tuplas, tuplas e anti-tuplas ou tuplas ativas e tuplas passivas.

Bonita [196] acrescenta ao conjunto de primitivas de Linda uma série de novas primitivas, com o objetivo de aumentar as funcionalidades oferecidas pelo modelo, através do provimento de espaços de tuplas múltiplos e manipulação de tuplas agregadas, e melhorar seu desempenho, oferecendo uma noção de recuperação de tupla de granularidade mais grossa com o fim de reduzir sobrecargas de comunicação. Resultados mostram que essa abordagem produz ganhos substanciais de desempenho, justificando seu uso.

Law-Governed Linda [173] introduz os conceitos de *controladores* e *leis*[3]. Estas consistem de regras que devem ser respeitadas pelos processos na realização de operações de comunicação, enquanto aqueles garantem o cumprimento das leis. Cada entidade tem seu controlador e seu conjunto de regras. Cabe ao controlador interceptar qualquer tentativa de comunicação do processo com o meio e permitir que ela se complete somente no caso dela obedecer a lei estabelecida. Um subconjunto restrito de PROLOG foi escolhido para formulação de leis. Observe que Law-Governed Linda não modifica o modelo de coordenação derivado de Linda nem sua implementação, ao contrário de Bauhaus Linda e Bonita, estabelecendo apenas um acesso controlado dos processos ao espaço de tuplas (dados), aumentando a confiabilidade de programas.

Objective Linda [140] foi desenvolvida tendo em vista sistemas abertos. Esta linguagem introduz um modelo de objetos adequado para este tipo de sistemas e independente da linguagem *host*. Tais objetos são descritos utilizando a linguagem OIL (Object Interchange Language). Operações sobre um espaço de objetos são suportadas para intercâmbio de objetos de forma adequada do ponto de vista de sistemas abertos. Ressaltamos também como importantes o suporte a hierarquias de espaços múltiplos de objetos, utilizando o conceito de *msets*, e a habilidade de objetos se comunicarem através de vários espaços de objetos. Assim como Objective Linda, outra linguagem que também permite a adequação de um modelo de coordenação baseado em Linda para sistemas abertos é Laura [213], a qual utiliza os conceitos de *agentes* e *serviços*.

Jada [62] é uma combinação de Java e Linda apropriada para sistemas abertos. Capaz de expressar coordenação de objetos e multi-tarefa, tem sido usado para construção de sistemas para WWW (*World Wide Web*) e projeto de linguagens de coordenação mais

[2] *Unordered set inclusion.*
[3] *Laws.*

complexas para WWW.

2.6.2.2 Manifold é uma implementação do modelo abstrato de concorrência IWIN (*Idealized Worker Idealized Manager*) [9]. As motivações que levaram ao surgimento deste modelo são as mesmas que motivaram a emergência do paradigma de coordenação, as quais foram discutidas em parágrafos anteriores. Nos parágrafos seguintes, apresentaremos seus principais conceitos e como Manifold os implementa efetivamente.

Os conceitos básicos do modelo IWIN são *processos*, *eventos*, *portas* e *canais*. Processos são as entidades executáveis, as quais podem implementar computações (processos trabalhadores[4]) ou coordenação entre processos (processos coordenadores[5]). Processos definem portas de comunicação, as quais possuem escopo local e podem ser de entrada e saída. Portas são conectadas através de canais de comunicação, os quais são discutidos posteriormente. Um mecanismo orientado a eventos também pode ser usado para comunicação entre processos. Dessa forma, um processo pode disparar uma *ocorrência de evento*, o qual é detectado pelos processos interessados em respondê-lo. O modelo de execução de processos é baseado em um sistema de *transição de estados*, onde transições são induzidas pela ocorrência de eventos. IWIN suporta um mecanismo de *comunicação anônima*, onde os processos se comunicam sem necessidade conhecimento da identidade dos outros processos. Esta característica é fundamental para diminuir o grau de acoplamento entre os processos, induzindo uma maior potencial para o reuso de componentes de *software*.

Em uma aplicação, podem existir vários níveis de hierarquia de coordenação entre processos, uma vez que um processo coordenador pode coordenar outros processos coordenadores. Esse mecanismo de abstração presta-se a estruturação de aplicações complexas. Os processos que pertencem ao nível mais baixo da hierarquia são chamados *trabalhadores atômicos*. Em geral, estes implementam computações em alguma linguagem base.

Canais de comunicação no modelo IWIN são do tipo *confiável*, de forma que não é assumida a possibilidade de perda, duplicação ou desordenamento das mensagens transmitidas através destes. Ao nível de coordenação, mensagens são tratadas como meras seqüência de *bits*, agrupadas em blocos de tamanho variável, chamados *unidades*, não havendo portanto a noção de tipo de canal. A interpretação do tipo de valor transmitido é realizado nas extremidades do canal (processo *fonte* e processo *destino*), a nível de computação. A conectividade dos canais pode variar durante a execução, de forma que é possível a exclusão de um canal ou a substituição de um dos processos em suas extremidades, fonte ou destino. O mecanismo de *reconfiguração* admite que *unidades pendentes* possam estar sendo transmitidas no canal durante o processo de desconexão, de forma que sua perda pode resultar em estados inconsistentes no programa. Para solucionar esse problema, o modelo IWIN pressupõe a existência de 5 modos de canais. No modo **S**, *comunicação síncrona*, é garantido que nunca há unidades pendentes durante o processo de reconfiguração. No modo **BB**, quando uma das extremidades do canal é desconectada a outra extremidade é desconectada automaticamente. No modo **BK**, o

[4] *workers*
[5] *coordinators*

canal é desconectado do processo produtor automaticamente tão logo é desconectado do processo consumidor. Entretanto a desconexão do produtor não implica que o canal será imediatamente desconectado do consumidor. No modo **KB** acontece o oposto, de forma que a desconexão do consumidor não implica a desconexão do produtor. No modo **KK**, a desconexão de um dos lados do canal não implica a desconexão do outro. A linguagem Manifold suporta todos os modos, com exceção do primeiro (**S**), assumindo que a comunicação é *assíncrona* sem perda de generalidade. Canais são chamados *streams*, em Manifold.

A linguagem Manifold foi projetada com o intuito de materializar os conceitos do modelo IWIN. Entretanto, como já foi dito, vale ressaltar que Manifold pressupõe que toda comunicação é assíncrona, enquanto IWIN admite ainda o modo de comunicação síncrona. Além disso, a distinção entre processos *trabalhadores* e *coordenadores* é reforçada. Essa caracterśitica facilita a adoção de uma abordagem multi-lingual, de forma que computações em processos pertencentes a uma mesma aplicação podem ser escritos em linguagens diferentes. Manifold é uma linguagem *fortemente tipada*, *estruturada em blocos* e *orientada a eventos*.

Em Manifold, todo processo implicitamente possui pelo menos uma porta de entrada e uma porta de saída. Processos coordenadores são chamados *manifolds*, cuja especificação consiste de um *cabeçalho* e um *corpo*. O cabeçalho fornece seu nome e seus parâmetros, enquanto o corpo especifica um bloco. Blocos especificam os estados que podem ser assumidos pelo processo. Um estado possui um rótulo, o qual identifica o evento que causa a transição para o estado em questão, e um bloco, o qual identifica as ações realizadas naquele estado ou estados aninhados. O modelo de execução é *preemptivo*, de forma que os estados podem ocorrer simultaneamente. Parâmetros de um manifold podem ser *eventos*, *processos* ou *portas*.

Uma forma especial de sub-programa, dentro do contexto de um *manifold*, é chamado de *manner*. Como qualquer sub-programa, *manners* tem um nome e parâmetros, os quais podem ser eventos, portas ou processos, e um corpo. O corpo de um *manner* pode ser um bloco, no caso deste ser do tipo *regular*, ou código C, o qual interfaceia com os processos Manifold por meio de bibliotecas de interface, no caso de *manners* do tipo *atômico*.

2.6.3 Programação Composicional

Assim como o conceito de coordenação, o conceito de *composicionabilidade* em programação paralela surgiu como uma forma de facilitar o desenvolvimento de programas sob este paradigma, aumentando o potencial de reuso de código sequencial, generalidade, heterogeneidade, e portabilidade. Dessa forma, podemos tratar o paradigma composicional como um modelo de programação por coordenação.

A principal característica de uma linguagem enquadrada no modelo composicional é a composição de programas a partir de componentes, os quais mantém suas propriedades. A composição é em geral é realizada por meio de operadores especiais, padrões que definem como os componentes estão relacionados entre si. Uma importante característica suportada por essas linguagens é a possibilidade de preservar o comportamento determinístico de seus componentes, simplificando o desenvolvimento de sistemas e permitindo que estes

possam ser construídos, testados e validados separadamente do sistema como um todo.

Existem basicamente duas categorias de linguagens concorrentes composicionais:

- Baseadas no conceito de *programação lógica concorrente*, como Strand [92, 207], PCN(*Program Composition Notation*) [91], Fortran-M [90] e C++ Composicional [57], os quais são discutidos em [87];
- Baseadas na noção de esqueletos [65], como SCL [6] [69], P3L [67] e Ektran [107].

Ao longo dos anos, o paradigma de programação lógica concorrente tem se mostrado um poderoso mecanismo para programação paralela, fato que pode ser verificado observando a grande quantidade de técnicas desenvolvidas para expressão de padrões efetivamente úteis de paralelismo. Linguagens como Strand buscam expressar os aspectos de coordenação de programas paralelos, deixando a cargo de uma linguagem mais apropriada, como C ou Fortran, a programação dos componentes computacionais. Usando essa abordagem, a separação entre os componentes sequenciais e concorrentes da linguagem não está intrínseca na semântica de construtores do paralelismo inseridos na linguagem computacional, mas explícita do ponto de vista sintático, além do ponto de vista semântico.

O uso de padrões pré-definidos para expressão de aspectos algorítmicos de sistemas foi primeiramente proposto por John Backus [15], entretanto foi Cole [63, 65] o primeiro a aplicar diretamente esse conceito à programação concorrente, com o advento do conceito de *esqueletos*. Estes são conjuntos de padrões pré-definidos a partir dos quais programas concorrentes podem ser compostos. O conceito de esqueletos é fruto do consenso de que existiam um conjunto pequeno de padrões a partir dos quais virtualmente qualquer programa concorrente poderia ser especificado. O sucesso da aplicação de esqueletos levou ao surgimento de linguagens desenvolvidas tendo por base este conceito, como SCL, P3L e Ektran. Outras linguagens incluíram esqueletos em seu projeto, como Caliban [136, 208] e Eden [41]. É importante observar a intrínseca relação entre o uso de esqueletos e a programação funcional, devido ao suporte nessas linguagens ao conceito de *funções de alta ordem*, um mecanismo de abstração de controle poderoso que permite a especificação de esqueletos sem acréscimos de extensões a linguagem base. Exemplos de linguagens paralelas funcionais de coordenação que tomam vantagem desse aspecto da programação composicional por esqueletos são SCL, Eden, Caliban e Delirium.

2.6.4 Linguagens Baseadas em Configuração

Linguagens de *configuração* são exemplos de MIL's (*Module Interconnection Languages*) [71], desenvolvidas para suprir os requisitos da programação de sistemas distribuídos tendo por base sua *especificação arquitetural*. Surgiram antes mesmo que o paradigma de coordenação emergisse como um modelo para programação concorrente. A característica que nos permite inseri-las dentro do contexto dos modelos de coordenação

[6] *Structured Coordination Language*

é a separação sintática da linguagem em dois subconjuntos: uma linguagem para especificação da computação realizada por unidades sequencias (componentes), em geral enriquecida com primitivas de comunicação por passagem de mensagens, e uma linguagem de configuração, responsável pela integração de tais unidades em uma aplicação distribuída mais complexa. Inspiradas na linguagem OCCAM [124] e linguagens de descrição arquitetural de *hardware*, tais linguagens tem fundamentação semântica no formalismo CSP [115]. A principal característica dessas linguagens é seu alto grau de modularidade e potencial para reuso de componentes, características bem vindas quando analisadas sob a ótica da engenharia de *software* e que favorecem a construção de sistemas complexos. Exemplos de linguagens que suportam configuração são PCL (*Proteus Configuration Language*) [205], Polylith [193], Durra [25], Conic [163], Darwin [160] e Olan [32]. Descreveremos adiante características relevantes das quatro últimas.

2.6.4.1 Conic [163, 142] pode ser enxergada como uma extensão para a linguagem PASCAL para programação de sistemas distribuídos segundo uma abordagem *construtivista* baseada no conceito de configuração. A linguagem de configuração adotada é responsável pela integração de componentes sequenciais (módulos), cujas interfaces são definidas por portas de entrada (*entry ports*) e saída (*exit ports*) fortemente tipadas. Há dois tipos de módulos: *tarefas* e *grupos*[7]. Os primeiros correspondem as unidades estritamente sequenciais, cuja computação é especificada com o subconjunto PASCAL da linguagem, enriquecida com primitivas para comunicação por passagem de mensagens. Instâncias concorrentes de tarefas, dentro de um mesmo espaço de endereçamento, podem ser agrupadas em módulos mais gerais, os grupos. Grupos também podem ser tratados como componentes em outros grupos, permitindo a definição de uma hierarquia de módulos em vários níveis. Um grupo, ao definir em sua especificação um *suporte executivo de execução*[8], é capaz de instanciar um *nó lógico*[9], unidade a partir do qual aplicações Conic são construídas. Em uma certa aplicação, se vislumbrarmos a hierarquia de módulos como uma árvore, tarefas correspondem às folhas e grupos aos nós intermediários e ao nó raiz. Este último representa a aplicação como um todo. Os nós filhos do nó raiz são grupos que necessariamente são capazes de instanciar nós lógicos. A facilidade para especificação de aninhamento de módulos dentro de uma hierarquia é importante para a estruturação composicional de *sistemas complexos*. Esta é uma das principais características que diferenciam Conic de Polylith e PCL, as quais não possuem essa facilidade.

É importante destacar que, em programas Conic, não há referências globais entre os módulos, nem um módulo pode acessar objetos declarados em outros módulos. Todas as referências são feitas a objetos locais (portas ou variáveis), mantendo-se dessa forma entre os módulos uma *indendência de contexto*, a qual induz um alto grau de modularidade e potencial para reuso de componentes.

Conic suporta *reconfiguração dinâmica*, permitindo a evolução e modificação do sis-

[7] *Task modules* e *group modules*.
[8] *Run-time executive support.*
[9] *Logical node type.*

tema em tempo de execução. Para certas classes de sistemas reativos, essa característica é particularmente importante. Entretanto, as possíveis alterações na estrutura da aplicação em tempo de execução são conhecidas em tempo de compilação, especificadas estaticamente em módulos separados. Vale ressaltar que essa abordagem limita o conjunto de tipos de módulo que podem ser usadas em uma aplicação a um conjunto fixo e pré-definido.

2.6.4.2 Darwin [160, 161] foi desenvolvida pela mesma equipe que desenvolveu a linguagem Conic, dentro do contexto do projeto do sistema Regis [162]. Por este motivo, é natural que seja considerada uma evolução desta linguagem. Como qualquer linguagem de configuração, Darwin baseia-se na noção de componentes, elementos básicos para composição de sistemas distribuídos. Entretanto, Darwin possui características peculiares que visam superar algumas limitações de Conic. A seguir, listamos algumas dessas características:

- Flexibilidade em relação a linguagem adotada para programação dos processos. Enquanto computações em Conic são expressas em uma linguagem baseada em PASCAL, em Darwin virtualmente qualquer linguagem poderia ser usada. Para isso, a programação dos componentes é realizada em módulos separados do programa de configuração, garantindo a ortogonalidade dos subconjuntos de configuração e computação da linguagem, necessário para que seja possível substituir a linguagem de programação dos componentes (computação) sem que haja necessidade de modificar a linguagem de configuração;

- Mecanismo mais geral de reconfiguração dinâmica de sistemas, com o suporte a instanciação *procastrinada*, onde os componentes são instanciados no momento em que seus serviços são requsitados em tempo de execução, e *dinâmica* (direta). Como vimos, Conic permite apenas a modificação de padrões estáticos de configuração pela invocação de um gerenciador;

- Suporte a definição de primitivas de comunicação definidas pelo usuário, enquanto Conic possui um conjunto pré-definido de tais primitivas;

- Distribuição física dos componentes em execução é completamente ortogonal a estrutura lógica do sistema.

Assim como em Conic, Darwin suporta a estruturação hierárquica de programas. Os conceitos de *processo* e *componente* em Darwin correspondem, respectivamente, aos conceitos de *tarefa* e *grupo* em Conic. Dessa forma, um componente pode ser estruturado a partir de processos e outros componentes, enquanto processos são especificados utilizando a linguagem de computação, a qual é sintaticamente independente da linguagem de configuração. Assim como em Conic, a comunicação também é realizada por portas de comunicação locais, as quais são conectadas para descrever a topologia de comunicação de processos.

2.6.4.3 CL [131, 132] CL foi desenvolvida no Centro de Infomática, Universidade Federal de Pernambuco, desde o final da década de 80. Esta linguagem suporta abstrações semelhantes aquelas já introduzidas para as demais linguagens de configuração. Vale ressaltar o suporte à reconfiguração dinâmica e à estruturação aninhada de *componentes*. CL introduz alguns conceitos originais, como a noção de *reconfiguração planejada*, uma modificação dinâmica na estrutura da aplicação considerada pelo projetista como possível de acontecer. CL suporta ainda o conceito de reconfiguração sincronizada, como uma de suas operações primitivas. Outros trabalhos inspirados em CL são C++CL [134], um *framework* C++ para o desenvolvimento extensível e reusável de aplicações distribuídas e ZCL [187], um arcabouço formal para especificação de aplicações distribuídas dinâmicas.

2.6.4.4 Durra [24, 25] foi desenvolvida no SEI (*Software Engineering Institute*), na universidade de Carnegie Mellow, EUA, a partir da segunda metade da década de 80. Sua principal motivação foi a construção de uma ferramenta de programação segundo uma abordagem que integrasse em um mesmo arcabouço a especificação, modelagem e prototipagem rápida de sistemas distribuídos, especialmentes aqueles com restrições temporais (*real-time systems*). Dessa forma, além da especificação da estrutura da aplicação, a linguagem de configuração de Durra permite a especificação de seu comportamento, restrições temporais e dependência de implementação (um componente pode possuir diversas implementações, as quais podem ser usadas em diferentes contextos).

Em Durra, tipos de componentes podem ser *tarefas*[10], a partir dos quais são instanciados as entidades executáveis (processos), ou *canais*, os quais descrevem padrões de comunicação entre processos. Observe que nas outras linguagens estudadas, os componentes descrevem somente entidades executáveis.

A abordagem multi-lingual é virtualmente suportada, permitindo a utilização de qualquer linguagem para programação de *tarefas*. Assim como Conic e Darwin, Durra oferece suporte a reconfiguração dinâmica, permitindo a modificação da estrutura de um programa em tempo de execução.

Durra foi projetada para suportar a construção de sistemas obedecendo três estágios. O primeiro corresponde a criação de uma biblioteca de componentes (*tarefas* e *canais*), os quais podem ser usados em diferentes contextos. Componentes são declarados como tipos e especificados por meio da linguagem de configuração de Durra. Para um certo componente, várias implementações podem ser construídas e disponibilizadas na biblioteca, obedecendo sua especificação mas possivelmente variando segundo características como a linguagem utilizada, o algoritmo empregado, o tipo de *hardware* suportado, etc. O objetivo dessa abordagem multi-versões é o suporte a *ambientes de execução heterogêneos* e *alto grau de portabilidade*. O segundo estágio corresponde a especificação da aplicação em si, a qual sintaticamente também corresponde a uma tarefa e deve ser incluída na biblioteca para reuso. É importante observar que, assim como Conic e Darwin, componentes podem ser especificados a partir da composição de outros componentes em qualquer grau de aninhamento, permitindo a estruturação de aplicações complexas. A terceira e última fase corresponde a execução da aplicação, onde os processos são instanciados e execu-

[10]do inglês, *tasks*.

```
task nome-da-task(lista-de-parâmetros)
    ports
      declaração de portas
    behaviour
      requires
            pre-condições
      ensures
            pos-condições
      timing
            restrições temporais e de ordenação de operações sobre portas
    attributes
      pares atributo-valor
    components
      declaração de componentes
    structures
      conexões de componentes
      transformações de dados
      diretivas de alocação de processos
    reconfigurations
      pares ação-condição
    clusters
      alocação de tarefas a processsadores
end nome-da-task
```

Figura 2.4. Estrutura da Configuração de uma Tarefa em Durra

tados nos processadores da arquitetura alvo, tarefa efetivada pelo sistema em tempo de execução de Durra.

A configuração de uma tarefa em Durra obedece a estrutura apresentada na figura 2.4. Nos parágrafos a seguir, descrevemos o significado de cada seção de uma configuração.

Portas, de entrada ou saída, são declaradas na seção **ports**. Assim como nas demais linguagens de configuração, Durra suporta o conceito de portas de comunicação tipadas, as quais são visíveis a nível de processo (*tarefa*) e configuração.

Na seção **behaviour** é especificado o comportamento que um processo instanciado a partir da tarefa deve assumir. Este é descrito em duas partes: *funcional* e *temporal*. Na parte funcional, são descritos predicados, na forma de *pré-condições* e *pós-condições*, aplicados sobre os dados que trafegam através das portas em cada ciclo de execução do processo. Pré-condições e pós-condições podem ser especificados utilizando, respectivamente, as cláusulas **requires** e **ensures**. Na parte temporal, o programador deve fornecer uma expressão [11] que descreve as restrições de ordem e tempo em que operações de comunicação são realizadas sobre as portas. As implementações de um tarefa devem satisfazer tais restrições.

Na seção **attributes** são descritos pares atributo-valor que definem propriedades gerais de um componente. Atributos tem um papel importante na construção de protótipos de *software*, permitindo a parametrização de um componente em termos da implementação utilizada dentro de um certo contexto.

Na seção **components** são instanciadas tarefas e canais (*componentes internos*) que compõem a estrutura da tarefa.

Na seção **structures**, é especificada a estrutura da tarefa, ou seja, como os componentes internos declarados na seção anterior são interligados. *Filas* são criadas para estabelecer as ligações lógicas entre portas de entrada e saída de processos distintos. Transformações podem ser especificadas nos dados que trafegam em uma fila, utilizando processos instanciados especificamente para este fim.

[11] *timing expression*

Na seção **reconfiguration** são especificadas modificações que podem ser realizadas na estrutura de uma *tarefa*, bem como as condições necessárias para que estas sejam ativadas.

Na seção **clusters** são declaradas as diretivas que devem orientar o compilador na alocação das tarefas aninhadas aos processadores físicos da arquitetura.

2.6.4.5 Olan A plataforma *middleware* Olan [32] foi desenvolvida no *Instituto Nacional de Pesquisa em Informática e Automação* (INRIA), França. Seu principal objetivo é a descrição de arquiteturas distribuídas complexas, sua configuração de acordo com os requisitos da aplicação e o desenvolvimento de aplicações sobre ambientes heterogêneos. A linguagem OCL (*Olan Configuration Language*), objeto de nosso interesse, é uma *linguagem de definição arquitetural* projetada especificamente para aplicações distribuídas. Inicialmente inspirada em Darwin, OCL herdou algumas de suas características, como a composição hierárquica de componentes e o suporte a instanciação *procastrinada* e *direta* de componentes. No entanto, possui algumas peculiaridades que a diferenciam, como o uso de *conectores* [201] como unidade básica para mediação da interação entre componentes. Conectores são uma abstração correspondente ao conceito de canais em Darwin, no entanto permitindo a especificação de parâmetros que ditam o modelo e protocolo de comunicação adotado. A interface dos componentes é descrita por meio de *serviços*, providos ou requisitados, enquanto em Darwin interfaces são descritas utilizando o conceito de porta, de entrada e saída. Um componente OCL pode prover serviços do tipo *clássico*, ativado por chamada de procedimento, ou *baseados em eventos*. Analogamente, os tipos de requisições possíveis são *síncronos* (chamadas de procedimento) ou *submissões de eventos*, para os quais uma resposta é esperada. A interface de um componente inclui ainda *atributos*, os quais descrevem seu estado corrente durante a execução, bem como suas propriedades.

É importante ressaltar a influência do paradigma orientado a objetos no projeto de Olan, sobretudo na descrição dos componentes individuais.

Olan suporta reconfiguração dinâmica por meio da instanciação dinâmica de componentes e do conceito de *coleções*. Nos moldes de Darwin, há em Olan três diferentes modos de instanciação de componentes, quanto ao tempo em que esta é efetivada.

i) **Implícita.** Ao mesmo tempo da instanciação do componente do qual ele faz parte;

ii) **Procastrinada.** Quando a primeira requisição para o componente é realizada;

iii) **Direta.** Quando uma ordem explícita de instanciação é enviada.

Coleções são conjuntos de instâncias de um mesmo componente, cuja cardinalidade pode variar durante a execução. Uma coleção é identificada por um nome e o tipo do componente associado. Um conector específico é usado para controlar a evolução da coleção. O acesso aos membros de uma coleção é realizado por um mecanismo de nomeação associativa, onde operações podem ser realizadas somente com alguns membros da coleção que satisfazem certas propriedades, expressas por seus atributos.

O mecanismo utilizado por Olan para a distribuição dos processos na arquitetura distribuída é mais sofisticado do que aquele empregado nas linguagens anteriormente estudadas. Por sofisticação entendemos o fato de que não apenas é levado em consideração o nó onde o componente executará, mas também fatores adicionais como os usuários para os quais o componente poderá executar. O mecanismo é bastane flexível, permitindo que projetistas de aplicações e administradores especifiquem políticas de alocação, evitando a nomeação direta de nós. A definição de tais políticas está intimamente ligada a noção de *contexto*. Um contexto é uma entidade abstrata unicamente identificada por um *usuário*, uma *aplicação* e uma *máquina*. A especificação da distribuição indica em que contexto um componente deve executar. A nível de OCL, a política de alocação é especificada em termos de predicados envolvendo *atributos de gerenciamento*[12], o qual correspondem a recursos físicos que devem ser usados em tempo de execução. O predicado é avaliado em tempo de instalação do componente, com o fim de encontrar o nó onde este executará. Um atributo de gerenciamento é composto de um conjunto de campos, os quais o caracterizam.

2.6.5 Outros Exemplos de Linguagens de Coordenação

Na literatura, é possível encontrar uma extensa lista de linguagens que seguem o paradigma de coordenação, além dos exemplos representativos ilustrados neste capítulo. Nos omitimos assim de uma série de linguagens, como GAMMA [21], LO [5], COOLL [53], MESSENGERS [94], Synchronisers [93], CoLa [114], Opus [58], RAPIDE [202], ConCoord [116], TOOLBUS [33], etc. Sugerimos [10] como *survey* das principais linguagens e modelos de coordenação.

2.6.6 Programação Paralela Funcional e Coordenação

Nesta seção, discutimos a aplicação do paradigma de coordenação no projeto de linguagens funcionais paralelas. Neste contexto, são reconhecidas duas motivações a integração do paradigma de coordenação com o paradigma funcional. No primeiro, linguagens funcionais são reconhecidas como uma forma elegante de definir meios de coordenação de processos, em especial devido a sua capacidade de abstrair fluxos de controle com o uso de avaliação *lazy* e funções de alta ordem. No segundo, o paradigma de coordenação emerge como um arcabouço apropriado para o projeto de linguagens paralelas funcionais, de forma a superar certas dificuldades encontradas em separar o código de especificação funcionais do código de configuração do paralelismo. Portanto, o paradigma functional pode ser aplicado em ambas as hierarquias dentro de um modelo de coordenação. No meio de coordenação, uma linguagem funcional pode ser usada para expressar a coordenação entre processos, em geral utilizando-se as facilidades de funções de alta ordem. Exemplos de linguagens que utilizam esta abordagem são SCL e Delirium. No meio de computação, linguagens funcionais pode ser usadas para expressar computações de forma ortogonal ao meio de coordenação, como no caso de Eden, Caliban e Haskell$_{\#}$.

Nas seções que se seguem descrevemos as principais linguagens paralelas funcionais integradas ao paradigma de coordenação. Nos capítulos que se seguirão, a linguagem

[12]No original, *management attributes*

Haskell$_{\#}$, produto desta tese de doutorado, será descrita em detalhes.

2.6.6.1 SCL - Structured Coordination Language [69] foi desenvolvida com o fim de suprir a necessidade de um modelo de coordenação suficientemente geral que suporte todos os requisitos para a construção de programas paralelos *estruturados*, *portáveis* e *verificáveis*, problema reconhecido como o maior desafio da programação paralela. SCL utiliza o paradigma de programação funcional em nível de coordenação com o fim de permitir a expressão de todos os aspectos essenciais do paralelismo, como *particionamento e distribuição de dados*, *comunicação entre processos* e *multi-tarefa* de uma maneira uniforme, implementando a noção de *esqueletos funcionais*. Atualmente, SCL tem sido implementada utilizando Fortran como linguagem para especificação de computação. SCL utiliza o poder expressivo das funcões de alta ordem das linguagens funcionais, para especificação de esqueletos. Observe-se que, nesta linguagem, o uso do paradigma funcional como pré-requisito necessário ao modelo, é realizado a nível de coordenação.

A habilidade de compor esqueletos pré-definidos provê o suporte necessário à construção de novos esqueletos, orientados a aplicações específicas ou a especificação de importantes aspectos da computação paralela. O suporte à *portabilidade*, no sentido de permitir a execução de programa em várias arquiteturas aproveitando ao máximo seu potencial de desempenho é oferecida pela implementação particular de cada esqueleto sobre arquiteturas específicas, de forma a tirar vantagem das características intrínsecas de cada arquitetura que afetam o desempenho de programas. Otimizações, as quais podem ser obtidas através de transformação de programas são também possíveis, principalmente devido ao emprego de linguagens funcionais de alta ordem na especificação dos esqueletos.

SCL define quatro classes de esqueletos que podem ser usados para construção de virtualmente quaisquer tipos de aplicações:

- **Esqueletos de configuração:** Modelam a divisão lógica e distribuição de objetos de dados visando o seu processamento em paralelo;
- **Esqueletos elementares:** Abstraem as operações básicas do modelo de computação paralela de dados;
- **Esqueletos de comunicação:** Modelam a comunicação entre processadores, expressa através de movimento de elementos em vetores distribuídos;
- **Esqueletos de computação:** Suportados no intuito de prover flexibilidade ao programador para organização do fluxo de controle de tarefas paralelas de um programa em um ambiente paralelo, de forma composicional.

2.6.6.2 Eden [41] extende a linguagem funcional *Haskell* com novos construtores sintáticos com o objetivo de prover tratamento explícito de concorrência em arquiteturas de memória distribuída. Seu modelo de coordenação utiliza a abordagem de comunicação baseada em *streams* para suporte a intercalação transparente de comunicação e computação na execução de processos. Em nível de coordenação, processos comunicam-se

através uma rede de comunicação, enquanto em nível de *computação*, o comportamento dos processos é especificado através de funções *Haskell*.

A construção de um programa Eden envolve basicamente a especificação de processos e suas interdependências de comunicação, estas representadas através de canais. Processos são definidos através de *abstrações* e criados através de *instanciações* a partir destas. A comunicação entre processos é realizada por meio de *passagem de mensagens* através de canais de comunicação. As operações de *envio* e *recebimento* são transparentes, não havendo necessidade de operadores explícitos para estes fins. Listas podem ser tramsmitidas eficientemente na forma de *streams*, sob demanda, isto é, um elemento de cada vez. Para modelagem de *sistemas reativos* [199][13], Eden incorpora os conceitos de *canais de resposta dinâmicos* e processos pré-definidos para suporte ao não-determinismo: **merge** e **split**, como forma de suportar maior expressividade para implementação de uma rica classe de sistemas distribuídos.

2.6.6.3 Caliban Caliban [136, 208] extende a linguagem *Haskell* para suporte ao processamento paralelo. Para isso, define um esquema funcional de anotações para permitir a especificação de uma rede estática de processos e alocação destes a processadores de uma arquitetura presumivelmente de memória distribuída. Processos Caliban são expressões que produzem listas de valores, interpretadas como *streams*. A comunicação entre os processos é realizada utilizando o conceito de canais de comunicação, os quais interligam *streams* produzidas por processos a *streams* consumidas por outros. Note que Caliban utiliza conceitos semelhantes a Eden, entretanto enquanto esta propõe-se a ser uma linguagem concorrente de uso geral, Caliban volta-se à construção de programas concorrentes tipicamente paralelos.

Uma anotação em Caliban é expressa através de uma estrutura de dados *Haskell*, especificada no contexto da cláusula especial **moreover**. Esta é definida no contexto da definição de uma função. As principais anotações de Caliban são as seguintes:

- **NoPlace:** Assertiva *nula*, sem significado;
- **Bundle** $[e_1, \cdots, e_n]$**:** Expressa que as expressões e_i, $1 \leq i \leq n$, devem executar no mesmo processador e qualquer referência não-local a estas envolve comunicação. Um esquema de preempção é adotado, de forma que para cada expressão é criada uma *thread* (processo leve);
- **Arc** e_1 e_2**:** Estabelece uma dependência de dados entre as expressões a e b, de forma que a consome a *stream* produzida por b ou vice-versa. A dependência de dados entre processadores responsáveis por duas expressões pode ser estaticamente verificada pelo compilador. Assim, quando é definida uma anotação **Arc** a b e o compilador verifica que não há dependência entre as expressões a e b ou quando o compilador verfica a existência de uma dependência quando nenhuma anotação **Arc** foi definida, uma *advertência* é informada pelo compilador.

[13]Sistemas que em geral não possuem uma condição de terminação, tais como sistemas de controle em automação industrial e sistemas operacionais.

- a_1 **And** a_2**:** Dadas duas anotações a_1 e a_2, permite a especificação de uma *composição paralela*, onde as redes de processos definidas por a_1 e a_2 executam em *conjuntos disjuntos de processadores*;

- a_1 **With** a_2**:** Dadas duas anotações a_1 e a_2, permite a especificação de uma *composição sequencial*, onde as redes de processos definidas por a_1 e a_2 executam no mesmo *conjunto de processadores*;

- **Annot** e**:** Recupera a anotação de alocação definida para uma expressão. Por exemplo, uma anotação do tipo `(Annot` e_1`) With (Annot` e_2`)` descreve uma rede de processos onde e_1 e e_2 são atribuídos ao mesmo conjunto de processos.

Utilizando essas anotações, podemos especificar redes de comunicação estáticas entre processos e controlar sua alocação a processadores em uma arquitetura de memória distribuída. Observe-se o exemplo seguinte, o qual expressa uma anotação:

((Bundle[a,b] And Bundle [c,d,e]) With (Annot f)) And (Arc a d) And Arc (a f)

Nesta anotação, a, b, c, d, e e f são expressões, separadas em três grupos (G_1=$\{a,b\}$, G_2 = $\{c,d,e\}$ e G_3 = $\{f\}$), de forma que processos que pertencem ao mesmo grupo executam concorrentemente em um mesmo processador. A anotação **And** garante que os grupos G_1 e G_2 sejam alocados a processadores distintos, enquanto a anotação **With** especifica que a rede de processos onde a expressão f será executada, inferida pelo uso de **Annot**, seja a rede formada pelos processadores alocados para os grupos G_1 e G_2. *Streams* devem conectar as expressões a e d e as expressões a e f, estabelecendo *canais de comunicação* entre os processos que as avaliam.

2.6.6.4 Delirium [157] pode ser considerada como o primeiro exemplo de linguagem paralela de coordenação a oferecer separação a nível *sintático* entre o *código de coordenação* e o *código de computação* das entidades, através do uso de uma linguagem separada para coordenação de processos. Ao contrário, em Linda, primitivas de coordenação são implementadas como extensão à linguagem *host*, tornando-se embutidas dentro desta linguagem. Delirium adota a abordagem inversa, tornando a linguagem *host* embutida na linguagem de coordenação, através do uso de *operadores*. Estes tem portanto o mesmo papel das primitivas de Linda, ou seja, unir as entidades coordenadas ao meio de coordenação.

Basicamente, Delirium é uma linguagem funcional convencional, suportando funções de alta ordem, recursão, *let-bindings*, e expressões condicionais. No entanto, não há nenhuma primitiva de computação definida na linguagem. Computação é especificada por meio de operadores definidos em outra linguagem, como C ou Fortran. Abstrai-se assim de forma completa a parte de coordenação da parte de especificação da computação realizada por cada entidade coordenada. Partindo do pressuposto de que, em geral, o processo de construção de um programa paralelo inicia-se com base em uma versão sequencial deste, a partir do qual são identificadas partes que podem ser executadas em paralelo, Delirium oferece um ambiente de programação bastante expressivo, permitindo que as

partes paralelas do programa sejam integradas utilizando o arcabouço de coordenação provido pela linguagem, evitando assim a reescrita de código.

O modelo de paralelismo de Delirium é determinístico, tornando tal modelo satisfatório para expressão de programas paralelos *síncronos*, utilizando a noção de *estruturas de coordenação* [158]. Entretanto, a grande crítica com relação ao modelo de coordenação adotado por Delirium é justamente sua dificuldade em expressar padrões irregulares de paralelismo, sendo assim bastante restritivo. Experimentos mostraram a ineficiência de implementações de programas paralelos que aderem a padrões comuns de paralelismo, como o modelo de *trabalhadores replicados* (*farm*) e *fork-join*. Entretanto, seus defensores arguem que são justamente esses padrões irrestritos de interação, os quais podem gerar programas não-determinísticos, que tornam os programas paralelos tão difíceis de manter e depurar.

Delirium tem sido implementada eficientemente sobre arquiteturas de memórias distribuídas, utilizando um esquema de avaliação de operadores baseado em fluxo de dados, através de um *grafo de coordenação* que modela as dependências de dados entre as unidades paralelas do programa. Uma importante observação relativa a esta implementação é a baixa sobrecarga gerada pelo seu sistema em tempo de execução. Isso pôde ser confirmado em uma aplicação de modelagem de retina em quatro processadores, onde a sobrecarga correspondeu a 1% do tempo total de execução.

CAPÍTULO 3

O MODELO # PARA PROGRAMAÇÃO PARALELA

A necessidade por ferramentas de mais alto nível que tornem a tarefa de programação paralela mais simples e abstrata para o desenvolvimento de aplicações complexas foi discutida nos capítulos 1 e 2. Especificamente neste último, foram discutidos aspectos relacionados à tarefa de programação paralela sobre arquiteturas distribuídas, em especial *clusters*. Ênfase foi dedicada às limitações das técnicas atuais, motivando a exposição do ponto de vista, adotado neste trabalho, a cerca das características que devem ser suportadas por um ambiente de programação paralela ideal, as quais visam tornar essa tarefa mais simples, aproximando seu nível de dificuldade à tarefa de programação seqüencial. O emprego de formalismos para a análise de propriedades formais de programas, modelagem de desempenho e simulação, algo negligenciado nas ferramentas existentes atualmente, deve ainda ser suportado. O modelo de coordenação foi introduzido como um importante passo conceitual para a evolução dos modelos de programação paralela de forma a atender esses requisitos.

O modelo #, apresentado neste capítulo, surgiu como consequência das idéias ilustradas nos capítulos anteriores. Sua concepção e evolução foi orientada pelo seguinte conjunto de premissas e suposições:

- Em ferramentas de programação paralela, o uso de **mecanismos implícitos** para exploração transparente do paralelismo em programas tem produzido resultados modestos com relação ao *desempenho*. A este fato atribui-se o difícil trato computacional característico dos algoritmos para solução ótima e generalizada dos problemas envolvidos no gerenciamento automático do paralelismo, como o *particionamento* e o *balanceamento de carga*, quando existentes. Especificamente no problema de particionamento, a determinação do grau ótimo de granularidade é ainda dificultado devido à dependência com a própria semântica da aplicação e de características intrínsecas à arquitetura alvo, as quais são difíceis de serem modeladas satisfatoriamente por meios formais;

- O uso de **mecanismos dinâmicos** para gerenciamento do paralelismo em tempo de execução também não têm produzido resultados animadores em casos gerais. Isso se deve à necessidade do uso de sistemas de gerenciamento do paralelismo em tempo de execução. Estes implementam algoritmos que executam concorrentemente às computações efetivas do programa, causando uma sobrecarga que soma-se à sobrecarga de comunicação já existente. Para reduzi-la, esses sistemas devem idealmente ser implementados de forma muito eficiente, possibilidade que contradiz-se ao difícil trato computacional dos problemas gerais associados ao gerenciamento automático do paralelismo. A solução para esse problema tem sido a implementação de al-

goritmos que se aplicam a instâncias simplificadas do problema geral ou o uso de *heurísticas*, as quais não possuem garantias de resultados ótimos;

- Com a grande evolução em termos de desempenho nas tecnologias de comunicação em redes de computadores e arquiteturas de processadores seqüenciais, com a redução dos custos associados a aquisição destas tecnlogias, o uso de **arquiteturas de memória distribuída** tornou-se viável em processamento de alto desempenho. Essas arquiteturas oferecem grande *escalabilidade*. A emergência da tecnologia de *clusters* [18], sobretudo compostos com computadores pessoais, tem tido um papel preponderante neste contexto, por estes oferecerem poder computacional comparável a supercomputadores, porém a um custo muito inferior;

- Programas paralelos de **granularidade fina** apresentam baixo desempenho quando executados sobre *arquiteturas de memória distribuída*, apesar da rápida evolução da tecnologia de comunicação nessas arquiteturas observado na última década;

- No paralelismo explícito, a **mistura do código sequencial (computações) com o código que lida com o gerenciamento do paralelismo** dificulta a construção e compreensão de programas paralelos, inviabilizando o reuso de código e tornando-o pouco portável. Nessa abordagem, multiplica-se à dificuldade inerente à programação sequencial a dificuldade associada ao gerenciamento do paralelismo. A análise formal de programas, uma possiblidade real devido a existência de formalismos consagrados para esse fim [115, 169, 170, 188], é impossibilitada de ser realizada automaticamente, uma vez que o paralelismo, embora explícito, encontra-se obscurecido no código. Este problema se agrava quando o código sequencial é escrito em uma linguagem imperativa, as quais já são de difícil trato formal em sua forma pura. No caso de linguagens paralelas que extendem linguagens pré-existentes, torna-se necessário construir novos compiladores e, em muitos casos, sistemas em tempos de execução para gerenciamento do paralelismo. Não há portanto um aproveitamento direto da tecnologia de compilação sequencial já existente, o que é importante na otimização de desempenho de programas de grossa e média granularidade, onde a maior parte do tempo de execução do programa é gasto no modo sequencial de execução;

- **Não existem metodologias para engenharia de programas paralelos em seu caso geral**. Atribui-se este fato à diversidade de modelos de programação e arquiteturas físicas destinadas ao suporte ao paralelismo, dificultando a adoção de uma abordagem padrão para engenharia de programas;

- Em aplicações paralelas de alto desempenho, sobretudo em computação científica e engenharia, a estrutura topológica e o padrão de interação entre os processos é geralmente regular e estática [186, 177, 28], sendo estas portanto suposições realistas no projeto de uma linguagem voltada à programação paralela em computação de alto desempenho.

Com base nas premissas e suposições estabelecidas, o modelo # é *estático*, *explícito*, e fundamentado na noção de *hierarquia de processos*. Processos são programados utilizando uma linguagem sequencial (linguagem *host*). Estes são então interligados em uma rede de comunicação descrita através de uma linguagem de configuração capaz de definir canais através dos quais os processos comunicam-se. As características das linguagens de configuração, empregadas largamente em sistemas distribuídos e descrição de *hardware*, favorecem o emprego de metodologias modulares de desenvolvimento de sistemas de grande escala [144].

No projeto do modelo #, a linguagem de configuração deve ser projetada de forma a manter capacidade expressiva, para a descrição de padrões de interação entre processos, equivalente às redes de Petri lugar/transição [188], um disseminado formalismo com grande número de ferramentas disponíveis para o seu suporte. Podemos então usar redes de Petri para permitir a análise de propriedades de programas descritos no modelo #. Embora não sejam equivalentes a máquinas de Turing, as quais são capazes de descrever quaisquer padrões de interação e organização topológica que podem ser encontrados em programas paralelos, redes de Petri são suficientemente expressivas para aplicação dentro do contexto de aplicações de alto desempenho, as quais em geral descrevem padrões regulares e estáticos de interação de processos e organização topológica. Redes de Petri podem ainda ser aplicadas para predição de custos de execução e comunicação de programas, utilizando algumas de suas variantes de alto nível, como, por exemplo, redes de Petri estocásticas e temporizadas. Os aspectos relativos ao uso de redes de Petri em ambientes de desenvolvimento baseados no modelo # serão discutidos em detalhes no Capítulo 5.

Em programas #, a hierarquia de processos garante que a especificação das *computações*, realizada por unidades sequenciais (processos) programadas com a linguagem *host*, encontra-se hierquicamente separada da especificação da *coordenação* entre estas por meio da linguagem de configuração. As vantagens resultantes devido ao suporte à hierarquia de processos são destacadas a seguir:

- A avaliação de desempenho e análise de propriedades formais de um programa paralelo podem ser realizadas em nível de coordenação, abstraindo-se quaisquer suposições a cerca da implementação das computações que caracterizam a funcionalidade das unidades sequenciais. De maneira independente, é também possível realizar a análise de propriedades formais e avaliação de desempenho de cada unidade sequencial, em nível de computação, utilizando formalismos possivelmente distintos daquele aplicado em nível de coordenação, porém apropriados à linguagem *host* empregada;

- Aproveitamento do *estado da arte* da tecnologia de compilação sequencial de programas, uma vez que a compilação das unidades sequenciais pode ser realizada de maneira independente a sua configuração na rede de processos, utilizando um compilador sequencial pré-existente, sem necessidade de modificá-lo. O uso de compiladores sequenciais eficientes tem impacto importante sobre o desempenho de programas #, uma vez que o modelo # favorece a descrição de programas paralelos de média e grossa granularidade. Devido a essa característica, o tempo de execução desperdiçado no modo sequencial é predominante. Uma biblioteca de passagens de

mensagens pode ser usada para gerenciamento do paralelismo. Neste trabalho, foi adotada MPI (*Message Passing Library*) [73], por esta ter se tornado um padrão *de facto* em *cluster computing*, sendo bem documentada e reconhecidamente eficiente. Os construtores do modelo # têm tradução direta para as primitivas de MPI;

Por favorecer a construção de programas paralelos de granularidade média e grossa e a minimização da sobrecarga do gerenciamento do paralelismo em sua implementação, o modelo # é apropriado para aplicação sobre *clusters* de computadores pessoais conectados por interfaces de rede convencionais, como Ethernet. Nesses ambientes, a latência de comunicação entre os nós de processamento é um fator crítico que pode afetar o desempenho de programas.

O estilo de programação característico do modelo # emprega fortemente a noção de composição hierárquica de programas e o conceito de esqueletos [63], os quais potencializam o reuso de parte de programas no nível de coordenação e a portabilidade entre arquiteturas.

Nas seções que se seguem, descreve-se detalhadamente o modelo #, introduzindo-se as abstrações suportadas por este modelo, as quais capturam a estrutura essencial presente na especificação de programas paralelos sob a perspectiva do meio de coordenação. Posteriormente, descrevemos a linguagem Haskell$_{\#}$, uma materialização do modelo #, a qual emprega a linguagem # para configuração de processos em nível de coordenação e a linguagem funcional Haskell para especificação das unidades sequenciais (módulos funcionais), em nível de computação. Respectivamente, introduziremos os construtores da linguagem #, sua tradução para MPI, e alguns exemplos simples de programas. Exemplos mais complexos, especialmente com o emprego de esqueletos e composição hierárquica serão ilustrados no capítulo 4 e apêndices.

3.1 AS PEÇAS BÁSICAS (COMPONENTES)

Componentes são abstrações para as entidades # que implementam funcionalidades, as quais, quando compostas, descrevem as computações realizadas pelo programa paralelo. Idealmente, cada componente implementa uma funcionalidade específica, sendo descrito unicamente pela sua interface, composta por argumentos e pontos de retorno (Figura 3.1).

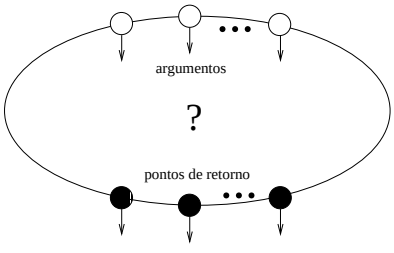


Figura 3.1. Componente

Com relação a implementação, existem dois tipos de componentes: *simples* e *compostos*. Componentes simples são programados com uso da linguagem *host*, descrevendo

computações sequenciais que caracterizam o meio de computação. Componentes compostos descrevem computações paralelas, sendo programados por meio da linguagem #, caracterizando o meio de coordenação. Portanto, estes são constituídos a partir de outros componentes, simples ou compostos, descrevendo uma hierarquia possivelmente aninhada. É fácil observar que, nessa hierarquia, os componentes localizados nas folhas são simples, enquanto aqueles localizados em nós intermediários são compostos.

Um programa # é definido por um componente em particular, o *componente de aplicação*, o qual implementa a funcionalidade da aplicação. Note que um componente de aplicação simples descreve um programa # sequencial, enquanto um componente de aplicação composto descreve um programa # paralelo.

3.2 ESTRUTURANDO COMPONENTES COMPOSTOS

Nesta seção, são introduzidas as abstrações usadas em nível de coordenação para a composição de componentes compostos a partir de outros componentes. Em nível de coordenação, o modelo # captura os aspectos essenciais à atividade de programação paralela, abstraindo-se dos aspectos relacionados a descrição de computações.

Chamamos de *configuração* o programa que define um componente composto. Uma configuração # constitui-se de um conjunto de declarações, as quais descrevem uma rede de unidades, abstrações para entidades executáveis.

A primeira declaração em uma configuração # é o seu cabeçalho, cuja estrutura sintática é descrita como se segue:

component $MyName < par_1, par_2, \ldots, par_n > \#\ (t_1, t_2, \ldots, t_m) \rightarrow (x_1, x_2, \ldots, x_p)$ **with**

Os items no cabeçalho são assim descritos:

- *MyName*: O nome, ou identificador, do componente;
- $par_1, par_2, \ldots, par_n$: Parâmetros formais estáticos da configuração, cujos valores atuais devem ser providos em tempo de compilação;
- $t_1, t_2, \ldots, t_m$: Os argumentos do componente composto;
- $x_1, x_2, \ldots, x_p$: Os pontos de retorno do componente composto.

A partir do cabeçalho, a ordem em que as demais declarações aparecem em uma configuração é irrelevante. Nas seções que se seguem, ao introduzirem-se as abstrações usadas no modelo # para descrição do meio de coordenação, apresentam-se exemplos que ilustram a sintaxe das declarações em uma configuração #. A sintaxe formal desta linguagem é detalhada no Apêndice D.

3.2.1 Interfaces: As Entidades Primitivas do Modelo

No modelo #, a noção de *interface* captura as informações essenciais para caracterização, em nível de coordenação, do comportamento dos processos que compõem um programa paralelo. Uma interface # é definida por uma coleção de portas de entrada e saída. A ordem em que estas são *ativadas* durante a execução descreve o *comportamento* descrito pela interface. Define-se a *ativação de uma porta* como a ação de efetivação de uma operação de comunicação sobre esta. O comportamento descrito por uma interface é configurado por meio de uma expressão regular controlada por semáforos, formalismo o qual foi demonstrado ser equivalente em poder expressivo às redes de Petri rotuladas terminais [126]. Essa equivalência permite a análise automatizada de propriedades de programas # usando este formalismo [52]. Em seguida, formalizaremos alguns aspectos relevantes a respeito da constituição de interfaces, como a definição de operações de composição e noções de equivalência. Posteriormente, descreveremos como efetivamente o comportamento de uma interface é descrito utilizando a linguagem #. Exemplos de declarações de interfaces serão apresentados com a finalidade de demonstrar aspectos práticos de seu uso.

3.2.1.1 Composição de Interfaces O operador # permite que interfaces sejam combinadas para formação de novas interfaces. Formalmente, uma interface, denominada Γ, pode ser definida usando a seguinte notação:

$$\Gamma \triangleright I \rightarrow O : B,$$

onde $I = \{i_1, i_2, \ldots, i_n\}$ é um conjunto de portas de entrada, $O = \{o_1, o_2, \ldots, o_m\}$ é um conjunto de portas de saída e B é o conjunto de sequências de ativação válidas para as portas da interface, o qual pode ser definido pela linguagem formal gerada pela expressão regular sincronizada por semáforos que descreve o seu comportamento, sobre o alfabeto $I \cup O$, cuja definição é introduzida no capítulo 5:

$$B \subseteq (I \cup O)^* \tag{3.1}$$

O operador associativo # é então definido da seguinte forma:

$$\Gamma_1 \# \Gamma_2 = I_1 \cup I_2 \rightarrow O_1 \cup O_2 : B_1 \overline{\odot} B_2,$$

$$\textbf{onde}:$$

$$\begin{array}{l} \Gamma_1 \triangleright I_1 \rightarrow O_1 : B_1 \\ \Gamma_2 \triangleright I_2 \rightarrow O_2 : B_2 \end{array} \tag{3.2}$$

$$(I_1 \cup I_2) \cap (O_1 \cup O_2) = \emptyset$$

A última restrição modela a possibilidade sobreposição de portas das interfaces. Neste caso, uma porta na nova interface pode aparecer na composição de ambas as interfaces originais, com a mesma direção.

O operador $\overline{\odot}$ efetua a *união sincronizada* dos comportamentos das interfaces Γ_1 e Γ_2, de tal forma que qualquer sequência de ativação de portas em $B_1 \overline{\odot} B_2$ seja válida com relação a B_1, ignorando as portas em $I_2 \cup O_2$, e com relação a B_2, ignorando as portas em $I_1 \cup O_1$. O operador $\overline{\odot}$ pode ser formalmente definido da seguinte forma:

$$
\begin{gathered}
L_{\mathsf{SCE}}((w_1 \odot u_1)\ s\ (w_2 \odot u_2)\ s\ \ldots\ s\ (w_n \odot u_n)) \subset B_1 \overline{\odot} B_2 \\
\textbf{onde}: \\
n \geq 1 \\
s \in (I_1 \cup O_1) \cap (I_2 \cup O_2) \\
\begin{gathered} w_1\ s\ w_2\ s\ \ldots\ s\ w_n \in B_1 \\ u_1\ s\ u_2\ s\ \ldots\ s\ u_n \in B_2 \end{gathered} \\
\begin{gathered} w_1\ w_2\ \ldots\ w_n \in ((I_1 \cup O_1) - \{s\})^* \\ u_1\ u_2\ \ldots\ u_n \in ((I_2 \cup O_2) - \{s\})^* \end{gathered}
\end{gathered}
\tag{3.3}
$$

L_{SCE} define a linguagem gerada por uma expressão concorrente sincronizada, cuja definição encontra-se na Seção 5.16. Expressões regulares sincronizadas são casos particulares de expressões concorrentes sincronizadas.

A definição cobre o caso onde não existe sobreposição de portas. Neste caso, o operador $\overline{\odot}$ possui o mesmo efeito do operador $\odot$, o qual representa todas as intercalações possíveis entre as cadeias em B_1 e B_2, podendo ser usado para modelar comportamento concorrente.

3.2.1.2 Equivalência de Interfaces Sejam as interfaces $\Gamma_1 \triangleright I_1 \rightarrow O_1 : B_1$ e $\Gamma_2 \triangleright I_2 \rightarrow O_2 : B_2$. Suponha $\delta_I : I_1 \rightarrow I_2$ e $\delta_O : O_1 \rightarrow O_2$ funções parciais e injetivas. Considere ainda a função total e injetiva $\tau : B_1 \rightarrow I_1 \cup I_2 \cup O_1 \cup O_2$, definida na equação 3.4.

$$
\begin{aligned}
&\tau(\epsilon) = \epsilon \\
&\tau(aw) = \begin{cases} \delta_I(a)\tau(w) & \text{, caso } a \in I_1 \wedge \delta_I \neq \perp \\ \delta_O(a)\tau(w) & \text{, caso } a \in O_1 \wedge \delta_O \neq \perp \\ \tau(w) & \text{, caso } (a \in I_1 \wedge \delta_I = \perp) \vee (a \in O_1 \wedge \delta_O = \perp) \end{cases}
\end{aligned}
\tag{3.4}
$$

Com respeito aos mapeamentos δ_I e δ_O, a interface Γ_1 é parcialmente equivalente à interface Γ_2 se, e somente se, $Im(\tau) \subseteq B_2$, onde $Im(f)$ representa o conjunto imagem da função f (Equação 3.5). Em outros termos, isso equivale a:

$$
\Gamma_1 \sqsubseteq \Gamma_2 \Leftrightarrow \forall w \in B_1 : \tau(w) \in B_2 \tag{3.5}
$$

Isso significa que qualquer sequência de ativação válida para as portas de Γ_1, supondo o mapeamento parcial das portas de Γ_1 às portas de Γ_2, é uma sequência válida em Γ_2. Assumindo-se que os mapeamentos δ_I e δ_O são bijetores, as interfaces Γ_1 e Γ_2 são *totalmente equivalentes* ($\Gamma_1 \equiv \Gamma_2$) se, e somente se, $\Gamma_1 \sqsubseteq \Gamma_2$, com respeito a δ_I e δ_O, e $\Gamma_2 \sqsubseteq \Gamma_1$ com respeito a ${\delta_I}^{-1}$ e ${\delta_O}^{-1}$.

skip	ação nula	λ	palavra vazia
wait s	primitiva de semáforos	ω_s, $\omega_s \in \Omega$	primitiva de semáforos
signal s	primitiva de semáforos	σ_s, $\sigma_s \in \Omega$	primitiva de semáforos
seq	seqüenciamento	$\cdot$	concatenação
alt	escolha	$+$	união
par	concorrência	$\odot$	intercalação
repeat	repetição	$*$	fecho da concatenação
$p?$	ativação de porta de entrada	p, $p \in \Sigma$	símbolo de alfabeto
$p!$	ativação de porta de saída	p, $p \in \Sigma$	símbolo de alfabeto

Tabela 3.1. Combinadores # para Descrição Comportamental de Interfaces e sua Correspodência aos Operadores de Expressões Regulares Controladas por Semáforos

3.2.1.3 Descrevendo o Comportamento de uma Interface na Linguagem

Em um programa paralelo, uma interface caracteriza o comportamento de uma classe de processos (unidades executáveis) instanciados a partir desta. Portas de entrada e saída da interface caracterizam informações que são trocadas entre pares de processos, com semântica bem definida. Entende-se que a informação essencial à descrição do comportamento de um processo em nível de cordenação pode ser caracterizada pela ordem em que as operações de comunicação (ativações de portas) ocorrem. A linguagem # oferece portanto o suporte à descrição comportamental de processos através de interfaces. Para isso, são definidos uma coleção de combinadores, os quais possuem correspondência com os operadores usados para composição de expressões regulares sincronizadas por semáforos (Tabela 3.1). A sintaxe abstrata para descrição comportamental de uma interface # pode ser caracterizada pela seguinte gramática:

$< behavior > ::= (\textbf{sem}\ \{s^+\};)^?\ < action >$

$< action > ::= pid?$
$\quad | \ pid!$
$\quad | \ \textbf{wait}\ s$
$\quad | \ \textbf{signal}\ s$
$\quad | \ \textbf{par}\ \{\ < action > (; < action >)^*\ \}$
$\quad | \ \textbf{seq}\ \{\ < action > (; < action >)^*\ \}$
$\quad | \ \textbf{alt}\ \{\ < action > (; < action >)^*\ \}$
$\quad | \ \textbf{repeat}\ < action > \textbf{until}\ < disjunction >$
$\quad | \ \textbf{skip}$

$< disjunction > ::= < conjunction > (\ |\ (\ < conjunction > \ ))^*$

$< conjunction > ::= < ports >\ \ (\ \&\ \ < ports >)^*$

$< ports > ::= <pid\ (\ \&pid)^* > \ |\ pid$

onde s denota um identificador de *semaforo* and pid um identificador de porta.

A seguir, é apresentada informalmente a semântica dos combinadores # que modelam o comportamento de interfaces, apresentados na Tabela 3.1:

- **seq**: Descreve ações definidas pela composição sequencial de uma coleção de ações;
- **par**: Descreve ações definidas pela composição concorrente (intercalação) de uma coleção de ações;
- **alt**: Descreve ações definidas pela escolha não-derterminística de uma ação entre uma coleção de ações.

- **repeat**: Ação que descreve a execução repetida de uma ação, até a satisfação de *condição de terminação*, opcionalmente atribuída pelo programador através da cláusula **until**. A condição de terminação é descrita por um predicado lógico sobre identificadores de portas na sua forma normal disjuntiva (FND). São suportados os conectores & ("**e**" lógico) e | ("**ou**" lógico). Os delimitadores < e > são usados, nas disjunções, para expressar sincronização de portas. A semântica de terminação de uma repetição é intimamente dependente da noção de porta *stream*, sendo detalhada na seção 3.2.1.5. Em alternativa à cláusula **until**, o programador pode usar a cláusula **count**, informando um número fixo de repetições para a ação. Caso nenhuma das cláusulas que delimitam o número de repetições seja informada, a ação é repetida infinitamente;

- **p?**: Ação de ativação de uma porta de entrada p. A operação completa-se quando um valor é recebido através da porta;

- **p!**: Ação de ativação de uma porta de saída p. A operação completa-se quando um valor é enviado através da porta;

- **wait** *sem*: Corresponde a primitiva *wait* aplicada a um semáforo contador *sem*. A ação é completada até que um número equivalente de ações *signal* tenham sido completadas no semáforo *sem*;

- **signal** *sem*: Corresponde a primitiva *signal* aplicada a um semáforo contador *sem*;

- **skip**: Ação sem efeito, também conhecida como "ação nula";

3.2.1.4 Portas *Stream* Em uma interface #, uma porta pode ser de dois tipos: *stream* ou não-*stream*. Portas deste último tipo descrevem a comunicação de um valor único do tipo atribuído à porta, sendo portanto ativadas uma única vez na execução do processo. Portas *stream* são usadas para transmissão de sequências de valores (*streams*) do tipo da porta, podendo ser ativadas várias vezes no contexto de ocorrências do combinador **repeat**. Um valor marcador especial deve ser usado para indicar o final de uma *stream*, o qual denominaremos EOS[1]. *Streams* podem ser aninhadas a outras *streams*, em profundidades arbitrárias, porém pré-estabelecidas, de aninhamento. Para suporte a essa funcionalidade, a configuração de uma porta *stream* (aninhada) deve informar o *fator de aninhamento* da *stream* transmitida. Além disso, é necessário usar a notação EOS i para indicar um marcador de final de uma *stream* aninhada a profundidade i. Portas que transmitem *streams* aninhadas devem ser ativadas no contexto de ocorrências do combinador **repeat** aninhadas a outras ocorrências desse mesmo combinador na profundidade correspondente ao aninhamento da *stream* em questão.

Por exemplo, considere uma porta *stream* p cujo fator de aninhamento informado é 3. Considerando a linguagem Haskell para descrever valores a serem transmitidos, a porta p pode ser usada na transmissão de listas do tipo `[[[`t`]]]` entre dois processos, sob demanda. A cada ativação, um valor do tipo t é transmitido. Vale ressaltar que t, sendo

[1] *End of stream.*

um tipo polimórfico, também pode ser uma lista. Entretanto, neste caso, a lista completa é transmitida na ativação. Suponha t = `(Int,Int)`. Considere o seguinte valor do tipo `[[[`t`]]]`:

```
[[[(1,2),(3,4)],[(5,6)]],[[(7,8),(9,0),(1,2)]],[[],[(3,4)],[(5,6),(7,8)]]]
```

Os valores efetivamente transmitidos através da porta *stream* p como efeito de cada ativação desta durante a execução do processo são:

```
{(1,2), (3,4), Eos 3, (5,6), Eos 3, Eos 2, (7,8), (9,0), (1,2), Eos 3,
 Eos 2, Eos 3, (3,4), Eos 3, (5,6), (7,8), Eos 3, Eos 2, Eos 1}
```

Cada lista Haskell em aninhamento i é transformada em uma *stream* em aninhamento i, sendo o marcador `Eos` i, onde i corresponde ao aninhamento da lista, usado para indicar o final da *stream* a cada final de lista.

3.2.1.5 Terminação de Repetição Uma vez introduzido o conceito de porta *stream*, é possível definir a semântica de terminação de ações de repetição (ocorrências do combinador **repeat**). Essencialmente, o combinador **repeat** oferece o suporte necessário para o controle da transmissão de *streams* através da ativação de portas em seu contexto.

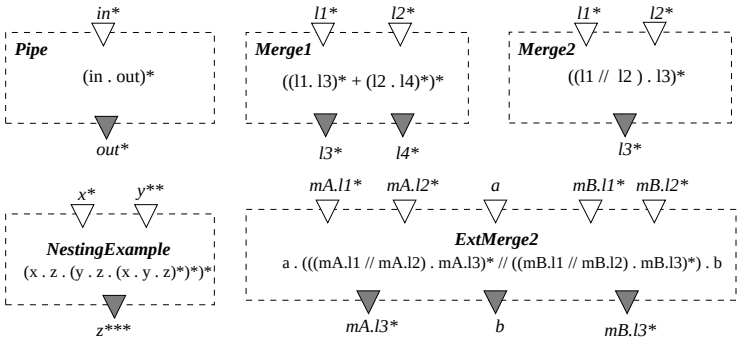
Em ocorrências do combinador **repeat**, uma condição de terminação pode ser especificada em termos de um predicado lógico em sua forma normal disjuntiva, onde as *variáveis lógicas* correspondem a identificadores de portas. São suportados os conectivos lógicos & ("*e*" lógico) e | ("*ou*" lógico). Os delimitadores $<$ e $>$, cuja semântica é detalhada adiante, pode ser usado para estabelecer a sincronização de *streams* transmitidas por portas distintas. A condição de terminação é testada ao final de cada iteração.

Seja d a profundidade de aninhamento de uma ocorrência R de um combinador **repeat** em relação às ocorrências mais externas de combinadores **repeat** que o englobam, se estes existirem. Caso contrário, $d = 1$. Somente identificadores de portas *stream* de fator de aninhamento igual ou menor que d podem aparecer na condição de terminação de R, referida como C na discussão que se segue. É necessário definir a valoração de uma variável lógica (identificador de porta) em C durante a execução do processo. Seja p uma porta *stream* de fator de aninhamento n, referenciada em C ($n \leq d$). No momento em que o valor de C é calculado, o valor da variável lógica associada à p é *verdadeiro* sempre que um marcador de final de *stream* Eos i, onde $i \geq d$, foi transmitido como efeito da sua última ativação. Caso contrário, o valor é *falso*. Assumindo a semântica dos conectivos & e | na lógica proposicional, porém ainda ignorando a ocorrência dos delimitadores $<$ e $>$, se C é verdadeiro então a ação de repetição termina.

A condição para a realização de uma nova iteração é obtida calculando-se a forma normal disjuntiva da negação da condição C, aqui denominada $\overline{C}$. Entretanto, nesta transformação, a identidade 3.6, a qual define o significado dos delimitadores $<$ e $>$, deve ser aplicada.

$$\neg < a_1 \wedge a_2 \wedge \ldots \wedge a_n >= \neg a_1 \wedge \neg a_2 \wedge \ldots \wedge \neg a_n \tag{3.6}$$

Se a condição $\overline{C}$ for satisfeita, uma nova iteração é executada. Caso contrário, um erro em tempo de execução ocorre ($\neg C \wedge \neg\overline{C}$). Essa condição de erro indica o não cumprimento a restrição imposta pela ocorrência dos delimitadores $<$ e $>$ nas conjunções de C. As portas que aparecem no contexto destes delimitadores devem estar sincronizadas a cada iteração, com respeito a natureza do valor transmitido por cada uma destas. Neste caso, ao final da iteração, todas as portas devem ter transmitido um valor `Eos` i, $i < n$, ou um valor de dados, como efeito de sua mais recente ativação. Em qualquer outro caso, um erro de execução é informado. No Capítulo 5, formalizaremos a semântica de terminação de repetições por meio de redes de Petri.




```
interface Pipe t u # in*::t → out*::u
                  behavior: repeat seq {in?; out!} until <in & out>

interface Merge1 t u # l1 → l3 like Pipe t u
                     # l2 → l4 like Pipe t u
                     behavior: repeat alt { seq {l1?; l3!};
                                            seq {l2?; l4!} } until <l1 & l2>

interface Merge2 t u # l1 → l3 like Pipe t u
                     # l2 → l3 like Pipe t u
                     behavior: repeat seq {par {l1?;l2?}; l3! } until <l1 & l2>

interface ExtMerge2 # (a::Int) → (b::[Int])
                    # mA like Merge2 Int Double
                    # mB like Merge2 Int Double
                    behaviour: seq { a?; par {do mA; do mB}; b! }

interface NestingExample t
                  # (x*,y**::t) → (z***::t)
                  behaviour: repeat seq {x?; z!;
                                         repeat seq {y?; z!;
                                                     repeat seq {x?; y?; z!} until z}
                                         until <y & z>}
                             until <x & y & z>
```

Figura 3.2. Exemplos de Interfaces

3.2.1.6 Exemplos de Declarações de Interfaces A Figura 3.2 apresenta exemplos de declarações de interfaces em uma configuração #, explorando alguns aspectos importantes a respeito de sua sintaxe. A primeira interface é nomeada *Pipe*, constituída por uma porta de entrada, de nome *in* e tipo t (polimórfico), e uma porta de saída, nomeada *out* e de tipo u. As variáveis de tipos t e u constituem parâmetros da interface. Ambas as portas são do tipo *stream* e possuem fator de aninhamento 1, o que é indicado pela ocorrência de um único símbolo "*" após seus respectivos identificadores. O comportamento da interface *Pipe* descreve a ativação sequencial e repetida das portas *in* e *out*, até

que as *streams* transmitidas por estas portas atinjam o final.

A interface *Merge1* constitui-se de quatro portas: duas portas de entrada, nomeadas $l1$ e $l2$, e duas portas de saída, nomeadas $l3$ e $l4$, definidas a partir da composição de interfaces que herdam características da interface *Pipe*. Essas restrições exigem que o comportamento de *Merge1* obedeça a seguinte restrição de equivalência parcial:

$$Merge1 \triangleright (l1, l3) \rightarrow (l2, l4) : B^{Merge1} \sqsubseteq Pipe \triangleright l_1 \rightarrow l_3 : B_1^{Pipe} \# Pipe \triangleright l_2 \rightarrow l_4 : B_2^{Pipe} \quad (3.7)$$

Na restrição acima, os conjuntos B_1^{Pipe} e B_2^{Pipe} correspondem às sequências válidas de ativações de portas descrita pelo comportamento da interface *Pipe*, assumindo-se, respectivamente, os pares de portas $l1/l3$ e $l2/l4$ no papel das portas *in* e *out* desta interface. O conjunto B^{Merge1} corresponde ao conjunto de sequências válidas de ativações de portas descritas pela interface de *Merge1* sobre as portas $l1$, $l2$, $l3$ e $l4$. A equivalência definida na Equação 3.7 diz respeito à funções de mapeamento (δ_I e δ_O) que identificam as portas correspondentes das duas interfaces. Particularmente para o caso de *Merge1*, é facilmente observável que:

$$Merge1 \triangleright (l1, l3) \rightarrow (l2, l4) : B^{Merge1} \equiv Pipe \triangleright l_1 \rightarrow l_3 : B_1^{Pipe} \# Pipe \triangleright l_2 \rightarrow l_4 : B_2^{Pipe} \quad (3.8)$$

A interface *Merge2* ilustra uma sobreposição de portas, representada pela porta $l3$. A restrição comportamental de *Merge1* pode ser vista como uma versão mais forte da restrição comportamental imposta a *Merge2*, onde:

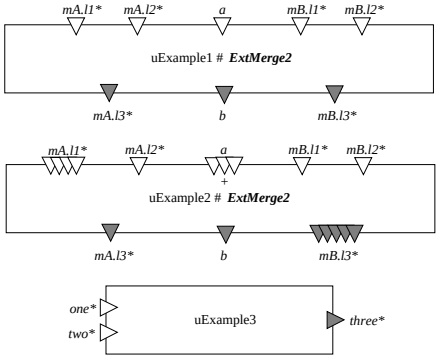
$$Merge2 \triangleright (l1, l2) \rightarrow l3 : B^{Merge2} \equiv Pipe \triangleright l_1 \rightarrow l_3 : B_1^{Pipe} \# Pipe \triangleright l_2 \rightarrow l_3 : B_2^{Pipe} \quad (3.9)$$

Assumindo o comportamento da interface *Pipe*, as sequências de ativação em *Merge2* são restringidas àquelas onde a cada ativação de $l3$ precedem ativações de $l1$ e $l2$, em ordem arbitrária.

A interface *ExtMerge2* é composta pela composição de interfaces instanciadas a partir de *Merge2*, com a adição de duas portas não-*stream*, *a* e *b*, respectivamente de tipos `Int` e `[Int]`. Os identificadores *mA* e *mB* podem ser usados para referenciar as portas herdadas de *Merge2*. O uso do combinador **do** indica que o comportamento induzido para a interface é literalmente substituído na posição onde ocorre. Dessa forma, o comportamento *ExtMerge2* é equivalente a:

```
seq { a?;
      repeat seq {par {mA.l1?;mA.l2?}; mA.l3! } until <mA.in1 & mA.in2>
      repeat seq {par {mB.l1?;mB.l2?}; mB.l3! } until <mB.in1 & mB.in2>
      b! }
```

A interface *NestingExample* ilustra o uso de portas *stream* com fator de aninhamento maior que 1, ativadas no contexto de ocorrências aninhadas de combinadore **repeat**. Observe que o número de símbolos "*" depois do identificador da porta identifica seu fator de aninhamento. Note ainda o atendimento a restrição que exige que em uma condição de terminação de uma ocorrência do combinador **repeat** em profundidade de aninhamento d, somente portas com fator de aninhamento maior ou igual que d podem ser referenciadas como variáveis lógicas.



unit uExample1 # *ExtMerge2*
unit uExample2 # *ExtMerge2* **groups** a<3>: **choice**, mA.l1<4>, mB.l3<5>:*distribute*
unit uExample3 # (one*::Int,two*::Int) → three*::Int **behaviour**: **repeat** {one?;two?;three!} **until** one

Figura 3.3. Unidades instanciadas a partir de interfaces

3.2.1.7 Especialização e Generalização de Interfaces No modelo #, *interfaces* estão para *unidades* assim como *tipos de dados* estão para *valores* em uma linguagem de programação convencional ou *classes* estão para *objetos* em linguagens orientadas à objetos. Os experimentos realizados com a modelagem de aplicações paralelas comuns com o modelo # sugeriram a necessidade de um mecanismo para abstração de interfaces, semelhante ao que a herança de classes oferece dentro do modelo de programação orientado à objetos. Com essa finalidade, interfaces podem ser definidas como *especializações* de outras interfaces, através do uso da cláusula **specializes** na sua declaração. Observe o exemplo abaixo:

interface *IPipe_Ext* # arg → ret
pipe **like** *IPipe*
behaviour: **seq** {initial?; **do** pipe}; final!}
specializes *IPipe* # pipe

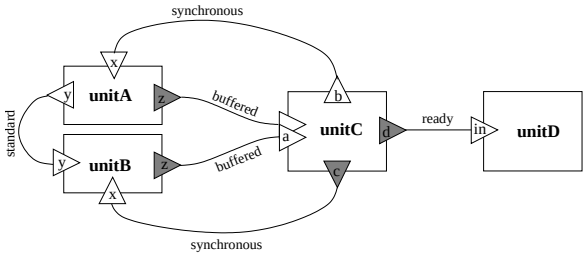
A interface *IPipe_Ext* é declarada como uma especialização da interface *IPipe*, incluindo duas portas adicionais, denominadas *arg* e *ret*. Uma interface pode ser uma especialização de uma ou mais interfaces, sendo necessários especificar-se o mapeamento de suas portas para cada especialização. A interface a partir da qual um conjunto de interfaces é especializada é dita uma generalização destas. O comportamento das interfaces especializadas deve ser compatível com o comportamento da interface mais geral, segundo as regras de equivalência parcial descritas anteriormente. A diferença da especialização de interfaces para o mecanismo de composição com o operador # reside no fato de que interfaces especializadas podem substituir as interfaces gerais nas situações onde estas ocorrem, especialmente na manipulação de unidades virtuais. No exemplo proposto, ambos os mecanismos são usados na declaração da interface *IPipe_Ext*. Os esqueletos MPI e a aplicação LU, introduzidos no Capítulo 5 e cujos códigos # encontram-se listados nos Apêndices A e B, respectivamente, fazem uso deste recurso para modularizar e simplificar seu código.

3.2.2 As Entidades Executáveis (Unidades)

No modelo #, *unidades* constituem abstrações para entidades executáveis (processos) em programas paralelos. São instanciadas a partir de interfaces, capturando seu conjunto de portas e comportamento. Dessa forma, unidades instanciadas a partir de uma mesma interface compartilham as mesmas características comportamentais em nível de coordenação.

Na configuração de uma unidade, é possível replicar portas de sua interface, constituindo *agrupamentos de portas*. Estes são tratados individualmente com respeito à ativação. As portas pertencentes a um agrupamento são identificadas individualmente por meio de índices quando referenciadas na configuração de canais de comunicação.

A Figura 3.3 ilustra a instanciação de duas unidades a partir da interface *ExtMerge2* (Figura 3.2). No segundo exemplo (*uExample2*), é ilustrada a sintaxe para replicação de portas na declaração de uma unidade, usando a cláusula **groups**. No exemplo em questão, as portas a, $mA.l1$ e $mB.l3$ são replicadas. O número entre os delimitadores $<$ e $>$ corresponde ao número de replicações da porta. Os índices que identificam individualmente as portas pertencentes a um mesmo agrupamento variam de 1 a n, onde n é o número de portas membro do agrupamento. Opcionalmente, após o símbolo ':', uma *função de ligação* é declarada, cujo significado será explicado na Seção 3.2.5.1. A unidade *uExample3* ilustra a declaração de uma interface diretamente no corpo da declaração da unidade. Este recurso é útil para simplificar o código # de um programa quando a interface de uma unidade não é compartilhada com outras unidades, não havendo necessidade de serem reusadas em outras configurações. Observe ainda que os nomes das portas da unidade são os mesmos da interface a partir da qual foi instanciada.




```
connect unitA→y to unitB←y standard
connect * unitA→z to unitC←a[1] buffered
connect * unitB→z to unitC←a[2] buffered
connect unitC→b to unitA←x synchronous
connect unitC→c to unitB←x
connect * unitC→d to unitD←in ready
```

Figura 3.4. Canais de Comunicação

3.2.3 Construindo a Rede de Communicação (Canais)

As unidades que constituem uma configuração sincronizam por meio de canais de comunicação, os quais interligam suas portas individuais. Canais # são ponto-a-ponto,

unidirecionais e tipados, ao estilo de OCCAM [124]. O *par de comunicação* de uma porta é definido como a porta a ela conectada por meio de um canal. A um canal # está associado um dentre três modos de comunicação possíveis, inspirados nos modos de comunicação suportados por MPI [74]. Esse recurso garante que operações de comunicação em canais # possam ser traduzidas diretamente para chamadas às primitivas de comunicação desta biblioteca. Os modos de comunicação de canais # são os seguintes:

- **synchronous** (*default*): O processo emissor permanece bloqueado até que o receptor copie a mensagem para o seu *buffer* de recebimento ;
- **buffered**: O processo emissor copia a mensagem a ser enviada em um *buffer* de envio e prossegue sua execução. O sistema em tempo de execução, concorrentemente, aguarda que o receptor da mensagem copie a mensagem;
- **ready**: Assume-se que o processo receptor está sempre pronto no momento do envio de uma mensagem pelo processo transmissor. Caso contrário, é anunciado um erro em tempo de execução.

A Figura 3.4 apresenta a topologia de uma configuração # hipotética, ilustrando a declaração de canais de comunicação interligando unidades. O símbolo "*", após a cláusula **connect**, caracteriza o canal como um canal que transmite uma *stream*. Somente portas *stream* podem ser conectadas por meio de canais *stream*.

3.2.4 Configurando Argumentos e Pontos de Retorno do Componente

Os argumentos e pontos de retorno declarados no cabeçalho da configuração de um componente devem ser associados a portas de entrada e saída, respectivamente, pertencentes a unidades declaradas na configuração. Estas portas não devem estar conectadas a nenhuma outra porta por meio de canais. A declaração **bind** é empregada com este propósito, como ilustrado no seguinte exemplo:

```
component PIPE<N> in → out with
⋮
bind pipe[1]←in to in
bind pipe[N]→out to out
```

Neste, a porta de entrada *in* da unidade *pipe*[1] é associada ao ponto de entrada *in* do componente PIPE. Analogamente, a porta de saída *out* da unidade *pipe*[*n*] é associada ao ponto de saída *out* do mesmo componente.

3.2.5 Especificando a Computação Realizada por uma Unidade

A configuração de uma unidade deve ainda descrever a computação que esta deve realizar. Para isso, a cada unidade é associada um componente, o qual descreve sua funcionalidade. A compatibilização entre o componente e a unidade é concretizada pela

definição de um mapeamento injetivo entre os argumentos e pontos de retorno do componente às portas de entrada e saída da interface da unidade, respectivamente. Um argumento associado a uma porta de entrada deve receber seus valores atuais a partir desta, como efeito colateral de sua ativação. Quando não associado a uma porta de entrada, um argumento deve receber seu valor explitamente. Analogamente, um valor de retorno associado a uma porta de saída deve ser enviado através desta, quando ativada. Caso contrário, é ignorado.

Ainda com respeito a compatibilidade da associação entre componentes e unidades, a ordem de ativação descrita para as portas da interface da unidade deve ser uma ordem válida com respeito a demanda de consumo dos argumentos e produção dos valores de retorno do componente, obedecendo o mapeamento especificado. Dessa forma, sempre que, durante a execução de uma unidade, um valor de um argumento do componente é requerido, a porta de entrada associada ao argumento deve ter sido ativada em um momento anterior, de forma que o valor esteja disponível. Analogamente, no momento da ativação de uma porta de saída, um valor deve ter sido produzido no ponto de retorno correspondente. A incompatibilidade entre componente e unidade causa um erro em tempo de execução. Entretanto, o modelo # permite que erros desse tipo possam ser apresentados de forma compreensível ao especialista humano, facilitando a depuração de erros lógicos em tempo de execução, uma das vantagens dessa abordagem em relação a programação MPI pura, onde a dificuldade de depuração de programas constitui um de seus pontos fracos.

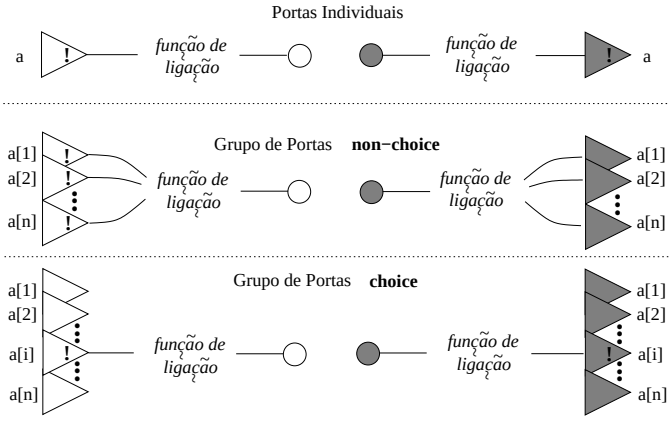
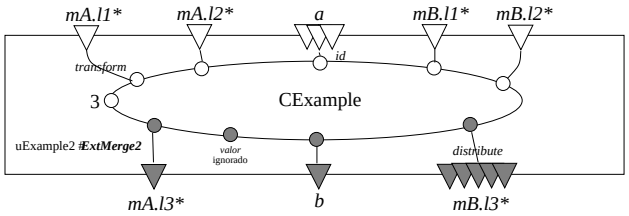


Figura 3.5. Agrupamentos de Portas

3.2.5.1 Funções de ligação são suportadas para aumentar a capacidade de compatibilização entre componentes e unidades. São associadas a portas de entrada e saída na configuração de unidades. Em portas de entrada, a função de ligação é aplicada ao valor recebido através desta. O valor retornado é então passado ao argumento associado à porta em questão. Analogamente, em portas de saída, a função de ligação é aplicada ao valor produzido no ponto de retorno associado. O valor retornado é enviado através da porta de saída em questão. Assume-se a função identidade (id) como *default*, sempre

que a função de ligação não é especificada explicitamente.

Funções de ligação são particularmente úteis quando associadas a agrupamentos de portas. Com relação ao tipo de função de ligação associado a um agrupamento, este pode ser classificado como *choice* ou *non-choice*. Em agrupamentos de portas de entrada do tipo *non-choice*, a função de ligação mapeia o conjunto formado pelos valores recebidos por cada porta individual pertencente ao agrupamento em um único valor, o qual é passado ao argumento associado. Em agrupamentos de portas de saída do tipo *non-choice*, a função de ligação mapeia o valor único produzido em um ponto de retorno em um conjunto de valores, enviados por cada uma das portas individuais do agrupamento. Em agrupamentos de portas do tipo *choice*, a função de ligação atua de forma análoga ao descrito para portas individuais. Para que isto seja possível, uma das portas é escolhida não-determiniscamente, dentre aquelas cujo par de comunicação encontra-se ativado. A palavra reservada **choice** é usada para especificar grupos de porta do tipo *choice*, seguida da função de ligação. Sua ausência indica que a porta é do tipo *non-choice*, sendo necessário somente a especificação da função de ligação. A função de ligação *default* para um grupo *choice* é *id*. A Figura 3.5 ilustra a semântica de funções de ligação quando aplicadas a portas individuais ou grupos de portas.



unit uExample2 # *ExtMerge2* **groups** a<3>: **choice**, mA.l1:*transform*, mB.l3<5>:*distribute*
as CExample<5,"uExample2"> (3,a,mA.l1,mA.l2,mB.l1,mB.l2) → (mA.l3,mB.l3, _, b)

Figura 3.6. Associação de Componentes à Unidades

3.2.5.2 Exemplo A Figura 3.6 ilustra a associação entre unidades e componentes. A unidade `uExample2`, instanciada a partir da interface *ExtMerge2* (Figura 3.2), é associada ao componente `CExample`. Ao quarto argumento do componente `CExample` é atribuído um valor explícito (3), não sendo este portanto associado a uma porta de entrada. O mesmo pode ser afirmado com relação ao segundo ponto de retorno deste componente, cujo valor é ignorado. Os valores entre os delimitadores $<$ e $>$ são os parâmetros estáticos atuais passados ao componente e podem ser omitidos quando o componente não os exige. As portas a e $mB.l3$ são replicadas de maneira a formar agrupamentos de 3 e 5 portas, respectivamente. O agrupamento de portas $mB.l3$ é do tipo *non-choice*, com função de ligação *distribute*, enquanto o agrupamento a é do tipo *choice* com função de ligação *default* (*id*, ou identidade). À porta $mA.l1$ é especificada a função de ligação *transform*.

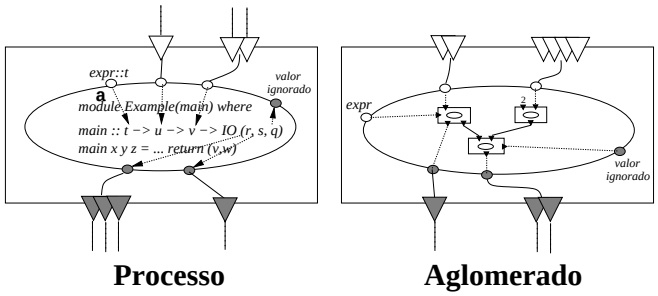


Figura 3.7. Aglomerados e Processos

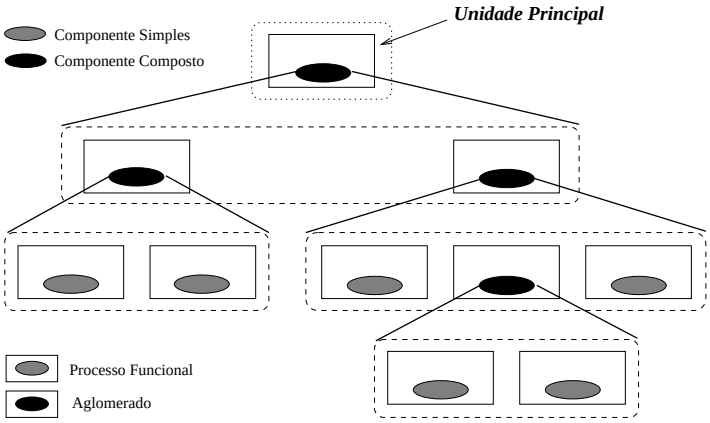


Figura 3.8. Hierarquia de Unidades em um Programa #

3.2.5.3 Aglomerados e Processos Unidades associadas a componentes compostos são denominadas *aglomerados*, enquanto aquelas associadas a componentes simples são denominadas *processos*. Portanto, aglomerados são unidades que implementam computações paralelas, enquanto processos implementam computações sequenciais. A Figura 3.7 ilustra o mapeamento de argumentos e pontos de retorno em aglomerados e processos. A capacidade de associar-se unidades a componentes simples, formando processos, constitui a "cola" que une os meios de coordenação e computação no modelo #. Na Figura 3.8, ilustra-se a hierarquia de unidades em uma aplicação. As unidades representadas nos nós da árvore são aglomerados, enquanto unidades representadas em suas folhas constituem processos.

3.2.6 Unidades Repetitivas e Não-Repetitivas

Processos podem ser classificados como *não-repetitivos* ou *repetitivos*. Em processos não-repetitivos, o estado final é atingido após a primeira avaliação de sua função *main*, enquanto em processos repetitivos esta é avaliada repetidamente e sequencialmente. Um aglomerado é repetitivo quando todas as unidades que o compõem são repetitivos. Um programa # cuja unidade *main* é repetitiva é chamado um programa repetitivo e não é capaz de atingir um estado de terminação. Um programa # termina quando todas

as unidades não-repetitivas que o constituem terminam. Um exemplo de declaração de unidade repetitiva é apresentado a seguir:

unit * sensor # () → data **behaviour**: data! **as** Sensor

O símbolo "*" caracteriza a unidade *sensor* como repetitiva. Em cada ativação, a unidade *sensor* envia um valor através de sua única porta não-*stream*, de nome *data*.

3.2.7 Unidades Virtuais

Não é obrigatório associar-se um componente a uma unidade. Unidades que não especificam um componente são denominadas *unidades virtuais*. Estas descrevem entidades executáveis parametrizadas com relação a sua funcionalidade. São usadas para construção de esqueletos topológicos, assunto discutido no Capítulo 4. A um componente que possui pelo menos uma unidade virtual, chamamos *componente abstrato*. A noção de componente abstrato confunde-se com a noção de *esqueleto topológico parcial*, sendo este conceito assim definido. O uso da palavra parcial advém do fato de que em um componente abstrato podem co-existir unidades não-virtuais e virtuais. Um componente abstrato onde todas as unidades são virtuais é considerado um *esqueleto topológico total*. Um componente de aplicação, o qual descreve a funcionalidade de um programa #, não pode ser abstrato, uma vez que sua funcionalidade não está completamente definida.

A operação de *nomeação de unidades* permite a nomeação de uma unidade não-virtual para ocupar o papel de uma unidade virtual na rede de processos de um componente abstrato, em qualquer nível de aninhamento. A unidade original deixa de existir. O comportamento da unidade não-virtual deve ser compatível com o comportamento da unidade virtual substituída, segundo a noção de equivalência parcial introduzida anteriormente. O código a seguir ilustra uma operação de nomeação:

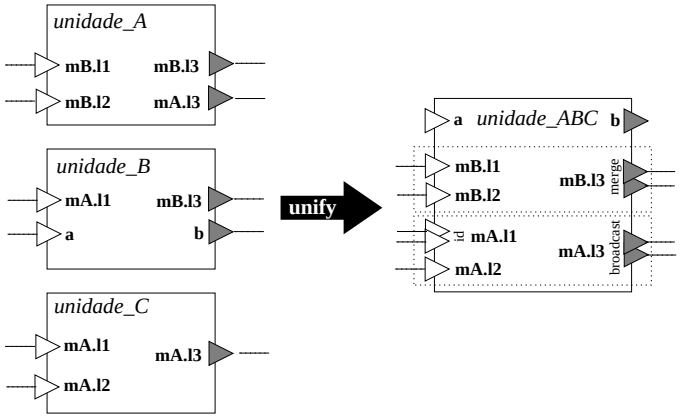
assign uExample2 **to** one_cluster.merge # mA

No exemplo acima, a unidade *uExample2* é nomeada para ocupar o papel da unidade *merge* do aglomerado *one_cluster*. A interface *mA*, a qual compõe a interface *ExtMerge2* de *uExample2*, estabelece o mapeamento das portas de *uExample2* para as portas de *one_cluster*.

Podem ocorrer conflitos no mapeamento das portas da unidade nomeada às portas da unidade virtual substituída, facilmente verificáveis pelo compilador. Estes ocorrem nas seguintes situações e sua resolução é de responsabilidade do programador:

i) As portas associadas constituem agrupamentos com números distintos de portas;

ii) As funções de ligação das portas associadas são distintas;



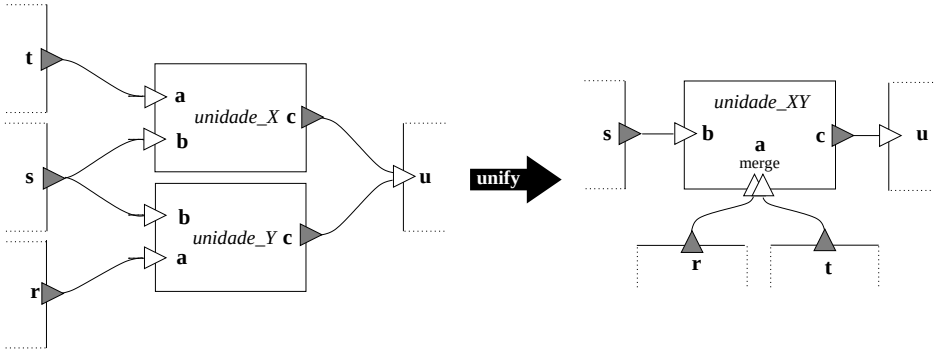
unify unidade_A # mB # () → mA.l3,
unidade_B # mA.l1 → mB.l3,
unidade_C # mA
to unidade_ABC # *ExtMerge2*
connections mA.l1<>:*choice*, mB.l3<>:*merge*, mA.l3<>:*broadcast*

Figura 3.9. Exemplo de Unificação

3.2.8 Operações sobre Unidades

Unidades podem ser compostas, decompostas e replicadas, obedecendo restrições que garantem a preservação do comportamento descrito por suas interfaces. Para isso, o modelo # define três operações que podem ser aplicadas sobre unidades, produzindo novas unidades que as substituem: *unificação*, *fatoração* e *replicação*. As duas primeiras são consideradas operações primitivas e são aplicadas sobre unidades virtuais, produzindo unidades virtuais. A terceira, por outro lado, pode ser aplicada sobre qualquer tipo de unidade, sendo um caso especial de fatoração quando aplicada sobre uma única unidade virtual. As seções seguintes provêem detalhes e exemplos a respeito de cada uma das operações. De forma geral, em uma operação válida, deve haver uma relação de equivalência parcial da(s) unidade(s) resultante(s) às unidade(s) original(is), de forma que o comportamento destas seja parcialmente preservado na rede de unidades resultante após a aplicação da operação.

3.2.8.1 Unificação A unificação aplica-se sobre um conjunto de unidades virtuais, resultando na criação de uma nova unidade virtual que assume o papel das unidades originais na rede de processos, deixando estas de existir. O programador pode definir a *interface* da nova unidade explicitamente ou delegar essa tarefa ao compilador, o qual gerará uma interface *default*. Esta é obtida pela aplicação do operador # sobre as interfaces das unidades originais. As interfaces das unidades originais podem ser sobrepostas na formação da interface da unidade resultante, sendo o agrupamento de portas aplicado sempre que portas unificadas estão conectadas a portas distintas. Exemplo de *unificação* é ilustrado na Figura 3.9. Neste, são unificadas três unidades virtuais, denominadas *unidade_A*, *unidade_B* e *unidade_C*, formando uma nova unidade virtual identificada por *unidade_ABC*.



unify unidade_X # (a,b) → c,
unidade_Y # (b,a) → c
to unidade_XY # (a,b) → c **behaviour**: **repeat seq** {a?;b?;c!} **until** <a & b & c>
connections a:<>:*merging*

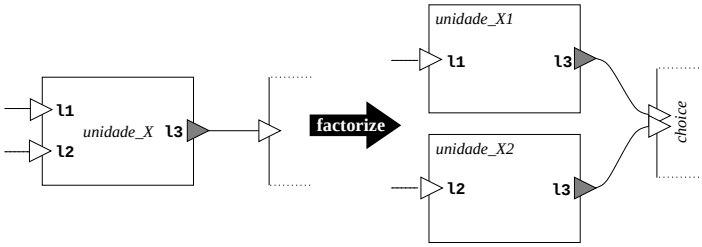
Figura 3.10. Exemplo de Unificação

Observa-se a sobreposição das interfaces das unidades originais na constituição da interface da unidade resultante. Por exemplo, as portas identificadas por $mA.l3$ nas unidades *unidade_A* e *unidade_C* passam a representar na *unidade_ABC* um única porta de entrada de mesmo nome. Como as duas portas $mA.l3$ originais estão conectadas a portas distintas na rede de processos, $mA.l3$ constituirá um agrupamento de portas na unidade resultante. Cada porta no agrupamento ($mA.l3[1]$ e $mA.l3[2]$) faz o papel de uma das portas originais, preservando a conectividade original da rede de unidades. A função de ligação do agrupamento de portas $mA.l3$ na *unidade_ABC* é configurado explicitamente pelo programador (*broadcast*). O mesmo pode ser afirmado com respeito às portas $mA.l1$ e $mB.l3$. Sempre que a função de ligação é omitida, assume-se a função *default*. Atente-se ainda para o fato de que o número de replicações da porta não precisa ser explicitado na declaração do agrupamento. Este corresponde ao número de ocorrências da porta nas interfaces das unidades originais.

Como forma de simplificar o processo de composição de programas utilizando unificação, é possível associar diretamente um componente a unidade resultante de uma unificação, usando notação semelhante a declaração de unidades não-virtuais. Observe o exemplo:

```
unify unidade_A # mB # () → mA.l3,
      unidade_B # (a,mA.l1) → (b,mB.l3),
      unidade_C # mA
      to unidade_ABC # ExtMerge2
                       as AComponent ([1,3..],mB.l1,mA.l1,mA.l2) → (mB.l3,mA.l3,_)
      connections mA.l1<>:choice, mB.l3<>:merge, mA.l3<>:broadcast
```

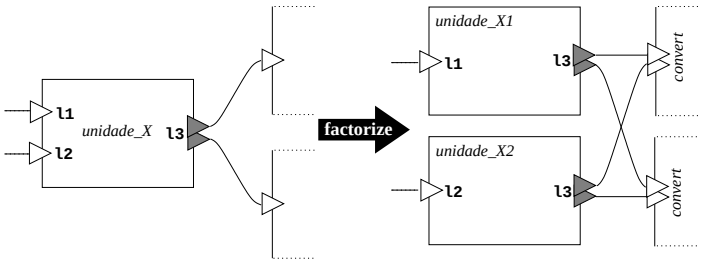
Nesta nova configuração, a *unidade_ABC* é uma unidade não-virtual, provendo a funcionalidade especificada pelo componente *AComponent*. Este recurso sintático evita que o programador precise aplicar o comando **assign** posteriormente, tornando o código # mais conciso.



unit unidade_X # *Merge2*

factorize unidade_X
to unidade_X1 # l1 → l3 **like** *Pipe* , unidade_X2 # l2 → l3 **like** *Pipe*
connections c<>:**choice** *convert*

Figura 3.11. Exemplo de Fatoração



unit unidade_X # *Merge2* **groups** l3<2>:**choice**

⋮

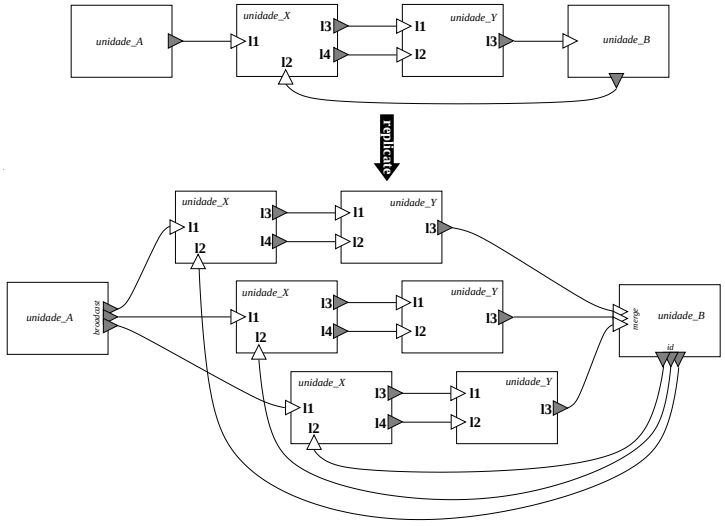
factorize unidade_X
to unidade_X1 # l1 → l3 **like** *Pipe* , unidade_X2 # l2 → l3 **like** *Pipe*
connections c<>:**choice** *convert*

Figura 3.12. Exemplos de Fatoração

É possível que um conjunto de portas unificadas, como resultado da unificação de unidades cujas interfaces sobrepõem-se, não origine um agrupamento na unidade resultante. Isso é possível quando os pares de comunicação das portas unificadas são os mesmos. Caso não houvesse unificação das unidades detentoras dessas portas, o compilador indicaria um erro, uma vez que canais # são ponto-a-ponto. Entretanto, com a unificação, as portas passam a fazer o papel de uma única porta, tornando o canal ponto-a-ponto, como ilustrado na Figura 3.10.

3.2.8.2 Fatoração A operação de fatoração é aplicada sobre uma unidade virtual, produzindo um conjunto de novas unidades virtuais. Exemplos de fatoração são ilustrados nas Figuras 3.11 e 3.12.

Na Figura 3.11, a unidade virtual *unidade_X* é decomposta em duas unidades virtuais, denominadas *unidade_X1* e *unidade_X2*. Observe que a fatoração da porta *l*3 causou a replicação de seu par de comunicação, formando um agrupamento de portas *choice* com função de ligação *id* (*default*).



unit unidade_X # *Merge1*
unit unidade_Y # *Merge2*

.
.
.

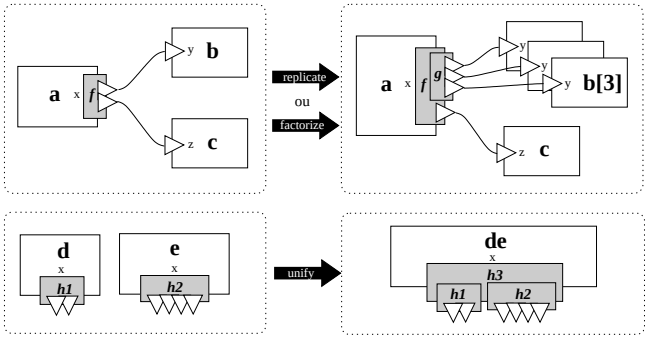
replicate 3 unidade_X , unidade_Y **connections** l1<>: *broadcast*, l2<>: *choice*, l3<>: *merge*

Figura 3.13. Um exemplo de replicação de unidades

Na Figura 3.12, ilustra-se o caso onde a porta $l3$ constitui um agrupamento de portas, ao invés de uma porta individual como no caso anterior. Neste caso, a função de ligação especificada é aplicada a cada uma das portas conectadas às portas individuais em $l3$. Essa regra extende-se para qualquer tipo de agrupamento de portas, inclusive agrupamentos aninhados, discutidos na Seção 3.2.9.

3.2.8.3 Replicação Seja $u_1, \ldots, u_n$ uma coleção de unidades. A operação de replicação permite que sejam criadas k cópias da sub-rede de comunicação induzida por estas unidades, obedecendo-se as seguintes restrições:

- Para cada unidade u_i, $1 \leq i \leq n$, são criadas k cópias, denominadas $u_i[1], \ldots, u_i[k]$;
- Se existe um canal, c, ligando uma porta p de certa unidade u_r a uma porta q de uma certa unidade u_s, então serão criados k canais, $c[1], \ldots, c[k]$, cada qual respectivamente ligando a porta p da unidade $u_r[i]$ à porta q da unidade $u_s[i]$, $1 \leq i \leq k$;
- Para todo canal c que liga uma porta p de uma unidade v a uma porta q de um unidade u_i, onde v não pertence ao conjunto de unidade replicadas, é criado um grupo de portas p na unidade v, contendo k réplicas da porta p, denominadas $p[1], \ldots, p[k]$, usando a função de ligação explicitamente especificada na cláusula **connections**. Além disso, são ainda criados k canais, identificados por $c[1], \ldots, c[k]$, cada qual ligando respectivamente a porta $p[i]$ da unidade v à porta q da unidade $u[i]$, $1 \leq i \leq k$.



replicate 3 b # y **connections** y<>: *g*

unify d # x → (), e # x → ()
to de # x → () **behaviour**: **repeat** x? **until** x
connections y<>: *h*3

Figura 3.14. Grupos Aninhados de Portas

O exemplo na Figura 3.13 ilustra o uso da operação de replicação. Neste, a sub-rede formada pelas *unidades_X* e *unidade_Y* é replicada em três cópias. Somente as portas conectadas às portas $l1$ e $l2$ da *unidade_X* e $l3$ da *unidade_Y* são replicadas. Atente-se para a configuração das funções de ligação nestes grupos de portas, usando a cláusula **connections**.

3.2.9 Agrupamentos Aninhados de Portas

O conceito de agrupamento aninhado de portas é motivado através do exemplo ilustrado na Figura 3.14. Neste, a unidade a possui um agrupamento de duas portas, identificado por x, cujas portas individuais, $x[1]$ e $x[2]$, estão conectadas às portas y e z das unidades b e c, respectivamente. Ao aplicarmos uma operação de replicação (ou fatoração) sobre a unidade b, é necessário replicar a porta $x[1]$. Isso implicaria na modificação do agrupamento de portas x da unidade a e a utilização de uma nova função de ligação, a qual seria uma composição da funções de ligação de x, f, com aquela declarada na cláusula **connections** na replicação, g. Entretanto, uma maneira mais simples e compatível de possibilitar isso é transformar a porta $x[1]$ em um agrupamento de portas aninhado a x. Como resultado, têm-se as portas $x[1][1]$, $x[1][2]$ e $x[1][3]$, as quais formam um agrupamento de portas identificado por $x[1]$. A função de ligação para o agrupamento $x[1]$ seria a função g, declarada na cláusula **connections**.

No segundo exemplo apresentado na Figura 3.14, as unidades d e e são unificadas. Na unidade resultante, os agrupamentos de portas identificados por x nas unidades originais são agrupados, formando um agrupamento aninhado cuja função de ligação, $h3$, é declarada na cláusula **connections**.

De uma forma geral, em uma replicação ou fatoração de uma unidade, a replicação de uma porta pertencente a um agrupamento gera um novo agrupamento, aninhado ao original, com sua própria função de ligação. Em uma unificação, um aninhamento de portas ocorre quando uma das portas em um conjunto de portas a serem unificadas,

como resultado de uma sobreposição de interfaces, é um agrupamento de portas.

Deve ser observada a compatibilidade de tipos das funções de ligação na formação de agrupamentos aninhados. Para isso, é suficiente obedecer-se as seguintes regras na formação destes:

i) Em uma fatoração ou replicação, o tipo de entrada e saída da função de ligação deve ser o mesmo do tipo das portas individuais (protocolos dos canais);

ii) Em uma unificação, o tipo de entrada e saída da função de ligação deve ser o mesmo do argumento, ou ponto de saída, do componente, associado ao grupo;

É importante reforçar que agrupamentos aninhados de portas não são suportados em declarações **unit**. Somente ocorrem como efeito colateral de operações sobre unidades.

3.2.10 Descrevendo Grandes Coleções de Entidades (Notação Indexada)

Em [220], Turner apresenta algumas considerações sobre ambientes de programação paralela explícita:

> "O potencial de desempenho deste tipo de arquitetura é enorme, mas como pode ser programada ? Uma idéia que pode ser desmentida de forma mais ou menos direta é a de que podemos tomar uma linguagem convencional e adicionar a esta facilidades para criação e coordenação explícita de processos. Isso pode ser possível quando o número de processos é pequeno, mas quando estamos nos referindo a milhares e milhares de processos independentes, isso não pode ser possível sob o controle consciente do programador."

Gelernter e Carriero [99] discordam de Turner, atribuindo sua falácia a suposição de que cada entidade em uma linguagem de coordenação deve ser criada e tratada individualmente. Na prática, porém, em programas paralelos de grande escala, a maioria dos processos são idênticos e tem a função de processar um pedaço de uma grande estrutura de dados. Turner não levou estes aspectos em consideração. Existem formas de manipular uma grande quantidade de processos sem referenciá-los individualmente. Citando Gelernter: "especificar explicitamente não significa especificar individualmente".

A linguagem de configuração # introduz em sua sintaxe a noção de *parâmetros estáticos* e *notação indexada* com o fim de permitir referenciar uma grande quantidade de entidades (unidades, portas, interfaces, etc.) de forma concisa.

Como vimos, parâmetros estáticos são declarados no cabeçalho de uma configuração #. Ao instanciar um componente a partir de uma configuração, o programador deve prover o valor do parâmetro. Parâmetros podem ser usados em qualquer expressão presente no código # que produza algum valor dos seguintes tipos: *caracteres*, *cadeias de caracteres*, *números inteiros*, *números em ponto flutuante*. Sua função é portanto a parametrização de uma configuração.

As declarações **index** permitem a declaração de índices, os quais podem assumir uma faixa de valores. Considere os exemplos:

index i,j,k **range** $[1..2 * N + 1]$

No exemplo acima, são declarados três índices, identificados por i, j e k, com variação entre 1 e $2 * N + 1$, onde N é um parâmetro estático.

Índices são úteis quando usados no contexto de *escopos de variação*. Escopos de variação são trechos de código delimitados pelos símbolos [/ e /]. São permitidos na declaração de coleções de entidades (ver sintaxe formal da linguagem # no Apêndice). O exemplo a seguir ilustra o uso de um escopo de variação na declaração de um índice:

index i **range** [1..3]
[/ **index** q[i] **range** $[1..i + 1]$ /]

o código # acima é equivalente ao seguinte código expandido:

index i **range** [1..3]
index q[1] **range** [1..2]
index q[2] **range** [1..3]
index q[3] **range** [1..4]

Dentro do escopo de variação o índice i varia de acordo com a variação declarada para este índice. Caso existam mais de um índice no contexto de um escopo de variação, serão consideradas todas as possíveis de combinações desses índices. O seguinte exemplo ilustra este fato:

index i **range** $[1..N]$
index j **range** $[1..M]$
interface IxExample ([/ in[i][j] /]::Int) → out::Int
behaviour: seq { [/ in[i][j]? /]; out! }

Sejam N e M, respectivamente iguais a 3 e 2, o código # acima é equivalente ao seguinte código expandido:

interface IxExample (in[1][1], in[1][2], in[2][1], in[2][2], in[3][1], in[3][2] :: Int) → out
behaviour: seq { in[1][1]? ; in[1][2]? ; in[2][1]? ; in[2][2]? ; in[3][1]? ; in[3][2]? ; out! }

No contexto de um escopo de variação, a variação de um índice pode ser limitada pelo valor de outro índice, como no exemplo que se segue para o índice j:

index i **range** $[1..3 * N]$
index j **range** $[1..i]$
[/ **unit** ix_example[i][j] # *AnInterface* in → out /]

Assumindo $N = 1$, o código # expandido equivalente é o seguinte:

index i **range** [1..3]
index j **range** $[1..i]$

unit ix_example[1][1] # *AnInterface* in → out
unit ix_example[2][1] # *AnInterface* in → out
unit ix_example[2][2] # *AnInterface* in → out
unit ix_example[3][1] # *AnInterface* in → out
unit ix_example[3][2] # *AnInterface* in → out
unit ix_example[3][3] # *AnInterface* in → out

O índice j varia sempre entre 1 e o valor corrente de i, de acordo com as regras de variação de escopo que serão detalhadas adiante.

Escopos de variação podem ser ainda aninhados, em qualquer nível de aninhamento. Considere o seguinte exemplo:

```
index i range [1..3]
index k range [1..2]
index j range [1..i]
[/ index q[i] range [1..i]
   [/ index p[k][j][i.] range [1..i] /] /]
```

O código # acima é equivalente ao seguinte código:

```
index i range [1..3]
index k range [1..2]
index j range [1..i]
index q[1] range [1..1]
index p[1][1][1] range [1..1]
index p[2][1][1] range [1..1]
index q[2] range [1..2]
index p[1][1][2] range [1..2]
index p[2][1][2] range [1..2]
index p[1][2][2] range [1..2]
index p[2][2][2] range [1..2]
index q[3] range [1..3]
index p[1][1][3] range [1..3]
index p[2][1][3] range [1..3]
index p[1][2][3] range [1..3]
index p[2][2][3] range [1..3]
index p[2][3][3] range [1..3]
index p[1][3][3] range [1..3]
```

O índice i aparece no contexto do escopo mais interno e no escopo mais externo. Entretanto, o escopo mais interno tem prioridade sobre o externo. O uso do '.' após o índice i no escopo mais interno, indica que o i referenciado corresponde aquele do nível de aninhamento imediatamente superior. A quantidade de símbolos '.' após um índice indica o aninhamento do índice ao qual está se referenciando.

3.2.10.1 Regra para Expansão de Escopos de Variação O processo de compilação inicia com a expansão dos escopos de variação. Em cada iteração do algoritmo, é expandido o nível de aninhamento mais externo, enquanto houver escopo a ser expandido. Seja o seguinte escopo de variação, com índices $i_1, i_2, \ldots, i_n$ em seu contexto:

$$[/ \ldots i_1 \ldots i_2 \ldots i_n \ldots /]$$

Considerando a existência de índices que dependem da variação de outros índices, define-se a função δ, a qual retorna os índices de cuja variação um índice depende. Os índices $i_1, i_2, \ldots, i_n$ devem ser enumerados da seguinte forma:

$$i_j \notin \delta[i_{j+1}]$$

Dessa forma, um índice não pode depender de algum índice que o sucede na enumeração. Para que isso seja possível, não podem existir índices mutuamente dependentesem uma configuração, restrição que pode ser verificado em tempo de compilação.

Seja ι a função que retorna um conjunto ordenado de valores de um índice recebido como argumento. Assim, podemos definir o conjunto ordenado Γ, o qual contém as associações de valores dos índices assumidos no escopo de variação, como $\iota[i_1] \times \iota[i_2] \times \ldots \times \iota[i_n]$. O escopo de variação é portanto expandido da seguinte forma:

$$\begin{gathered} \ldots i_1 \ldots i_2 \ldots i_n \ldots \\ \Downarrow \\ \ldots s_1 \ldots s_2 \ldots s_n \ldots, \forall(s_1, s_2, \ldots, s_n) \in \Gamma \\ \text{onde } s_k = \iota[i_k] \end{gathered}$$

3.2.11 Módulo de Disparo

Um módulo de disparo é usado para instanciar um programa #, definindo sua *unidade principal* (*main*) e associando a esta o componente de aplicação que define a funcionalidade do programa. Na Figura 3.15, é apresentado o módulo de disparo para uma aplicação de multiplicação de matrizes.

A *unidade principal* é sempre identificada por *main*. Um valor constante é passado como parâmetro ao componente *MatrixMultiplication*, indicando a dimensão da matriz. Caso o componente *MatrixMultiplication* possuísse pontos de entrada e saída, os valores requisitados por estes poderiam ser supridos com expressões Haskell, como mostrado no exemplo da Figura 3.16.

O componente *OneComponent* da aplicação *Demonstration* possui um parâmetro estático, do tipo *String*, para o qual é passado a constante `"Hello World !!!"`, e dois pontos de entrada, os quais recebem como argumento os valores produzidos pelas expressões `[1..]` e `fat 10`. Estes são passados para a função *main* da unidade cuja porta está conectada ao ponto de entrada, dentro da hierarquia de unidades da aplicação, sendo avaliadas de acordo com o mecanismo de avaliação de expressões de Haskell.

A função *finalize* é obrigatoriamente usada quando a unidade principal possui pontos de saída. Os argumentos passados a esta função são os valores produzidos nesses pontos de saída, na ordem em que são declarados, como ilustrado na Figura 3.16.

application MatrixMultiplication **with**

use MatrixMultiplication **in** `'matrix_multiplication.hcl'`

unit main as MatrixMultiplication<4>

Figura 3.15. Módulo de disparo para o programa de multiplicação de matrizes

```
application Demonstration with

{#
fat 0 = 1
fat n = n * fat (n-1)
#}

use OneComponente in 'one_component.hcl'

unit main as OneComponent<"Hello World!!!"> ([1..], fat 10) → (a::Int; b::Int)

{#
finalize :: Int → Int → IO()
finalize x y = print x >>
print y
#}
```

Figura 3.16. Uso de código Haskell em um módulo de disparo

```
library Basic_Interfaces where

interface ISource # in → () behaving as: in?
interface ITarget # () → out behaving as: out!
interface ISourceTarget # in → out behaving as: seq {out!;in?}
```

Figura 3.17. Exemplo de Biblioteca

3.2.12 Bibliotecas de Interfaces

Bibliotecas de de interfaces são usadas para armazenar declarações de interfaces que podem ser reusadas em várias configurações. São usadas como forma de melhorar a estruturação de código reusável.

O exemplo na Figura 3.17 mostra um exemplo de biblioteca, a qual declara três interfaces: *ISource*, *ITarget* e *ISourceTarget*. Estas são usadas na configuração dos componentes que descrevem os esqueletos de comunicação coletiva que serão introduzidos no Capítulo 4. Componentes podem importar estas interfaces usando a declaração **import**, como no exemplo seguinte:

```
import Basic_Interfaces
```

3.3 ESTRUTURANDO COMPONENTES SIMPLES (MÓDULOS FUNCIONAIS)

Enquanto componentes compostos implementam o meio de coordenação em programas #, descrevendo as entidades que compõem o *mundo dos processos*, componentes simples implementam seu meio de computação, expressando as funcionalidades inerentes às unidades executáveis do programa.

O modelo # foi projetado de forma que virtualmente componentes simples possam ser programados em qualquer linguagem sequencial (linguagem *host*), pertencentes aos

principais paradigmas de programação existentes. Embora esta funcionalidade multi-lingual não esteja atualmente disponível, é fácil descrever como isso pode ser feito. Para tal, é importante classificar os tipos de módulos funcionais que podem ocorrer em um programa #, com respeito a forma como a produção de dados através de seus pontos de retorno é induzida, o qual é influenciado pelo tipo linguagem *host* considerada em sua implementação:

- **Orientados ao Fluxo de Controle (*Assíncronos*)**: Incluem-se nessa classe módulos implementados em linguagens cujas computações são expressas sob o paradigma imperativo tradicional, como C, Fortran, Java, etc. Linguagens funcionais que empregam estratégia *eager* de avaliação também são representantes desta classe de linguagens, como ML [172]. Linguagens orientadas ao fluxo de controle constituem a maioria das linguagens que têm sido empregadas com sucesso em computação científica;

- **Orientados à Demanda**: Incluem-se as linguagens onde os dados são produzidos sob demanda. As principais representantes são linguagens funcionais não-estritas implementadas sob a estratégia *lazy* de avaliação [129, 225], como Haskell [121, 211] e Miranda [221]. Os pontos de entrada são lidos de acordo com a demanda por valores necessários ao cálculo do valor a ser retornado.

- **Orientados ao Fluxo de Entrada de Dados**: Incluem-se linguagens onde os dados são produzidos em função dos valores recebidos como entrada[2], como Id [79, 178];

Na Seção 6, discutiremos a implementação de processos associados a módulos funcionais pertencentes às três classes enumeradas acima.

3.4 A LINGUAGEM HASKELL$_{\#}$ E SUA EVOLUÇÃO

A linguagem Haskell$_{\#}$ [148, 147, 51] consiste em uma materialização para o modelo # de programação paralela. A linguagem funcional Haskell, de semântica não-estrita, é utilizada como linguagem *host*, na programação de componentes simples, também chamados módulos funcionais nesse contexto. Pode-se assim dizer que, em Haskell$_{\#}$, a linguagem de configuração # implementa o meio de *coordenação*, enquanto Haskell implementa o meio de *computação*. Uma vez que Haskell é uma linguagem que utiliza a estratégia *lazy* de avaliação, os módulos funcionais em Haskell$_{\#}$ são *orientados à demanda*.

O uso de Haskell em nível de computação possui duas motivações importantes:

i) Complementa o arcabouço de prova formal de propriedades de programas paralelos no modelo #. Linguagens funcionais puras, cuja semântica inspira-se no λ-calculus, possuem forte fundamentação matemática que favorece o uso de mecanismos formais para análise e prova de programas, ao contrário do que ocorre com linguagens imperativas, intrinsecamente relacionadas ao modelo arquitetural de computadores

[2] *Data-flow languages.*

```
module Tracking(main) where

import Track
import Tallies
import Mcp_types

main :: User_spec_info → [(Particle,Seed)] → IO ([[Event]],[Int])
main user_info particle_list =
    let
        event_lists = map f particle_list
    in
        return (event_lists, tally_bal event_lists)
    where
            f (particle@(_,_,_, e, _), sd) = (Create_source e):(track user_info particle [] sd)
```

Figura 3.18. Um Exemplo de Módulo Funcional Haskell$_{\#}$

proposto por Von-Neumann e que inspira a arquitetura dos computadores contemporâneos [15];

ii) O mecanismo de avaliação *lazy* permite a ortogonalidade sintática entre os meios de computação e coordenação. Nenhuma extensão ou biblioteca especial é necessária à linguagem Haskell para que módulos funcionais possam ser integrados no meio de coordenação descrito pela linguagem de configuração #. Este aspecto será explicado com maiores detalhes adiante.

Um exemplo de módulo funcional é ilustrado na Figura 3.18, extraído do programa MCP$_{\#}$, descrito no Capítulo 4. Um módulo funcional é portanto um módulo Haskell, o qual exporta uma função *main* que caracteriza a funcionalidade do módulo. Os argumentos e pontos de retorno do componente simples induzido por um módulo funcional correspondem respectivamente aos argumentos e elementos da tupla retornada, através da mônada IO, pela função *main*. O argumento *particles* e o valor retornado *event_lists* são listas consumidas e produzidas, respectivamente, sob demanda, segundo o mecanismo de avaliação *lazy* de expressões característico de Haskell. Podem portanto ser associadas à portas *stream*, de forma que seus valores possam ser recebidos ou enviados sob demanda. Esse recurso permite a intercalação entre computação em comunicação em programas Haskell$_{\#}$ sem que seja necessário extender a linguagem Haskell com primitivas de comunicação em nível de computação, garantindo a ortogonalidade entre os dois níveis de programação e fortelecendo a propriedade de hierarquia de processos no modelo #.

Em sua primeira versão [147], Haskell$_{\#}$ suportava primitivas de comunicação definidas sobre a mônada IO nos módulos funcionais. Devido a quebra de hierarquia de processos advinda do uso desta abordagem, tais primitivas foram abolidas. Na versão subseqüente [146, 47], adotou-se o mecanismo atualmente empregado de mapear os argumentos e elementos da tupla de retorno às portas de entrada e saída especificadas através da linguagem de configuração. Entretanto, adotou-se semântica estrita para a função *main* do módulo funcional, de forma que as portas de entrada deveriam ser lidas para que os

argumentos estivessem disponíveis antes de sua avaliação. Ao final da avaliação da função *main*, os valores retornados eram enviados através das portas de saída. A leitura das portas de entrada e saída seguia a ordem estrita e sequencial imposta pela sua ordem de declaração. Não havia portanto o suporte a intercalação entre operações de comunicação e computação, o que tornava a linguagem inapropriada para a caracterização de uma vasta classe de aplicações paralelas importantes.

Embora bastante restritivo, esse mecanismo permitiu a implementação de uma hierarquia de processos real, além de oferecer o primeiro mapeamento de programas Haskell$_{\#}$ para redes de Petri [149]. A versão mais recente de Haskell$_{\#}$, apresentadas neste trabalho, supera as restrições expressivas identificadas na versão anterior, alcançando a equivalência ao poder expressivo das redes de Petri. Reforça-se ainda a propriedade de hierarquia de processos com novos mecanismos de abstração. Mecanismos mais poderosos para análise de propriedades de programas com redes de Petri, incluindo a simulação de desempenho e análise de custos de comunicação são agora suportados e serão discutidos no Capítulo 5.

CAPÍTULO 4

TÉCNICAS E ARTEFATOS PARA O DESENVOLVIMENTO DE PROGRAMAS PARALELOS NO MODELO

Neste capítulo, são discutidas técnicas de programação paralela voltadas ao desenvolvimento de programas #, utilizando exemplos representativos. Sob um mesmo arcabouço, o modelo # é capaz de lidar com três importantes noções de modularidade, surgidas nas últimas três décadas: *programação composicional*[183, 71], *programação baseada em esqueletos*[63] e *programação orientada a aspectos*[139]. Essa característica atribui ao modelo # grande flexibilidade no desenvolvimento de componentes reutilizáveis e na sua composição com vistas ao processo de desenvolvimento de *software* paralelo, em diversos níveis e etapas deste processo. O modelo # mostra-se particularmente apropriado ao desenvolvimento de programas paralelos de larga escala, para implementação de classes de aplicações que têm se tornado comuns com o emergimento de arquiteturas de supercomputação distribuída baseadas na infra-estrutura de redes de longa distância, conceito conhecido como *grid computing*[89].

Será discutida ainda a derivação de programas # a partir de programas MPI, consequência da experiência adquirida ao portar-se programas contidos no pacote NPB (*NAS Parallel Benchmarks*), programados em MPI, para o modelo #. Este trabalho sugeriu ainda como programas SPMD (*Single Program Multiple Data*), uma importante técnica empregada largamente no desenvolvimento de aplicações científicas de alto desempenho, podem ser implementadas no modelo #.

4.1 PROGRAMAÇÃO # COMPOSICIONAL

A programação composicional abrange as técnicas tradicionais aplicadas para o desenvolvimento modular de programas, com suporte ao reuso de componentes, onde os módulos que compõem o sistema são enxergados como "caixas-preta" que abstraem a implementação de funcionalidades específicas usadas na composição de um *software*. Suas origens remontam a década de 70 [183, 184, 185, 71], quando técnicas de modularização de programas foram propostas em resposta à demanda por ferramentas que oferecessem um nível mais alto de abstração para o desenvolvimento de aplicações, as quais tornavam-se cada vez mais complexas do que aquilo que poderia ser gerenciado pela capacidade expressiva característica das linguagens de programação disponíveis àquela época. Os problemas advindos da inadequação das linguagens para o desenvolvimento de grandes projetos de *software* levou ao contexto que ficou conhecido como a *crise do software* [192].

É conhecido o fato de que o esforço exigido de um ser humano para solucionar um problema complexo pode ser significamente reduzido dividindo este problema em problemas mais simples (sub-problemas). A solução para o problema original pode então

ser obtida pela integração das soluções obtidas para os sub-problemas. Esta estratégia, conhecida como *dividir-para-conquistar*, despertou a atenção da comunidade interessada no desenvolvimento de técnicas que facilitassem a construção de aplicações de grande porte, reduzindo seus custos e tempo de desenvolvimento e tornando-as mais confiáveis e fáceis de manter. *Modularidade* tornou-se um conceito importante em *engenharia de programas*, sendo hoje considerado o seu atributo mais importante [192]. Embora o advento de linguagens que exploram aspectos modulares de programação remontem a um período anterior, citamos [71] como referência para as idéias seminais a respeito do que trata-se neste trabalho como uma disciplina composicional para o desenvolvimento de software, onde módulos, possivelmente reusáveis, são integrados por meio de MIL's (*Linguagens para interconexão de módulos*[1]). Estas idéias inspiraram o desenvolvimento de muitas linguagens e paradigmas de programação contemporâneos. Em [39] são estudados importantes aspectos modernos relacionados a estrutura de linguagens baseadas na composição hierárquica de componentes, em especial com respeito ao processo de compilação de programas.

Na programação composicional, um programa é construído a partir da composição de um conjunto de componentes funcionalmente independentes, possivelmente reutilizáveis. *Independência funcional* é garantida quando um módulo (componente) possui duas características:

- **Alto grau de coesão**: Exige que cada módulo possua uma única funcionalidade dentro do sistema;
- **Baixo grau de acoplamento**: Estabelece que a interface de cada módulo deve ser simples e bem-definida, de forma que esta seja a única forma de acessar a funcionalidade do módulo. Na medida do possível, a interação entre os módulos deve ser evitada.

4.1.1 Componentes "Caixa-Preta" em Programas

No modelo #, componentes compostos não-abstratos e componentes simples podem ser enxergados como componentes "caixa-preta", funcionalmente independentes, cuja funcionalidade pode ser acessada por meio dos argumentos e pontos de retorno que constituem suas respectivas interfaces.

Componentes não-abstratos (simples ou compostos) permitem o reuso de partes de programas respectivamente no nível de coordenação e computação. Portanto, enquanto componentes simples encapsulam funcionalidades implementadas por meio de computações sequenciais, componentes compostos não abstratos encapsulam funcionalidades implementadas por meio de computações paralelas. Uma vez que, no modelo #, não é feita nenhuma distinção sobre o caráter simples ou composto de um componente na sintaxe de associação entre unidades e componentes, componentes sequenciais podem ser substituídos por componentes paralelos equivalentes, e vice-versa, contanto que suas interfaces sejam compatíveis, segundo as restrições impostas pela interface da unidade associada ao componente.

[1] *Module Interconnection Languages.*

Outra interessante possibilidade advém da potencialidade do modelo # para exploração de hierarquias de paralelismo em *constelações*[2]. Atualmente, a despeito da popularização destas arquiteturas em problemas que tradicionalmente demandam pelo uso de supercomputadores dedicados (ver Capítulo 2), há carência por ferramentas de programação que explorem suas potencialidades [31].

Vale ressaltar que o mecanismo de programação composicional era o único suportado na versão anterior da linguagem Haskell$_{\#}$[47, 146]. Entretanto, somente eram suportados os atuais componentes simples, em nível de computação (*módulos funcionais*). O mecanismo de abstração provido pelos componentes não-abstratos, em nível de coordenação, o qual constitui dos mais importantes atributos do modelo #, ainda não era suportado.

Com o fim de exemplificar e esclarecer aspectos relevantes sobre o uso de componentes "caixa-preta" na construção de programas #, será introduzida uma implementação # para a aplicação CSM (*Climate System Model*) [38], a qual possui implementação em Fortran com suporte de paralelismo gerenciado por MPI.

4.1.2 A Aplicação CSM$_{\#}$

CSM (*Climate System Model*) é uma aplicação desenvolvida, originalmente em Fortran/MPI, no *National Center for Athmosferic Research* (NCAR), EUA, para simulação climática [38]. Sua implementação no modelo # é denominada CSM$_{\#}$. Na Figura 4.1.2, ilustramos a topologia da rede de processos de mais alto nível desta aplicação. As unidades **atm**, **ice**, **lnd** e **ocn** implementam modelos climáticos para, respectivamente *atmosfera*, *criosfera* (interação mar/gelo), *biosfera* (terra) e *hidrosfera* (oceano). A unidade acopladora, **cpl**, tem a finalidade de receber os fluxos calculados por cada uma das unidades climáticas e compô-los, enviando de volta às unidades climáticas os fluxos atualizados. Esta sequência é repetida em passos temporais. Assim como na implementação original, dois dos componentes climáticos são estáticos: criosfera (**ice**) e hidrosfera (**ocn**). Isto significa que as unidades **ice** e **ocn** lêem os fluxos, correspondente a dados obtidos por medições reais, a partir de arquivos armazenados em disco. As demais unidades, **atm** e **lnd** são dinâmicas, implementando solução para PDE's que descrevem o comportamento desses sistemas.

O código # que implementa o meio de coordenação da aplicação CSM$_{\#}$ é apresentado na Figura 4.2. A interface das unidades que implementam os modelos climáticos (**atm**, **lnd**, **ice** e **ocn**) constitui-se de duas portas de entrada (*iir* e *ir*) e duas portas de saída (*iis* e *is*). As portas *iir* e *iis* são usadas para troca das mensagens iniciais entre a unidade climática e o acoplador. As demais (*ir* e *is*) são usadas para as troca de informação sobre os fluxos calculados da unidade climática, em cada passo temporal, com o acoplador. A unidade acopladora (**cpl**) possui portas de entrada e saída associadas a cada porta das unidades climáticas. No programa CSM, a unidade climática que implementa o modelo climático do *oceano* sincroniza em intervalos de tempo mais longos que as demais unidades climáticas. Para implementar esse fato, *streams* aninhadas foram empregadas nas interfaces das unidades.

[2]Clusters de computadores multi-processados constituídos de 16 ou mais unidades de processamento [31].

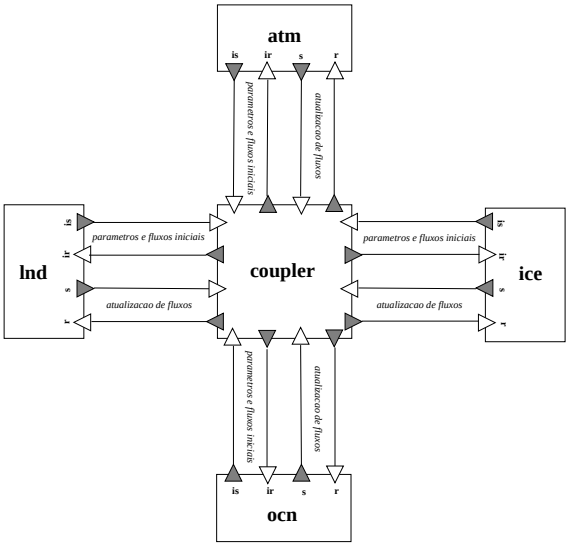


Figura 4.1. Topologia de CSM$_{\#}$

```
configuration CSM< ncpl_a, ncpl_o > with

use CPL, ATM, LND, LND, OCN, ICE

interface ICoupler # (iar,ar**) → (ias,as**)
                   # (ilr,lr**) → (ils,ls**)
                   # (iir,ir**) → (iis,is**)
                   # (ior,or*) → (ios,os*)
          behavior: seq { iar?; ias!; iir?; iis! ior?; ios?; ilr?; ils?;
                          ir?; or?; lr?; ar?;
                          repeat seq {os!; repeat {lr?; as!; is! ir?; ar?}
                                          until counter (ncpl_a div ncpl_o);
                                      or?}
                          until <ls & lr & as & ar & is & ir & os & or >

interface IModel # (ir, r*) → (is, s*)
          behavior: seq {is!; ir? repeat seq {s!; r?} until <r & s>}

unit cpl  # ICoupler as CPL
unit ocn  # IModel  as OCN
unit atm  # IModel  as ATM
unit lnd  # IModel  as LND
unit ice  # IModel  as ICE

connect atm.is to cpl.iar, synchronous
connect lnd.is  to cpl.ilr, synchronous
connect ice.is  to cpl.iir, synchronous
connect ocn.is  to cpl.ior, synchronous

connect cpl.ias to atm.ir, synchronous
connect cpl.ils to lnd.ir, synchronous
connect cpl.iis to ice.ir, synchronous
connect cpl.ios to ocn.ir, synchronous

connect atm.s  to cpl.ar, synchronous
connect lnd.s  to cpl.lr, synchronous
connect ice.s  to cpl.ir, synchronous
connect ocn.s  to cpl.or, synchronous

connect cpl.as to atm.r, synchronous
connect cpl.ls to lnd.r, synchronous
connect cpl.is to ice.r, synchronous
connect cpl.os to ocn.r, synchronous
```

Figura 4.2. Código de Configuração CSM$_{\#}$

Em $CSM_{\#}$, os modelos climáticos encontram-se implementados sob a forma de componentes simples, sendo portanto unidades de execução sequencial, como no CSM original. O código FORTRAN que implementa as computações numéricas e operações de I/O realizadas por CSM foi reusado, poupando-se o trabalho de uma implementação destes em Haskell. Para isso, foram empregados tipos abstratos de dados com a finalidade de encapsular as estruturas de dados (arrays) processadas pelo programa Fortran. Além disso, FFI (*Foreign Function Interface*)[81] é necessária para provimento de suporte à chamada de rotinas FORTRAN a partir do módulo funcional Haskell. A parte Haskell da implementação dos componentes pode ser enxergada como um invólucro responsável por aderir a funcionalidade computacional provida pelo código Fortran ao meio de # coordenação. O uso de Haskell como cola para aderir computações numéricas especificadas em linguagens Fortran ou C, mais apropriadas para este fim, é uma das possibilidades que têm sido estudadas com a finalidade de permitir o suporte multi-lingual ao ambiente #, favorecendo o reuso de códigos científicos pré-existentes, reconhecidamente eficientes e devidamente validados pela experiência advinda de seu uso. A aplicação $CSM_{\#}$ constitui um exemplo prático do emprego dessa técnica. Adiante, serão apresentadas outras técnicas que podem ser usadas com esse mesmo fim, em especial o uso de esqueletos topológicos.

Observe que o código de configuração de $CSM_{\#}$ não faz nenhuma suposição a cerca do caráter sequencial dos componentes *Atm*, *Lnd*, *Ocn* e *Ice*. Portanto, seria possível os implementar em paralelo, como componentes compostos não-abstratos, sem necessidade de alterar o código de configuração de $CSM_{\#}$. Este nível de abstração elevado é fruto da independência funcional característica de componentes "caixa-preta" (simples e compostos não-abstratos) no modelo #. A implementação paralela das unidades climáticas sugere o paralelismo explorado em diversos níveis hierárquicos. No nível acima, os componentes climáticos executam paralelamente, sincronizando com o módulo acoplador a cada passo de tempo. No nível abaixo, o paralelismo é explorado na estrutura interna a cada componente climático, os quais implementam soluções paralelas para os sistemas de equações que estes implementam para cálculo dos fluxos a cada passo temporal.

4.1.3 Explorando Hierarquias de Paralelismo em CSM

Muitos algoritmos paralelos eficientes para solução de sistemas de equações diferenciais não são apropriados para execução em ambientes distribuídos, como *clusters*, tendo em vista a grande quantidade de comunicação necessária durante a execução do programa paralelo. Entretanto, em geral estes podem ser eficientemente implementados sobre computadores multi-processados, onde os processadores comunicam-se por meio de um espaço de endereçamento comum. Tendo em vista tais limitações, têm se tornado comum o emprego de constelações em supercomputação científica, *clusters* de computadores multi-processados, para aplicação em classes de problemas que tradicionalmente só poderiam ser tratados de forma viável por supercomputadores. Entretanto as ferramentas de programação paralela atualmente disponíveis carecem de meios de suporte à hierarquia de paralelismo suportadas nestas arquiteturas[31].

Os componentes de $CSM_{\#}$ são exemplos de programas cuja implementação paralela

não é eficiente sobre arquiteturas distribuídas, mas que podem tornar-se atrativos quando implementados sobre arquiteturas multiprocessadas. Para uma implementação eficiente de $CSM_{\#}$, o compilador # poderia então ser informado explicitamente pelo programador sobre a geração de código que faça uso de uma biblioteca apropriada para gerenciamento de paralelismo em arquiteturas multi-processadas, como openMP[180], para versões paralelas das unidades que implementam os componentes climáticos. Dessa forma, o programa $CSM_{\#}$ poderia ser executado sobre um *cluster* formado por 5 computadores multiprocessados, os quais executariam paralelamente os componentes climáticos e o acoplador. A comunicação entre os computadores constituintes do *cluster* seria efetivada com uso de MPI, apropriado em sistemas paralelos distribuídos, enquanto openMP seria encarregado da comunicação entre os processos instanciados em cada nó multi-processado.

4.2 PROGRAMAÇÃO # BASEADA EM ESQUELETOS (TOPOLÓGICOS)

Na seção anterior, foi discutido como componentes não-abstratos (simples ou compostos) podem ser usados para modularização de programas #, através da técnica de programação composicional. Nesta seção, discute-se como componentes abstratos são capazes de oferecer suporte a formas alternativas de modularização, as quais não podem ser capturadas pela programação composicional convencional. Introduz-se assim a noção de esqueletos topológicos parciais e como estes podem ser usados para construção de programas #.

Esqueletos de algoritmos constituem uma importante técnica de programação introduzida por Murray Cole[64], na década de 80, com o intuito de capturar padrões comuns existentes em algoritmos paralelos. O uso de esqueletos permite que programas paralelos sejam portados de uma arquitetura para outra sem que haja perda significativa de eficiência, uma vez que para cada arquitetura particular o esqueleto (padrão de computação paralela) seria implementado da forma mais eficiente possível. Esqueletos oferecem ainda o suporte a um nível mais alto de abstração para a construção de programas paralelos a partir da composição de componentes reusáveis.

Muitas linguagens têm oferecido o suporte a esqueletos ou foram desenvolvidas com base neste conceito [42, 208, 43, 69, 67, 107]. Em geral, cada uma busca definir seu próprio conjunto de esqueletos reutilizáveis para a construção de programas. Essa diversidade motivou alguns trabalhos que visavam oferecer uma classificação para esqueletos, em função da experiência adquirida anteriormente com o seu uso [107].

No modelo #, emergem os denominados *esqueletos topológicos parciais* [48] como um importante estilo de programação modular, o qual acrescenta ao estilo composicional descrito na seção anterior a capacidade de descrição e reuso de padrões topològicos de organização de processos e seus respectivos comportamentos individuais, informações consideradas essenciais na configuração do paralelismo. O suporte à esqueletos no modelo # advém do suporte à *componentes abstratos*, os quais, como vimos, são componentes que possuem em sua constituição uma ou mais *unidades virtuais*. Isto permite a parametrização da computação realizada pelo componente. O conceito de esqueleto topológico parcial coincide portanto com a noção de componente abstrato. Esqueletos # podendo ser enxergados como uma generalização da noção de esqueleto tradicional-

mente encontrada na literatura, capturada pela sub-classe de esqueletos # denominada *esqueletos topológicos totais*, componentes abstratos onde todas as unidades são virtuais. Alguns exemplos clássicos de esqueletos totais são *pipe-line*, *farm*, *systolic mesh* e *tree*, definidos originalmente por Cole e implementados adiante como configurações #. Diz-se assim que em um esqueleto parcial a computação pode estar parcialmente definida.

O mecanismo de programação baseado em esqueletos pelo emprego de componentes abstratos eleva o nível de modularidade e reusabilidade suportado pelo um modelo #, ao permitir a descrição de padrões de topologia e interação entre processos as quais podem se sobrepôr, o que não é possível utilizando componentes "caixa-preta", descritos na seção anterior. Isso é possível uma vez que componentes abstratos podem compartilhar unidades virtuais, por meio da possibilidade de unificação destas em uma configuração (Seção 3.2.8.1). As unidades virtuais, em componentes abstratos, permitem ainda parametrizar e manipular informações sobre a estrutura interna de um componente dentro da configuração que o reusa, informação que encontra-se totalmente escondida em componentes não-abstratos. O uso de esqueletos topógicos pré-existentes podem portanto expôr às ferramentas de compilação a estrutura da rede de comunicação de processos de uma aplicação, permitindo que seja gerado código otimizado, com balanceamento de carga apropriado, de acordo com as informações providas pelos esqueletos usados para compôr a topologia da aplicação, assumindo-se as características inerentes de uma certa arquitetura paralela.

Estes e outros aspectos concernentes à programação baseada em esqueletos topológicos parciais serão discutidos a seguir através de exemplos. Será apresentada inicialmente uma biblioteca de esqueletos inspirados naqueles introduzido por Murray Cole, em seus trabalhos originais. Discutiremos então como esqueletos podem ser compostos através de aninhamento e sobreposição e usaremos estas técnicas para especificar a estrutura topológica de um algoritmo paralelo (sistólico) bastante conhecido para multiplicação de matrizes. Posteriormente, será introduzida uma biblioteca de esqueletos que abstraem as funcionalidades de comunicação coletiva fornecidas por MPI. O uso destes em uma aplicação permite informar explicitamente, porém em alto nível, ao compilador sobre o uso de suas primitivas de comunicação coletiva na geração do código MPI da aplicação. Usaremos aplicações extraídas do pacote de análise de desempenho de arquiteturas paralelas NPB (*NAS Parallel Benchmarks*) para ilustrar o uso de esqueletos MPI. A implementação dos esqueletos MPI no compilador # será discutida no capítulo 5.5 deste trabalho.

4.2.1 Uma Biblioteca de Esqueletos Elementares

A seguir, são introduzidas implementações # para alguns esqueletos tradicionalmente presentes em linguagens concorrentes que suportam a noção de esqueletos. Estes são inspirados naqueles inicialmente propostos por Cole e considerados representativos para aplicação em uma vasta classe de aplicações paralelas, segundo classificações propostas para esqueletos [107].

```
component Pipe_Line<N> in → out with

index i range [1,N-1]
index j range [1,N]

interface IPipe t u # in*::t → out*::u
          behavior: repeat seq {in?; out!} until <in & out>

[/unit pipe[j] # Pipe in → out;/]

connect * pipe[i]→out to pipe[i+1]←in/]

bind pipe[1]←in to in
bind pipe[N]→out to out
```

Figura 4.3. Esqueleto Pipe_Line

4.2.1.1 O Esqueleto Pipe_Line O componente abstrato Pipe_Line, cujo código # encontra-se ilustrado na Figura 4.3, interliga um conjunto de N unidades denominadas $pipe[i]$, $1 \leq i \leq$ N, segundo uma estrutura de *pipe-line*. Esta caracteriza-se pelo processamento de um fluxo de dados que parte de sua unidade mais à esquerda ($pipe[1]$) até sua unidade mais à direita ($pipe[N]$), passando por cada unidade intermediária, sequencialmente. A unidade $pipe[i]$ é chamada de *i-ésimo estágio* do *pipe-line*. Em cada estágio, o dado que chega a porta de entrada da unidade que o define sofre uma transformação, descrita pela funcionalidade da unidade, e é enviado pela sua porta de saída.

A interface de uma unidade $pipe[i]$, denominada *IPipe*, define uma única porta de entrada (*in*) e uma única porta de saída (*out*), ambas do tipo *stream*. Seu comportamento estabelece que *in* e *out* devem ser ativadas alternadamente até que as *streams* transmitidas por ambas as portas, as quais são sincronizadas (note o uso dos delimitadores $<$ e $>$ na condição de terminação da ocorrência do combinador **repeat**), cheguem ao seu final, fato caracterizado pela transmissão do marcador Eos 1, como descrito na Seção 3.2.1.

O componente Pipe_Line possui um ponto de entrada e um ponto de saída, os quais são respectivamente ligados (declaração **bind**) à porta de entrada da unidade $pipe[1]$ e à porta de saída da unidade $pipe[N]$.

O esqueleto *pipe-line* é comumente empregado para descrever o processamento paralelo de um fluxo sequencial de dados que pode ser processado em uma sequência de estágios, onde cada estágio modela uma transformação sobre um item da sequência de entrada e pode ser modelado por um processo. Durante a execução, cada processo envolvido pode realizar suas respectivas tarefas simulteamente aos demais, sobre o item de dados que encontra-se no estágio da computação o qual implementa.

4.2.1.2 O Esqueleto Farm Outro padrão bastante comun de interação entre processos em programas paralelos pode ser expresso pelo esqueleto Farm, cujo código # é apresentado na Figura 4.4. Neste, uma unidade distribuidora, denominada *distributor*, distribui um conjunto de dados, a serem processados, entre um conjunto de unidades trabalhadoras, denominadas $worker[i]$, $1 \leq i \leq N$. Uma função de ligação (`DivideWork`) aplicada sobre o agrupamento de portas de saída da unidade distribuidora (*out*), passada com argumento ao instanciar-se o componente, é usada para fornecer ao programador a flexibilidade de especificar como os dados devem ser distribuídos entre as unidades trabalhadoras.

```
component Farm<N, DivideWork, CombineWork> with

interface IDistributor t # () → out*::t behavior: out!
unit distributor # IDistributor

interface IWorker t u # in*::t → out*::u behavior: seq {in?; out!}
unit worker # IWorker

interface ICollector u # in*::u → () behavior: in?
unit collector # ICollector

connect distributor→out to worker←in
connect worker→out      to collector←in

replicate N worker connections in<>: DivideWork, out<>: CombineWork
```

Figura 4.4. Esqueleto FARM

```
component Master_Slave<N> () → result with

use Farm

interface IMaster # () → result # d like IDistributor # c like ICollector
          behavior: seq {d.out!; c.in?; result!}

unit farm as Farm

unify farm.distributor # d, farm.collector # c to master # IMaster

unit slave # IWorker
assign slave to farm.worker

bind master→result to result
```

Figura 4.5. Esqueleto MASTER_SLAVE

Após o processamento dos sub-conjuntos de dados providos pela unidade distribuidora, as unidades trabalhadoras enviam o resultado produzido a uma unidade coletora, denominada *collector*. Esta recebe os items de dados produzidos por cada unidade trabalhadora por meio de um agrupamento de portas de entrada (*in*). Uma função de ligação, passada como parâmetro ao componente (`CombineWork`), permite ao programador configurar como os dados recebidos a partir das unidades trabalhadoras devem ser combinados pela unidade coletora.

O padrão FARM é comum na implementação distribuída de soluções paralelas orientadas a dados (*paralelismo de dados*), onde grandes massas de dados devem ser processadas em paralelo por uma coleção de processos. Uma variação comumente empregada do esqueleto FARM é o esqueleto MASTER_SLAVE (Mestre/Escravo), onde as unidades distribuidora e coletora constituem a mesma unidade. Podemos então definir o esqueleto MASTER_SLAVE unificando as unidades *distributor* e *collector* do FARM em uma unidade mestre, identificada por *master*. As unidades trabalhadoras passam a chamar-se escravas, identificadas por $slave[i], 1 \leq i \leq N$. Adicionalmente, é criada uma porta de saída por onde a unidade mestre envia o resultado da computação realizada, de forma a que possa ser usado por outro processo, ou ignorado caso não se faça útil na aplicação. O código # do componente MASTER_SLAVE é apresentado na Figura 4.5.

4.2.1.3 O Esqueleto Grid Algoritmos sistólicos em grafos e malhas surgem na implementação de um grande variedade de aplicações paralelas. Na Figura 4.6, apresentamos o código # para o esqueleto GRID, o qual implementa uma topologia sistólica particu-

```
component GRID<N> # ([/ l[i] /], [/ u[i] /]) → ([/ r[i] /], [/ d[i] /]) with

use PIPELINE

index i, j range [1,N]

interface IGrid # row like Pipe # col like Pipe
                behavior: repeat seq {par {row.in?,col.in?}; par {row.out!;col.out!}}
                                until < row.in & row.out & col.in & col.out >

[/unit grid_cols[i] # in → out as PIPE_LINE<N>in → in /]
[/unit grid_rows[i] # in → out as PIPE_LINE<N>out → out /]

[/ bind grid_cols[i]→out to u[i] /]
[/ bind grid_cols[i]→out to d[i] /]
[/ bind grid_rows[i]→out to l[i] /]
[/ bind grid_rows[i]→out to r[i] /]

[/unify grid_cols[i].pipe[j] # row,
        grid_rows[j].pipe[i] # col to cell[i][j] # IGrid/]
```

Figura 4.6. Esqueleto GRID

lar, característica de uma vasta classe de programas que empregam padrões sistólicos de interação entre processos, onde estes estão organizados segundo um *grid* quadrado ($N \times N$).

O esqueleto GRID pode ser visto como uma generalização bidimensional do esqueleto PIPE_LINE. Com base nesta constatação, o esqueleto GRID apresenta-se aqui implementado pela sobreposição de um conjunto de $2 \times N$ unidades associadas a componentes PIPE_LINE. Os N primeiros estabelecem a relação entre os processos dispostos nas N linhas do grid, enquanto os N últimos caracterizam a disposição dos processos dispostos em suas N colunas.

No *grid* induzido pela configuração do componente GRID, cada unidade é identificada por $cell[i][j]$, onde i indica o número da linha e j indica o número da coluna. O comportamento sistólico de cada unidade *cell* pressupõe que estas realizem a seguinte sequência de operações a cada passo de sincronização: enviar a informação que guarda, através das portas conectadas a suas unidades adjacentes à direita ($row.out$) e abaixo ($col.out$); receber as informações guardadas pelas unidades adjacentes conectadas às portas à esquerda ($row.in$) e acima ($col.in$); e realizar uma computação. Ao final da computação, cada unidade *cell* deve verificar uma condição de terminação, a qual assume-se que torne-se verdadeira para todas as unidades simultaneamente. Este padrão de interação é capturado pela interface *IGrid*, a qual instancia as unidades $cell[i, j]$, caracterizando o comportamento sistólico das unidades instanciadas a partir dessa interface.

O componente GRID possui $2*N$ pontos de entrada e $2*N$ pontos de saída, associados respectivamente às portas de entrada dos processos localizados nas bordas acima e à esquerda e aos processos localizados nas bordas à direita e abaixo.

4.2.1.4 O Esqueleto Torus Uma interessante adaptação do esqueleto GRID apresentado anteriormente induz uma estrutra conhecida como *torus* (*toroidal grid*). Esta estrutura é definida a partir de um esqueleto *mesh*, com a conexão das portas correspondentes nos processos localizados nas bordas do grid, formando uma estrutura circular. O código para um esqueleto que implementa um *torus* é apresentado na Figura 4.7. Uma estrutura toroidal pode ser empregada, por exemplo, na implementação de uma multi-

```
component Torus<N> with

index i range [1..N]

use Grid

interface ITorus # IGrid

unit grid # ([/ l[i] /], [/ u[i] /]) → ([/ r[i] /], [/ d[i] /])
          as Grid ([/ l[i] /], [/ u[i] /]) → ([/ r[i] /], [/ d[i] /])

[/ connect grid→r[i] to grid→l[i] /]
[/ connect grid→d[i] to grid→u[i] /]
```

Figura 4.7. Esqueleto Torus (Grid Circular)

```
component Hypercube <N, SplitData, CombineData> with

index i range [1,N]
index j range [1,log N]

interface IHypercube in::t → out::t
          behavior: repeat seq {out!; in?} until counter (log N)

[/ unit cell[i] # IHypercube group in<log N>:SplitData, out<log N>:CombineData /]

[/ connect cell[i].out[j] to cell[i xor 2^j].in[j] /]
```

Figura 4.8. Esqueleto Hypercube

plicação de matrizes, como exemplificaremos adiante.

4.2.1.5 O Esqueleto Hypercube O esqueleto Hypercube, cujo código é apresentado na Figura 4.8 implementa um dos mais úteis padrões de interação de processos em programação paralela, usado extensivamente em algoritmos SPMD que requerem comunicação eficiente do tipo *todos-para-todos* entre processos. Alguns exemplos são certos algoritmos empregados em transposição de matrizes, redução de vetores, disseminação (*broadcast*) de dados, etc[88].

Devido ao arranjo topológico dos processos e canais formando uma estrutura em hipercubo, o número de mensagens necessárias para que todos os processos sejam informados sobre o conteúdo do estado local do outro é $2 * N * (log\ N)$, onde N é o número de processos. Isso pode ser facilmente observado analisando o comportamento descrito para a interface *IHypercube*, onde a interação entre os processos é realizada em $log\ N$ passos. A cada passo, uma unidade de Hypercube comunica o seu estado local para seus vizinhos. Em seguida, recebe de seus vizinhos a informação a respeito de seus respectivos estados locais. Ao definir-se um aglomerado a partir do esqueleto Hypercube, funções de ligação deve ser passadas aos argumentos `SplitData` e `CombineData` deste componente, configurando-se como, entre os processos adjacentes no hipercubo, os dados são distribuídos no envio e agrupados no recebimento, respectivamente.

Para um programa # que utiliza o esqueleto Hypercube ao descrever sua topologia de processos, o compilador # poderia ser capaz de aproveitar-se desta informação para geração de código eficiente e alocação otimizada de processos em uma arquitetura onde os nós estivesses organizados em hipercubo. Embora este interessante recurso não se encontre implementado atualmente pelo compilador #, não há restrição para tal, sendo um dos

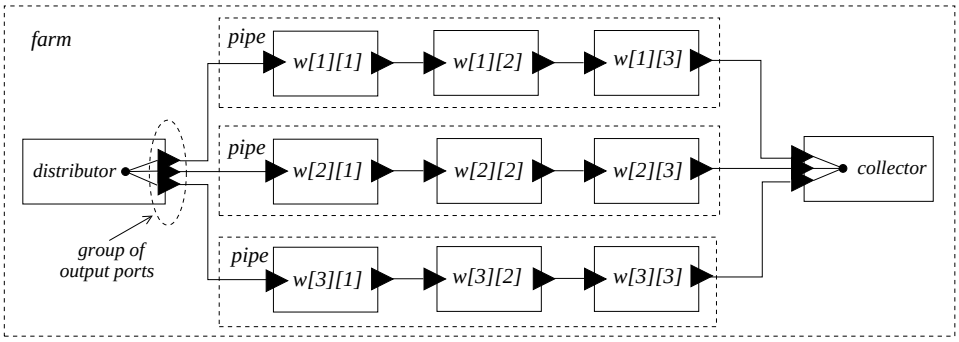


Figura 4.9. Aninhando um *Pipe-line* em um *Farm*

trabalhos futuros que deverão ser originados deste trabalho. A mesma observação é válida para os demais esqueletos, assumindo-se o fato realista de que muitas arquiteturas de processamento de alto desempenho importantes assumem certas topologias características.

4.2.2 Compondo Esqueletos: Aninhamento e Sobreposição

O mecanismo de suporte à esqueletos suportado pelo modelo # permite a definição de novos esqueletos a partir da composição de esqueletos pré-existentes. Este recurso foi empregado para configuração da maioria dos esqueletos definidos na seção anterior, com exceção de FARM e PIPE_LINE, os quais são primitivos. Para tal, foram usadas as técnicas básicas de *aninhamento* e a *sobreposição*, as quais são possíveis devido a existência das operações de *unificação* e *fatoração* sobre unidades, discutidas no Capítulo 2.

4.2.2.1 Aninhamento de Esqueletos O *aninhamento de esqueletos* é um mecanismo bastante expressivo para a descrição de topologias de rede complexas. Entretanto, poucas são as linguagens baseadas em esqueletos que exploram esta funcionalidade. TPascal [43] é um dos poucos exemplos de linguagem que provêem o suporte ao aninhamento de esqueletos. No modelo #, unidades instanciadas a partir de esqueletos podem substituir outras unidades, virtuais, dentro da configuração, através de nomeação (declaração **assign**). Utilizando este recurso, é fácil imaginar, por exemplo, um componente *farm* cujas unidades trabalhadoras (*workers*) são *pipe-lines*, como ilustrado na Figura 4.9.

4.2.2.2 Sobreposição de Esqueletos No modelo #, a sobreposição de esqueletos é possível devido a possibilidade de unificar-se unidades virtuais pertencentes a esqueletos distintos. Um exemplo simples é um *farm* onde as unidades trabalhadoras estão conectadas por meio de um *pipe-line*, ilustrado no diagrama da Figura 4.10). Para isso, a i-ésima unidade trabalhadora (*worker[i]*) do *farm* é unificada a i-ésima unidade estágio do *pipe-line*. Observe-se a sobreposição das portas de entrada da primeira unidade trabalhadora do *farm*, $worker[1]$, e da primeira unidade estágio do *pipe-line* $stage[1]$, as quais quando unificadas formam a unidade $pipeworker[i]$. O mesmo vale para as portas de saída pertencentes a última unidade trabalhadora do *farm*, $worker[n]$, e da última

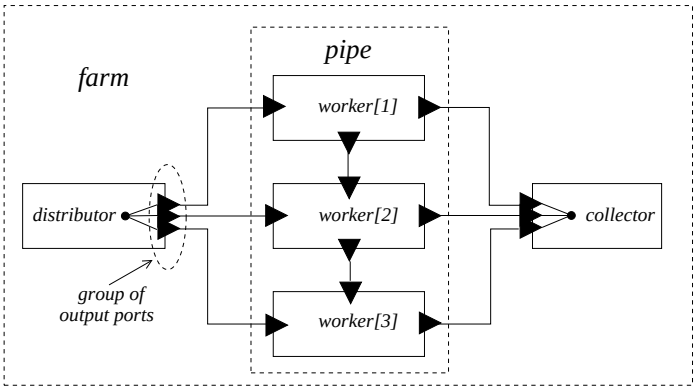


Figura 4.10. Soprepondo um *Farm* e um *Pipe-line*

unidade estágio do *pipe-line*, $pipe[n]$. No exemplo da Figura 4.10, n é igual a 3.

Nos exemplos de esqueletos básicos apresentados na seção anterior, a sobreposição de esqueletos PIPE_LINE foi empregada para a especificação do esqueleto MESH.

4.2.3 Exemplo: Multiplicação de Matrizes

Nesta seção, é apresentada a implementação de um algoritmo de multiplicação de matrizes (quadradas) utilizando um algoritmo sistólico onde os processos devem estar organizados em uma estrutura *toroidal*. Sejam $A_{N\times N}$ e $B_{N\times N}$ matrizes de entrada. O algoritmo calcula a matriz $C_{N\times N}$, tal que $C = A \times B$, em N iterações. Em uma iteração k, $1 \leq k \leq N$, cada unidade, denominada $matmult[i][j]$, onde i e j correspondem a linha e coluna que definem a localização do processo no *grid*, armazena os elementos $A(i,k)$ e $B(k,j)$ das matrizes de entrada. Estes são somados e o resultado é acumulado. Ao final dos N passos, cada processo $matmult[i][j]$ armazena o valor de $C(i,j)$. As matrizes A e B são distribuídas entre os processos $matmult$ pelas unidades mA e mB, segundo um arranjo inicial pré-determinado, especificado pela função de ligação *divide_matrix*. O resultado final calculado é enviado por cada processo ao processo mC.

Vale ressaltar que a suposição de que cada processo computa um elemento da matriz em uma multiplicação de matrizes não é uma suposição realista. Em problemas reais de computação científica, a ordem das matrizes é suficientemente grande para que a granularidade de paralelismo induzida pelo algoritmo acima torne inviável seu uso em arquiteturas convencionais. Além da não disponibilidade de uma quantidade suficiente de processadores, contribui para este fato o baixo desempenho causado pela excessiva sobrecarga de comunicação e sincronização. Técnicas de *aglomeração* podem ser usados para aumentar a granularidade da solução em uma implementação real. Outros algoritmos paralelos de multiplicação de matrizes podem ainda ser empregados. O exemplo apresentado é útil para os propósitos desse trabalho, por demonstrar como esqueletos podem ser compostos por sobreposição para constituir a topologia da rede de processos de uma aplicação, como discutido a seguir.

Podemos implementar o algoritmo de multiplicação de matrizes acima descrito pela

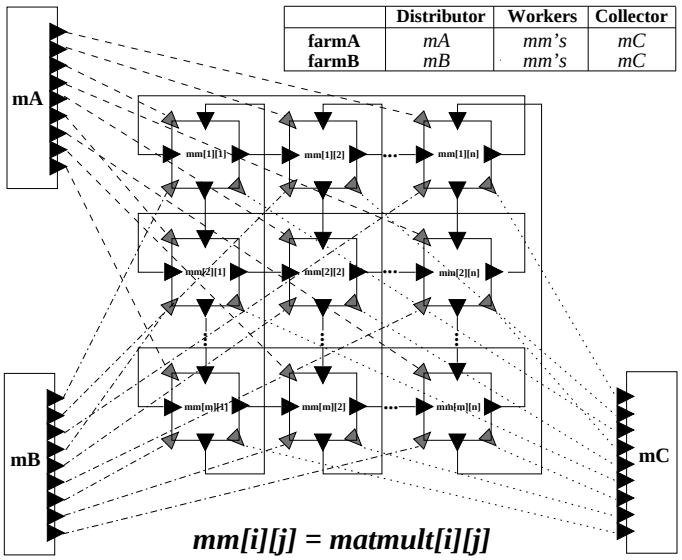


Figura 4.11. Topologia de um Programa # para Multiplicação de Matrizes

```
component MatrixMultiplication<N> with

index i, j range [1,N]

use MatrixMultiplication.MatMult
use MatrixMultiplication.ReadMatrix
use MatrixMultiplication.ShowMatrix

use Skeletons.Common.Farm
use Skeletons.Common.Torus

interface IMatMul
          tr like ITorus (l,t) → (r,b)
          fA like IWorker aij → cij
          fB like IWorker bij → cij
          behavior: seq { par {aij?; bij?};
                          repeat seq {l?; t?; r!; b!} until counter N
                          cij! }

unit mA # IDistributor as ReadMatrix N → out
unit mB # IDistributor as ReadMatrix N → out
unit mC # ICollector as ShowMatrix in → ()
unit farmA as Farm<N*N, divide_matrix, combine_matrix>
unit farmB as Farm<N*N, divide_matrix, combine_matrix>
unit mmgrid as Torus<N>

[/ unify farmA.worker[(i+1)*N + j] # fA,
         farmB.worker[(i+1)*N + j] # fB,
         mmgrid.meshcell[i][j]     # tr
         to matmult[i][j] # IMatMul as MatMult (N,a,b,l,u) → (r,d,c) /]

unify mmfarmA.collector # in → (),
      mmfarmB.collector # in → () to showmatrix # ICollector

assign mA to mmfarmA.distributor
assign mB to mmfarmB.distributor
assign mC to showmatrix
```

Figura 4.12. Multiplicação de Matrizes em HCL

sobreposição de esqueletos FARM e TORUS. O diagrama apresentado na Figura 4.11 ilustra tal arranjo. O esqueleto TORUS é usado para especificar a topologia em que encontram-se organizados os processos *matmult*, enquanto o esqueleto FARM especifica a relação destes com os processos *mA*, *mB* e *mC*. Para isso, são declaradas duas unidades instanciadas a partir do componente virtual FARM: $farmA$ e $farmB$. O primeiro emprega como distribuidor o processo *mA*, como trabalhadores os processos *matmult* e como coletor o processo *mC*. O segundo distingüe-se do primeiro por empregar o processo *mB* como distribuidor. As duas unidades FARM, portanto, sobrepõem-se, possuindo em comum os processos trabalhadores e coletor. Para isso é empregada a operação de unificação, que unifica os trabalhadores e coletores correspondentes das duas unidades. Os processos *matmult* e *mC* são então nomeados para substituir as unidades resultantes das unificações, respectivamente.

O código que implementa o módulo funcional $MatMult$ foi apresentado como exemplo no Capítulo 3, na Figura 3.18.

4.2.4 Esqueletos MPI

Nesta seção, é introduzida uma biblioteca de esqueletos capazes de abstrair o uso do conjunto de primitivas de comunicação coletiva suportadas por MPI[73], biblioteca de suporte à programação paralela distribuída por passagem de mensagens sobre a qual a linguagem # tem sido implementada. Este é um importante exemplo de como esqueletos podem ser usados para prover ao compilador informações topológicas de alto nível que podem ser usadas na geração de código mais eficiente. O uso destes esqueletos será demonstrado nos exemplos que compõem a próxima seção. Sua implementação e análise do ganho de desempenho devido ao seu uso serão assuntos a serem discutidos no Capítulo 6.

Os códigos # que implementam os esqueletos MPI encontram-se apresentados no Apêndice A. Em sua implementação, foram identificados três padrões topológicos comuns, induzidos pela semântica das primitivas de comunicação coletiva de MPI: *um-para-todos*, *todos-para-um*, *todos-para-todos*, os quais são implementados como configurações # pelos esqueletos PRIM_ONETOALL, PRIM_ALLTOONE e PRIM_ALLTOALL, respectivamente. As topologias desses componentes abstratos encontram-se ilustradas na Figura 4.13.

Os esqueletos PRIM_ONETOALL, PRIM_ALLTOONE e PRIM_ALLTOALL são empregados na implementação dos esqueletos MPI propriamente ditos. O esqueletos BCAST, SCATTER, SCATTERV, REDUCE_SCATTER e SCAN obedecem a topologia definida pelo esqueleto PRIM_ONETOALL. A diferença fundamental entre o esqueleto BCAST e os esqueletos SCATTER e SCATTERV é que, no primeiro, a função de ligação é pré-definida (*broadcast*), enquanto nos últimos esta é configurável pelo programador, ao instanciar-se o esqueleto. Observando-se as configurações de SCATTER e SCATTERV pode-se verificar que estas são iguais. Embora, em nível da programação #, tratem-se do mesmo esqueleto, o compilador pode fazer suposições diferentes em função de qual dos dois está sendo utilizado em uma aplicação, sendo capaz de induzir chamadas às primitivas de comunicação coletiva apropriadas de MPI para cada esqueleto. Mais especificamente, o uso

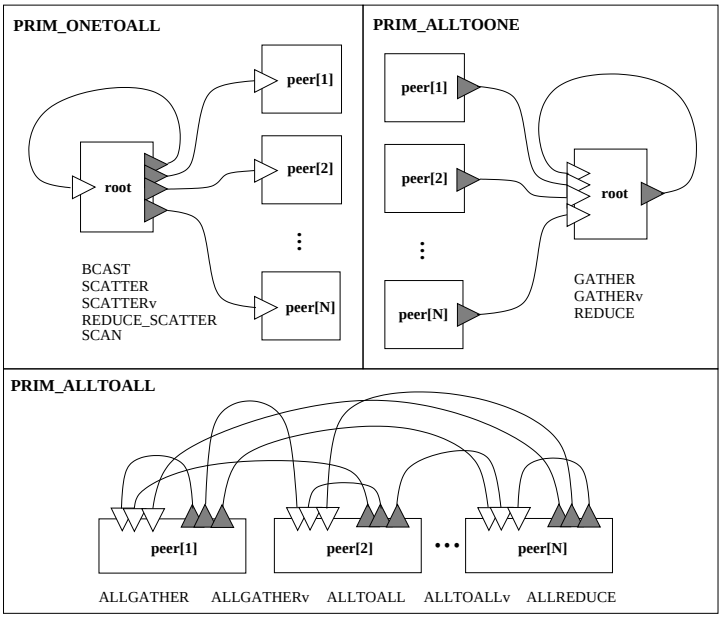


Figura 4.13. Topologias dos Esqueletos MPI Primitivos

de SCATTER força a suposição de que os dados que serão enviados para cada processo, pelo processo *root*, possuem o mesmo tamanho, em *bytes*, o que não pode ser assumido caso esteja-se usando SCATTERV, mais flexível. O uso deste último resulta em uma sobrecarga ligeiramente maior, por lidar com *buffers* de tamanhos diferentes para cada unidade que compõe o componente. Em REDUCE_SCATTER, a mesma suposição sobre o tamanho fixo do *buffer* enviado a cada processo é assumida, entretanto, uma operação (`OPERATION`) é aplicada sobre estes, elemento-a-elemento, antes do resultado ser enviado a todos os processos. O compilador assume que o *buffer* possui elementos do tipo de dado informado (`DATATYPE`), caso contrário o resultado é imprevisível, constituindo um erro de responsabilidade do programador. O esqueleto SCAN realiza uma redução prefixada dos dados que se encontram distribuídos entre os processos, utilizando a operação e assumindo-se o tipo de dados especificados.

Os esqueletos que assumem a topologia PRIM_ALLTOONE são GATHER, GATHERV e REDUCE. Assim como ocorre com SCATTER e SCATTERV, o código que define os dois primeiros esqueletos são iguais. Analogamente, o uso de GATHER e GATHERV será sensível somente a um compilador # que assuma geração de código específico para instâncias de esqueletos MPI. O primeiro esqueleto, GATHER, supõe que os *buffers* de dados recebidos pelo processo *root* de cada unidade do grupo possuem tamanho fixo, suposição que não é válida para GATHERV. O esqueleto REDUCE comporta-se como GATHER, entretanto, no processo *root*, uma operação pré-definda (`OPERATION`) é aplicada elemento-a-elemento sobre os *buffers* recebidos das demais unidades que constituem o componente. Para isso, é necessário explicitar o tipo de elemento do *buffer*, através do parâmetro `DATATYPE`, associado a um tipo de dados MPI.

Por fim, os esqueletos ALLGATHER, ALLGATHERV, ALLTOALL, ALLTOALLV, ALLRE-

DUCE são definidos segundo a topologia descrita pelo esqueleto primitivo PRIM_ALLTOALL. ALLGATHER e ALLTOALL diferenciam-se pelo primeiro pré-definir a função de ligação (*broadcast*) usada nos agrupamentos de portas de sáida presentes nas interfaces das unidades do componente. O esqueleto ALLTOALL é mais flexível, permitindo ao programador definir outras formas de distribuir o valor produzido pelo ponto de saída associado ao agrupamento entre as portas que o constitui. ALLGATHERV e ALLTOALLV são os análogos de ALLGATHER e ALLTOALL, possuindo o mesmo código # destes, respectivamente. A utilização dos últimos permite ao compilador assumir que o *buffers* recebidos e enviados entre as unidades do que constituem o componente tem tamanho fixo. É possível afirmar que ALLREDUCE está para ALLGATHER assim como REDUCE está para GATHER. Uma operação (`OPERATION`) elemento-a-elemento é aplicada, em cada unidade, sobre os *buffers* de dados recebidos dos demais processos, assumindo-se que estes são constituídos por sequência de elementos do tipo `DATATYPE`, correspondente a um tipo MPI pré definido.

Classes de interfaces são empregadas na especificação dos esqueletos MPI que assumem a existência de uma unidade raiz (*root*). Ao usar-se um esqueleto MPI deste tipo, pode-se declarar a interface da unidade virtual que comporá o esqueleto como o tipo mais geral, deixando a cargo do compilador inferir a interface especializada de acordo com a unidade a qual esta é associada. Exemplos deste tipo de emprego para classes de tipo, em esqueletos MPI, são ilustrados adiante, na construção das aplicações simuladas LU, SP e BT de NPB.

No Capítulo 6, será mostrado como os esqueltos MPI podem ser implementados no compilador #, de forma a permitir a geração de código MPI mais apropriado e eficiente.

4.2.5 Implementando NAS Parallel Benchmarks (NPB) no Modelo

NPB[16, 27] é uma coleção de 8 aplicações, inicialmente sugeridas e posteriormente disponibilizadas pelo centro de pesquisa da NASA(*National Agency for Space Administration*), agência espacial norte-americana, localizado em Ames, em meados de 1994, para avaliar o desempenho de sistemas de supercomputação para execução eficiente do programa NAS (*Numerical Aerodynamic Simulation*), um grande esforço que era conduzido àquela época para avançar o estado-da-arte da aerodinâmica computacional, aplicação de grande interesse da NASA .

Inicialmente, os problemas que compõem o NPB foram especificados em alto nível, sem preocupações com implementação, tendo em vistas as diferentes características de cada arquitetura paralela. Dessa forma, os provedores de sistemas de supercomputação poderiam implementá-los da forma mais eficiente e apropriada, assumindo-se as características intrínsecas de suas arquiteturas. Foram definidos classes padrão para caracterizar tamanhos de problemas a serem processados para efeito de medição e comparação. Inicialmente, duas destas classes foram defindas para cada problema: A e B. Apesar dessa flexibilidade, uma série de regras foram definidas para que fossem válidas as medições de desempenho conduzidas pelos programadores. Por exemplo, os códigos deveriam ser desenvolvidos em C ou Fortran, delimitando-se apenas os tipos de extensões e bibliotecas permitidas. Mais tarde, algumas dessas regras foram relaxadas e surgiram novas classes de problemas.

Com a disseminação dos *clusters* e consolidação das arquiteturas de memória distribuída, foi apresentado o NPB 2.0[27]. Nesta versão, ao contrário da primeira, foram apresentados implementações padrões para o NPB original, utilizando-se Fortran 77 provido da biblioteca MPI para configuração e gerenciamento do paralelismo, além de algumas pequenas modificações nas definições originais dos problemas. Mais recentemente, a versão 3.0 de NPB apresentou versões Java^{TM} e HPF para alguns dos *benchmarks*. O natural aumento do desempenho das arquiteturas de alto desempenho ao longo do tempo motivou ainda o surgimento de novas classes de problemas (C, D e W).

Com a finalidade de avaliar o desempenho de programas implementados dentro do modelo #, bem como algumas de suas funcionalidades, como o uso de esqueletos MPI descritos anteriormente, foi implementado um subconjunto de NPB, constituído por 3 bechmarks (EP, IS e CG) e 3 aplicações simuladas (LU, SP e BT). Os aspectos relativos à análise de desempenho serão apresentados no Capítulo 6, inclusive incluindo figuras de desempenho que analisam o ganho de desempenho advindo do uso de esqueletos MPI. Neste capítulo, estamos interessados em demonstrar os aspectos relativos à engenharia de programas paralelos de alto desempenho no modelo #, em especial o uso de esqueletos MPI na composição da topologia de aplicações reais. As seções seguintes descrevem as implementação realizadas, cujo código completo encontra-se listado no apêndice B.

4.2.5.1 O Kernel EP ("*Embaraçosamente Paralelo*"): O kernel EP gera pares de desvios randômicos gaussianos de acordo com um esquema específico [27], tabulando os dados em um formato pré-determinado. Foi desenvolvido para estimar o limite alcançável para o desempenho em ponto flutuante em uma arquitetura paralela.

A implementação do kernel EP no modelo # é a mais simples dentre as apresentadas no apêndice B. Sua rede de processos é composta por n unidades, nomeadas $ep\_unit[i]$, $1 \leq i \leq n$. Estas são instanciadas a partir da mesma interface, nomeada *IEP*, e associadas ao mesmo componente, como é característico para programas # do tipo SPMD. Utilizando o operador de composição de interfaces #, a interface *IEP* é especificada a partir de instâncias da interface *IAllReduce*, identificadas por sx, sy e q. Uma interface *IAllReduce* (Seção A.1) constitui-se de uma porta de entrada e uma porta de saída, respectivamente nomeadas *in* e *out*. Portanto, a interface *IEP* possui três portas de entrada ($sx.in$, $sy.in$, $q.in$) e três portas de saída ($sx.out$, $sy.out$, $q.out$). O módulo funcional EP descreve a computação realizada pelas unidades *ep_unit*. Possui quatro argumentos e três pontos de retorno. O primeiro argumento é reservado ao recebimento explícito dos parâmetros que configuram a instância de problema executada por EP. Os demais são mapeados nas portas da interface da unidade.

A topologia da rede de processos de EP é definida pela sobreposição de três instâncias do esqueleto AllReduce, cada qual envolvendo três sub-conjuntos disjuntos de portas de cada unidade, respectivamente prefixadas por sx, sy e q. Isso reflete a ocorrência de três chamadas à primitiva *MPI_AllReduce* no código original de EP. Os aglomerados sx_comm, sy_comm e q_comm são instâncias de esqueletos. Para isso, não possuem interface, mas um componente abstrato (esqueleto *AllReduce*) é associado a cada. As unidade virtuais que compõem os aglomerados sx_comm, sy_comm e q_comm são unificados de forma a definir a topologia do programa EP segundo o modelo #.

Além de prover alto grau de modularidade, esqueletos MPI provêem informações topológicas de alto nível que permitem ao compilador fazer uso mais eficiente das primitivas de MPI de comunicação coletiva, otimizando o desempenho. A mesma abordagem será usada na implementação dos outros *benchmarks*, com algums detalhes adicionais inerentes a cada um.

4.2.5.2 O kernel IS (*Ordenação de Inteiros*): O kernel IS realiza a ordenação paralela de uma sequência de N chaves, usando o algoritmo tradicionalmente conhecido como *bucket sort*. As chaves são geradas por um algoritmo sequencial [16] segundo uma distribuição uniforme.

A rede de processos funcionais de IS é composta pelas unidades $is\_unit[i]$, $1 \leq i \leq n$. A interface que as instancia é nomeada *IIS*, descrevendo o comportamento de três portas de entrada e três portas de saída, a partir da composição de instâncias das interfaces *IAllReduce* (*bs*), *IAllToAll* (*kb*) e *IRShift* (*k*). Um comportamento cíclico é descrito pelo emprego do combinador (**repeat**), uma vez que portas *stream* estão presentes na interface. Isso modela o fato de que, no código C original, a informação correspondente a estas portas é transmitida dentro do escopo de uma iteração, usando primitivas MPI. Cada elemento da *stream* corresponde ao valor transmitido em uma iteração. O componente (módulo funcional) associado às unidades *is_unit*, o qual define a computação realizada por estas, possui quatro argumentos e três pontos de retorno. O primeiro argumento é usado para passagem dos parâmetros que caracterizam o tamanho de problema em consideração. Os demais são mapeados às portas da interface.

Esqueletos MPI são empregados para constituição da topologia da rede de IS. São empregados dois esqueletos: AllReduce e AllToAllv. As portas *bs* são conectadas segundo o esquema descrito por AllReduce (ver apêndice A para maiores detalhes), enquanto as portas *kb* são conectadas segundo a topologia descrita por AllToAllv. Os aglomerados *bs_comm* e *kb_comm* são usados para configurar a topologia dos esqueletos MPI em IS.

Além dos dois esqueletos MPI acima descritos, é ainda empregado o esqueleto RShift, o qual modela um deslocamento à direita de dados entre uma coleção de m processos. Para isso, as unidades virtuais indexadas de 2 à $m-1$ dentro do contexto do aglomerado *k_shift* instanciam o esqueleto RShift. Cada uma destas possui uma porta de entrada e uma porta de saída. A unidade virtual indexada por 1 possui somente uma porta de saída, enquanto a unidade virtual indexada por m possui exatamente uma porta de entrada.

Os aglomerados abstratos *bs_comm*, *kb_comm*, and *k_shift* são sobrepostos pela unificação de suas unidades, de forma a configurar a topologia virtual da rede de processos de IS. Em seguida, as unidades não-virtuais *is_unit*'s são nomeadas para ocupar o lugar das unidades virtuais, configurando a rede final.

4.2.5.3 O Kernel CG (*Gradiente Conjugado*): O kernel CG implementa uma solução para um sistema linear esparso, não estruturado, baseado no método do gradiente conjugado. O método da potência inversa é usada para encontrar uma estimativa do

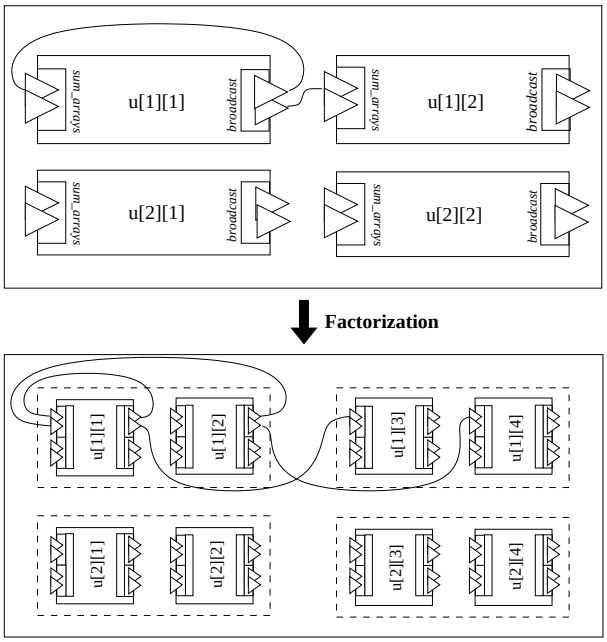


Figura 4.14. Topologia do Esqueleto TRANSPOSE

maior auto-valor de uma matriz esparsa definida, positiva e simétrica com um padrão randômico de valores não nulos.

A topologia original do kernel CG, especificada em FORTRAN/MPI, é composta por n processos, onde n é uma potência de 2, organizados em um *grid* retangular. Os elementos da matrix esparsa são uniformente distribuídos entre os processos de acordo com sua posição relativa. Assuma que r e c denotam o número de linhas e colunas do *grid* ($r \times c = n$), respectivamente. Se n é um quadrado perfeito então $r = c$, senão $c = 2 \times r$.

Com a finalidade de demonstrar o alto poder expressivo da linguagem # na especificação concisa de topologias complexas, uma generalização do esquema descrito acima foi usado na implemetação # de CG, porém de maneira mais elegante e simples que em sua versão original. Nesta versão, as unidades de CG continuam organizadas em um *grid* retangular, entretanto não há mais restrição para o valor que n pode assumir. O programador deve prover o parâmetro *dim*, cujo valor corresponde ao número de linhas do *grid*, e o parâmetro col_factor, o qual, quando multiplicado a *dim*, resulta no número de colunas do *grid* ($n = dim \times col\_factor$). Qualquer número de unidades pode ser configurado segundo este esquema, porém note que mais de uma combinação de valores para *dim* e col_factor resulta em um mesmo valor para n, induzindo topologias diferentes que podem afetar o comportamento dinâmico e desempenho do programa. Em uma aplicação real, o programador deve ajustar o valor desses dois parâmetros de forma a obter um melhor desempenho na arquitetura alvo, assumindo as características da instância de problema que será executada.

A interface que instancia as unidades *cg_unit[i]*, $1 \leq i \leq n$, descreve o comportamento de sete portas de entrada e nove portas de saída, definidas a partir da composição de instâncias das interfaces *IAllReduce* (rho, aux, $rnorm$, $norm\_temp\_1$ e $norm\_temp\_2$) e

ITranspose (q e r). As portas x e *zeta* foram especificadas explicitamente. A interface *ITranspose* descreve o comportamento de uma porta de entrada e uma porta de saída, nomeadas x e w, respectivamente e é usada na especificação do esqueleto TRANSPOSE. O módulo funcional CG possui seis argumentos e sete pontos de retorno. Com exceção do primeiro argumento, usado para informar o módulo funcional sobre a instância de problema que será executada, estes são mapeados às portas da interface da unidade.

Dois esqueletos são empregados para configuração da topologia de comunicação de CG: ALLREDUCE e TRANSPOSE. O primeiro é usado na troca de dados realizada em produtos escalares nos processos localizados na mesma linha do *grid*, enquanto o segundo é usado em multiplicações paralelas de matrizes, sempre que uma operação de transposição é realizada nos dados armazenados pelos processos no *grid*.

No código FORTRAN/MPI original, um esquema complicado é empregado, misturando o código que configura da topologia da rede de processos com chamadas às primitivas de comunicação de MPI e especificação de computações, tornando difícil o entendimento da estrutura topológica de CG sem uma análise cuidadosa dos valores dos parâmetros providos em cada instância de problema. O modelo # separa explicitamente estes componentes, em uma abordagem que incentiva a modularidade.

A cada linha de processos, três aglomerados ALLREDUCE são necessários para definir a topologia de CG, nomeadas $rho\_comm[i]$, $aux\_comm[i]$, $rnorm\_comm[i]$, $norm\_temp\_1[i]$ e $norm\_temp\_2[i]$, $1 \leq i \leq rows$. Estes descrevem, respectivamente, a topologia de interconexão das portas em rho, aux, $rnorm$, $norm\_temp\_1$ e $norm\_temp\_2$ das unidades que constituem CG. Somente dois aglomerados abstratos definem padrões de interação descritos pelo esqueleto TRANSPOSE, porém envolvendo todos os processos na rede. São chamados respectivamente q_comm e aux_comm, conectando as portas em q e aux, respectivamente, de cada unidade. Como nos outros *kernels*, as unidades virtuais no contexto dos aglomerados sufixados com *comm* são unificadas, formando a topologia virtual da rede onde estão organizadas as unidades de CG.

O código do esqueleto TRANSPOSE também encontra-se apresentado no apêndice B. Este esqueleto organiza unidades virtuais segundo os valores dos parâmetros *dim* e col_factor. Inicialmente, um *grid* quadrado de unidades, com dimensões *dim*, é organizado. As portas são conectadas de forma realizar uma operação de transposição dos dados armazenados pelas unidades, usando funções de ligação apropriadas sobre grupos de portas. Cada uma destas unidades é fatorada em col_factor unidades, resultando em um *grid* retangular com *dim* linhas e $dim * col\_factor$ colunas. O diagrama na Figura 4.14 ilustra a operação de fatoração que resulta na topologia das unidades virtuais do esqueleto TRANSPOSE. Com o objetivo de torná-la mais simples de entender, somente os canais conectados às portas da unidade $u[1][1]$ ports aparecem no diagrama. Note que, em consequência da fatoração, grupos aninhados de portas surgem na constituição da topologia de TRANSPOSE. Além disso, canais são replicados de acordo com as regras de fatoração especificadas na Seção 3.2.9.

4.2.5.4 A Aplicação Simulada LU: A aplicação LU provê a solução para um sistema de equações lineares resultantes da discretização de um sistema sintético de cinco equações diferencial parciais (EDP) não-lineares, representativo na solução de problemas

comuns em dinâmica dos fluidos computacional (CFD) [16]. Para solução do sistema em questão, é empregado o procedimento SSOR (*sobre-relaxação sucessiva e simétrica*[3]), cuja convergência depende de um parâmetro ω. Os processos, em potência de dois, estão organizados segundo um *grid*, sobre o qual o método *wavefront* é empregado extensivamente para comunicação [26]. Dentre os programas que compõem NPB, LU difere dos demais por ser o único que explora a comunicação de uma grande quantidade de mensagens curtas ($\cong$ 40 *bytes*).

Do ponto de vista do modelo # (Apêndice B.5), LU exercita a habilidade da linguagem de configuração # em lidar com processos com grande número de portas de entrada e saída, organizados por meio da sobreposição de vários esqueletos. Ao total, um processo LU é composto de 30 portas de entrada e 30 portas de saída, organizados pela sobreposição de 20 esqueletos, instanciados a partir de 8 componentes abstratos, dos quais 6 são próprios da aplicação e os demais esqueletos MPI. Estas características nos motivaram a desenvolver os mecanismos de modularização na descrição de interfaces, usando o operador #, simplificando o código, principalmente no que diz respeito a sua legibilidade, e otimizando a capacidade de reuso de partes.

Os esqueletos EXCHANGE_1B, EXCHANGE_3B, EXCHANGE_4, EXCHANGE_5 e EXCHANGE_6 implementam as topologias de comunicação em diversas fases de comunicação longo da execução do programa, a qual implementa o método *wavefront*. Sua nomenclatura segue o padrão usado nas versões originais de LU, escrita em FORTRAN/MPI, de fora a facilitar a análise comparativa entre as duas versões.

O uso de classes de interfaces permite a abstração da interface das unidades que comõem o *grid*. Nos esqueletos EXCHANGE_4, EXCHANGE_5 e EXCHANGE_6, as interfaces das unidades virtuais que compõem são diferentes de acordo com sua disposição no *grid*. Na configuração do esqueleto de aplicação LU, o uso de classes de interfaces evita assim que seja necessário tratar cada tipo de unidade de acordo com sua localização no *grid*, resultando em um código mais simples e modular.

4.2.5.5 As Aplicações Simuladas SP e BT: Três conjuntos de sistemas não-acoplados de equações, nas direções x, y e z, sequencialmente e nesta ordem são resolvidas pelas aplicações SP e BT. Esse sistema é escalar pentadiagonal em SP e tridiagonal com blocos 5x5 em BT. Um esquema de multi-partição é usado para solução desses sistemas em NPB 2.3, a partir de onde derivamos a solução #, uma vez que este oferece bom balanceamento de carga e usa comunicação de mais grossa granularidade[27]. Ambos assumem a existência um número de processos potência de 2. As topologias # de SP e BT são idênticas, sendo necessário portanto a implementação de um único esqueleto de aplicação (SOLVESYSTEM), comun à ambas as aplicações, a partir do qual estas são instanciadas. O código # de SP e BT é dessa forma apresentado no Apêndice B.4. A implementação de seus respectivos módulos funcionais diferencia os componentes BT e SP. Ilustra-se assim a reusabilidade de especificações topológicas, em nível de coordenação, uma importante funcionalidade provida pelo modelo # de programação paralela, possibilitada devido ao suporte a noção de esqueletos.

[3] *Sucessive and Symetric Over-Relaxation.*

Vale ressaltar que na implementação SP e BT emprega-se a mesma técnica de composição usada na implementação das demais versões # de programas NPB. Os esqueletos COPY_FACES, X_SOLVE, Y_SOLVE e Z_SOLVE descrevem fases de comunicação na versão FORTRAN/MPI padrão disponibilizada em NPB 2.3 e, assim como LU, a nomenclatura segue aquela empregada nas versões originais.

4.2.6 A Implementação dos Módulos Funcionais de EP, IS, CG, LU, SP e BT

Na versão atual para a linguagem #, denominada Haskell$_{\#}$, módulos funcionais devem ser programados em Haskell. Foi portanto necessário reescrever o código de cada um destes nesta linguagem. Nesta implementação, foram utilizadas *arrays unboxed* imutáveis (tipo `UArray`). No Capítulo 6, serão apresentados resultados que comparam a eficiência do código Haskell para lidar com aplicações científicas numéricas, com FORTRAN ou C. Esse trabalho tem nos sugerido técnicas que poderiam ser utilizadas sob a linguagem Haskell com a finalidade de melhorar seu desempenho, sobretudo no que diz respeito à computação sobre *arrays*.

Um trabalho que deverá a ser conduzido em breve é a inclusão do suporte a outras linguagens, em especial sob o paradigma imperativo, como C e Fortran, para construção dos componentes simples, de forma a ser possível que aplicações atualmente escritas nessas linguagens, com suporte paralelo via MPI, possam ser portadas para o ambiente #. Neste caso, os códigos sequenciais da versão original de NPB deverão ser reusados e ficará bastante evidente a superioridade expressiva do modelo #, sendo ainda possível realizar uma análise comparativa de desempenho apurada. Espera-se que a perda de eficiência seja mínima ou até mesmo nula, contrabalançeada positivamente pela maior modularidade e verificabilidade do código desenvolvido segundo o modelo #. No capítulo 6, ao apresentar-se como processos assíncronos pode ser implementados no modelo #, abre-se o caminho para que isso seja realidade.

4.3 O SUPORTE A ASPECTOS NO MODELO

O estilo de programação orientado à aspectos[139] surgiu ao final da década de 90 como uma extensão aos mecanismos de modularização convencionalmente empregados na construção de programas até então (Seção 4.1), os quais são incapazes de expressar a separação de funcionalidades que, embora conceitualmente distingüíveis no projeto de um *software*, encontram-se inerentemente entrelaçados ao longo código dos diversos módulos que compõem. A tais funcionalidades, foi atribuída a denominação de *aspectos*. Exemplos típicos de aspectos são: código de depuração de programas, protocolos de sincronização em programas concorrentes, observadores de estados dos objetos em um programa orientado à objetos, etc. A linguagem *AspectJ*[138] é a primeira a oferecer o suporte à modularização de aspectos, extendendo a linguagem Java.

No modelo #, aspectos em nível de coordenação podem ser modularmente programados sem que sejam necessárias extensões à sua linguagem de configuração, permitindo a descrição de funcionalidades topológicas e comportamentais presentes em um programa as quais encontram-se inerentemente entrelaçadas na configuração de cada unidade que

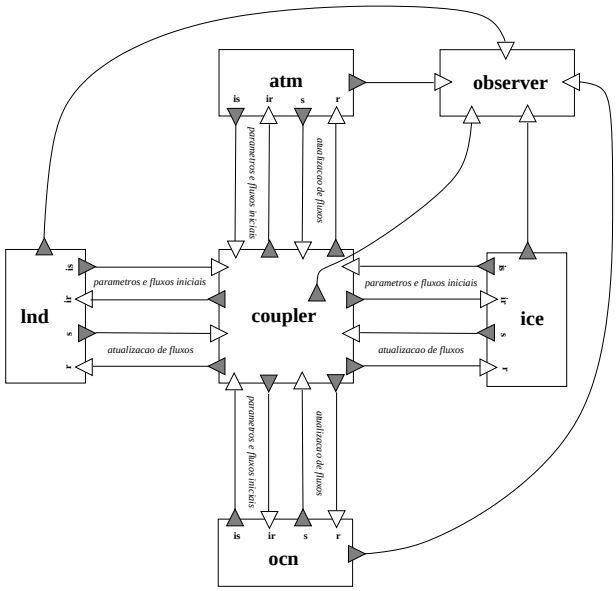


Figura 4.15. $CSM_{\#}$ com Processo Observador

compõe sua topologia. Com esse propósito, empregam-se os esqueletos topológicos parciais, devido a sua capacidade de sobreposição comportamental para formação de novas configurações que herdem caracterśticas dos esqueletos originais. O exemplo a seguir ilustra como aspectos podem ser dicernidos e modularizados na programação dentro do modelo #.

4.3.1 Um Observador de Estado para CSM

Um *observador de estado* consiste em um processo capaz de coletar periodicamente, em tempo de execução, informações relevantes a respeito do estado da computação de uma coleção de processos em um programa paralelo. No modelo #, a presença de um processo observador em um programa induz a existência de um aspecto, uma vez que é necessário que as unidades observadas enviem a informação sobre o seu estado explicitamente ao processo observador. Dessa forma, em uma implementação ingênua, o código de configuração que implementa a funcionalidade do *aspecto observador* encontra-se entrelaçada nas configurações de cada unidade observada e do próprio processo observador. Com o intuito de demonstrar como aspectos podem ser modularmente tratados na programação #, implementaremos uma nova versão para o programa $CSM_{\#}$, descrito na Seção 4.1.2, onde um processo observador colhe a cada iteração informações a respeito do estado corrente das unidades climáticas e aclopadora. Para uma aplicação de modelagem climática, um processo observador pode ser útil para permitir vários tipos de monitoração durante a execução, uma vez que esta pode executar durante um longo período de tempo.

Na Figura 4.3, é apresentada a rede de processos de $CSM_{\#}$, com o processo observador de estado incluído. Uma implementação trivial para esta configuração é apresentada na Figura 4.16. É fácil observar o entrelaçamento entre as configurações que descrevem o

```
configuration CSM< NCPL_A, NCPL_O > with

use CPL, ATM, LND, LND, OCN, ICE, Observer

interface IObserver (st_inf_atm*, st_inf_lnd*, st_inf_ocn*, st_inf_ice*, st_inf_cpl*) → ()
          behavior: repeat par {st_inf_atm?; st_inf_lnd?; st_inf_ocn?; st_inf_ice?; st_inf_cpl?}
                    until <st_inf_atm & st_inf_lnd & st_inf_ocn & st_inf_ice & st_inf_cpl >

interface ICoupler # (iar,ar*) → (ias,as*) # (ilr,lr*) → (ils,ls*) # (iir,ir*) → (iis,is*) # (ior,or*) → (ios,os*)
                  # () → (st_inf::StateInfo)
          behavior: seq { iar?; ias!; iir?; iis! ior?; ios?; ilr?; ils?; ir?; or?; lr?; ar?;
                          repeat seq {os!; repeat {st_inf!; lr?; as!; is! ir?; ar?}
                                          until counter (NCPL_A div NCPL_O);
                                      or?}
                          until <ls & lr & as & ar & is & ir & os & or >

interface IModel # (ir, r*) → (is, s*, st_inf)
          behavior: seq {is!; ir? repeat seq {st_inf!; s!; r?} until <r & s>}

unit cpl  # ICoupler as CPL (iar, iir, ilr, ior, ar, ir, lr, or) → (ias, iis, ils, ios, as, is, ls, os, st_info)
unit ocn # IModel   as OCN (ir, r) → (is, s, st_info)
unit atm # IModel   as ATM (ir, r) → (is, s, st_info)
unit lnd  # IModel   as LND (ir, r) → (is, s, st_info)
unit ice  # IModel   as ICE (ir, r) → (is, s, st_info)

unit observer # IObserver as Observer

connect atm.is to cpl.iar, synchronous
connect lnd.is  to cpl.ilr, synchronous
connect ice.is  to cpl.iir, synchronous
connect ocn.is  to cpl.ior, synchronous

connect cpl.ias to atm.ir, synchronous
connect cpl.ils to lnd.ir, synchronous
connect cpl.iis to ice.ir, synchronous
connect cpl.ios to ocn.ir, synchronous

connect atm.s  to cpl.ar, synchronous
connect lnd.s  to cpl.lr, synchronous
connect ice.s  to cpl.ir, synchronous
connect ocn.s  to cpl.or, synchronous

connect cpl.as  to atm.r, synchronous
connect cpl.ls  to lnd.r, synchronous
connect cpl.is  to ice.r, synchronous
connect cpl.os  to ocn.r, synchronous

connect cpl.st_inf to observer.st_inf_cpl, synchronous
connect atm.st_inf to observer.st_inf_atm, synchronous
connect ice.st_inf to observer.st_inf_ice, synchronous
connect lnd.st_inf to observer.st_inf_lnd, synchronous
connect ocn.st_inf to observer.st_inf_ocn, synchronous
```

Figura 4.16. Código de Configuração $CSM_{\#}$ com Observador

aspecto observador e a funcionalidade inerente ao CSM.

O mecanismo tradicional de modularização, descrito na Seção 4.1 sob a denominação de programação composicional, não pode ser aplicado na modularização do aspecto observador, devido ao seu inerente entrelaçamento com a funcionalidade da aplicação. Entretanto, usando o conceito de sobreposição de esqueletos topológicos, a construção de uma implementação modularizada e extensível é possível. Os passos para derivá-la são enumerados a seguir e se aplicam a qualquer caso de ocorrência de aspectos em programas #:

i) Separar a descrição topológica da aplicação da descrição da funcionalidade das unidades que a compõe, construindo-se o chamado *esqueleto de aplicação fundamental*;

ii) Especificar um esqueleto que descreva a topologia do aspecto, chamado *esqueleto de aspecto*. Um esqueleto de aspecto pode ser reusável por outras aplicações;

```
configuration CSM_Skeleton< ncpl_a, ncpl_o > with

interface ICoupler # (iar,ar*) → (ias,as*)
                   # (ilr,lr*) → (ils,ls*)
                   # (iir,ir*) → (iis,is*)
                   # (ior,or*) → (ios,os*)
          behavior: seq { iar?; ias!; iir?; iis! ior?; ios?; ilr?; ils?;
                          ir?; or?; lr?; ar?;
                          repeat seq {os!; repeat {lr?; as!; is! ir?; ar?}
                                          until counter (ncpl_a div ncpl_o);
                                      or?}
                          until <ls & lr & as & ar & is & ir & os & or >

interface IModel # (ir, r*) → (is, s*)
          behavior: seq {is!; ir? repeat seq {s!; r?} until <r & s>}

unit cpl  # ICoupler
unit ocn  # IModel
unit atm  # IModel
unit lnd  # IModel
unit ice  # IModel

connect atm.is to cpl.iar, synchronous
connect lnd.is  to cpl.ilr, synchronous
connect ice.is  to cpl.iir, synchronous
connect ocn.is  to cpl.ior, synchronous

connect cpl.ias to atm.ir, synchronous
connect cpl.ils to lnd.ir, synchronous
connect cpl.iis to ice.ir, synchronous
connect cpl.ios to ocn.ir, synchronous

connect atm.s  to cpl.ar, synchronous
connect lnd.s  to cpl.lr, synchronous
connect ice.s  to cpl.ir, synchronous
connect ocn.s  to cpl.or, synchronous

connect cpl.as to atm.r, synchronous
connect cpl.ls to lnd.r, synchronous
connect cpl.is to ice.r, synchronous
connect cpl.os to ocn.r, synchronous
```

Figura 4.17. Componente que Descreve a Topologia de CSM$_{\#}$ (Esqueleto de Aplicação)

iii) Construir uma nova versão do esqueleto de aplicação com a inclusão do aspecto. Para isso, é necessário sobrepôr o esqueleto de aplicação fundamental e o esqueleto que descreve o aspecto. Vários esqueletos de aspecto podem ser sobrepostos ao esqueleto de aplicação neste processo. Ainda é possível especificarem-se várias versões do esqueleto de aplicação, com suporte a diferentes aspectos;

iv) Construir o componente não-virtual que descreve a aplicação, a partir do esqueleto de aplicação fundamental ou um dos esqueletos de aplicação resultantes da inserção de aspecto(s) no passo (iii).

O código do esqueleto fundamental de CSM$_{\#}$ (CSM_Skeleton) encontra-se apresentado na Figura 4.17. Observe que a única diferença desta configuração para aquela apresentada na Figura 4.2 é a não especificação dos módulos funcionais que descrevem o comportamento computacional das unidades, constituindo assim um esqueleto total. O código do esqueleto que descreve a topologia do aspecto observador encontra-se apresentado na Figura 4.18. No código da Figura 4.19, o esqueleto fundamental de CSM$_{\#}$ é sobreposto ao esqueleto Observer, o qual descreve o *aspecto observador*, formando o esqueleto CSM_Observed_Skeleton. Este é empregado para definir a configuração do componente que define a aplicação CSM$_{\#}$ com observador de estado, cujo código encontra-se na Figura 4.16.

```
configuration Observer<N,COMBINE> where

index i range [1,N]

interface IObserved () → st_inf*
          behavior: repeat st_inf! until st_inf

[/unit observed[i] # IObserved/]

interface IObserver st_inf* → ()
          behavior: repeat st_inf? until st_inf

unit observer # IObserver groups st_inf<N>:COMBINE

[/ connect observed[i]→st_inf to observer←st_inf[i] /]
```

Figura 4.18. Configuração do Aspecto Observador (Esqueleto de Aspecto)

```
configuration CSM_Observed_Skeleton where

use Observer
use CSM_Skeleton

unit csm as CSM_Skeleton
unit eyes as Observer< 5 >

interface IObservedCoupler # icpl like ICoupler # ieye like IObserved
interface IObservedModel # imod like IModel # ieye like IObserved

unify csm.cpl # icpl, eyes.observer[1] # ieye to ocpl # IObservedCoupler
unify csm.atm # imod, eyes.observer[2] # ieye to oatm # IObservedModel
unify csm.ice # imod, eyes.observer[3] # ieye to oice # IObservedModel
unify csm.lnd # imod, eyes.observer[4] # ieye to olnd # IObservedModel
unify csm.ocn # imod, eyes.observer[5] # ieye to oocn # IObservedModel
```

Figura 4.19. Componente Abstrato para a Topologia de $CSM_{\#}$ com a Introdução do Aspecto Observador

```
configuration CSM_Observed where

use CPL, ATM, ICE, LND, OCN
use CSM_Observed_Skeleton

unit csm as CSM_Observed_Skeleton

unit cpl # IObservedCoupler as CPL
unit atm # IObservedModel  as ATM
unit ice # IObservedModel  as ICE
unit lnd # IObservedModel  as LND
unit ocn # IObservedModel  as OCN

assign cpl to csm.ocpl
assign atm to csm.oatm
assign lnd to csm.olnd
assign ocn to csm.oocn
```

Figura 4.20. Configuração de $CSM_{\#}$ Descrevendo a Funcionalidade das Unidades

O procedimento de derivação descrito vale para o desenvolvimento de qualquer programa # que deseje fazer uso do conceito de aspectos para modularizar funcionalidades inerentemente entrelaçadas nas unidades que compõem a topologia do programa. O mecanismo descrito separa o comportamento inerente ao programa paralelo, descrito pelo esqueleto de aplicação fundamental, dos aspectos nela introduzidos através dos seus respectivos esqueletos.

4.4 PARALELIZANDO UM PROGRAMA SEQÜENCIAL PRÉ-EXISTENTE

Nesta seção, é demonstrado como um programa sequencial pode ser paralelizado dando origem a uma versão #. O mecanismo não exige a necessidade de modificação no código sequencial original, a menos da reorganização das funções que compõem o programa em módulos funcionais Haskell$_{\#}$, os quais, como vimos, confundem-se com meros módulos Haskell. Devido a este fato, em programas Haskell bem estruturados modularmente, pode ocorrer que não seja necessário nem mesmo a reorganização das funções, constituindo-se os módulos originais os próprios módulos funcionais. Esta característica se deve a hierarquia de processos suportada pelo modelo #, o qual separa as preocupações de configuração da coordenação entre processos e especificação das computações em níveis ortogonais de programação.

A aplicação escolhida para demonstrar os aspectos descritos é MCP-Haskell[108], uma versão Haskell simplificada de MCNP (Monte Carlo N-Particle), aplicação desenvolvida durante muitos anos em Los Alamos [232]. Esta envolve a simulação estatística do comportamento de partículas (fótons, nêutrons, elétrons, etc.) quando trafegam através de um conjunto de objetos de formatos e materiais especificados. Em [108], duas implementações de uma versão simplificada de MCNP são apresentadas, respectivamente escritas nas linguagens Haskell (MCP-Haskell) e Id[113] (MCP-Id), com o fim de comparar o desempenho destas. A versão # para MCP-Haskell, aqui apresentada, denomina-se MCP-Haskell$_{\#}$.

Características como importância, inerente natureza paralela e disponibilidade de implementação Haskell foram as motivações para a escolha de MCP-Haskell para demonstrar a paralelização de código sequencial para a linguagem #. Um dos fatos mais importantes demonstrados com a paralelização de MCP-Haskell para o modelo # é a liberdade dada ao programador para concentrar-se somente nas tarefas inerentes ao processo de paralelização propriamente dito, abstraindo-se de preocupações e suposições sobre o código que implementa as computações realizadas pelas tarefas paralelas.

4.4.1 Decomposição Funcional de MCP-Haskell

A análise da implementação sequencial de MCP-Haskell permitiu a identificação do seguinte conjunto de módulos funcionais:

- *ProblemDef*: Obtém os parâmetros que configuram a instância do problema que será executada a partir de uma arquivo em disco ($inp2$). Estes são enviados através de pontos de retorno aos módulos responsáveis pelas computações propriamente ditas, configurando a computação a ser realizada;

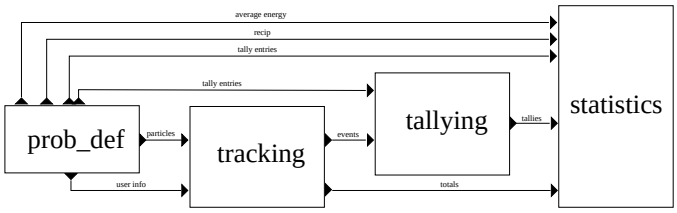


Figura 4.21. Rede de Processos Preliminar de MCP-$Haskell_{\#}$

- *Tracking*: Recebe como entrada parâmetros que caracterizam a instância em consideração do problema e uma *stream* (lista) de fótons. O movimento e colisões com o núcleo de cada partícula são simulados, produzindo uma lista de eventos para cada partícula que é enviado através de um ponto de retorno. É ainda computada e transmitida a informação a respeito de todos os ganhos e perdas de fótons que ocorrem durante a simulação;

- *Tallying*: Recebe como entrada uma *stream* de listas de eventos para cada fóton, construindo um resumo estatístico (desvio padrão e médio) do comportamento de cada partícula, de forma que estimativas de erro possam ser produzidas;

- *Statistics*: Recebe como entrada um conjunto de parâmetros que permitem que este módulo efetue um conjunto de cálculos estatísticos sobre informações recebidas dos módulos *Tracking* e *Tallying*, os quais caracterizam o resultado da simulação;

As funções em MCP-Haskell que implementam os módulos funcionais acima descritos são reorganizadas, sem necessidade de modificações, com a finalidade de construirem-se os módulos funcionais que serão usados para compôr o programa #. O código destes encontra-se apresentado na Seção C.

4.4.2 Configurando a Topologia da Rede de Processos

Na Figura 4.21, é apresentada a topologia preliminar de MCP-$Haskell_{\#}$, após a decomposição funcional de MCP-Haskell. A cada módulo funcional é associada uma unidade na configuração. A estrutura sequencial de dependência entre as unidades sugere, a princípio, que nenhum paralelismo real é suportado. Entretanto, uma vez que as unidades *tracking* e *tallying* atuam sobre cada partícula individual recebida a partir de *prob_def*, uma estrutura de *pipe-line* pode ser usado para que *tracking* e *tallying* possam atuar em paralelo, como sugere a existência da unidade *tracking_tallying* no código # apresentado na Figura 4.22. Neste, omitimos a declaração das interfaces das unidades. No apêndice C, apresentamos o código completo de MCP-$Haskell_{\#}$, incluídos os seus módulos funcionais.

O paralelismo explorado na versão de MCP-$Haskell_{\#}$ descrito na Figura 4.22 é funcional, no sentido de que este é induzido pela topologia em que se encontram inerentemente organizadas as unidades induzidas pelos módulos funcionais da aplicação.

O *paralelismo funcional* inerente à dependência de dados entre os módulos funcionais de MCP-$Haskell_{\#}$ não constitui a única, nem a mais viável, fonte de paralelismo que

```
component MCP with

interface IProbDef # () → (user_info, particles, tally_entries, recip, avg_e, all_tallies)
          behavior: seq { recip!; avg_e!; all_tallies!; tally_entries!; user_info!;
                          repeat particles! until particles }

interface ITracking # (user_info) → (totals)
                    # process_particles like IPipe particles → events
          behavior: seq { user_info?;
                          do process_particles;
                          totals! }

interface ITallying # (tally_entries) → ()
                    # process_events like IPipe events → tallies
          behavior: seq { tally_entries?;
                          do process_events }

interface IStatistics # (avg_e, recip, totals, tallies) → ()
          behavior: seq { avg_e?; recip?; all_tallies?;
                          repeat tallies? until tallies;
                          totals? }

unit tracking_tallying # particles → tallies as Pipe_Line<2>

unit prob_def  # IProbDef  as ProbDef
unit tracking  # ITacking  as Tracking
unit tallying  # ITallying  as Tallying
unit statistics # IStatistics as Statistics

assign tracking # process_particles to tracking_tallying.pipe[1]
assign tracking # process_events    to tracking_tallying.pipe[1]

connect prob_def→user_info        to tracking←user_info,           synchronous
connect prob_def→particles        to tracking_tallying←particles,  synchronous
connect prob_def→tally_entries[1] to tallies←tally_entries,        synchronous
connect prob_def→tally_entries[2] to statistics←tally_entries,     synchronous
connect prob_def→recip            to statistics←recip,             buffered
connect prob_def→avg_e            to statistics←avg_e,             buffered
connect tracking→totals           to statistics←totals,            buffered
connect tracking_tallying→tallies to statistics←tallies,           synchronous
```

Figura 4.22. Código # da Topologia Preliminar de MCP-Haskell$_{\#}$

pode ser explorada nesta aplicação. Podemos ainda empregar o *paralelismo de dados*, tomando vantagem de uma de suas importantes características: as estatísticas que descrevem o comportamento de cada partícula podem ser obtidas independentemente. Portanto, várias instâncias do aglomerado *tracking_tallying* podem ser usadas para processar em paralelo sub-conjuntos de partículas. Para implementar o paralelismo de dados induzido pela independência das partículas é suficiente replicar a unidade *tracking_tallying* em n cópias, sendo n um parâmetro estático que deve ser acrescentado ao cabeçalho da configuração, como ilustrado a seguir:

```
component MCP< n > with
⋮
replicate n tracking_tallying # particles → tallies
        connections particles<>: choice
                    tracking←user_info<>: broadcast
                    tracking→totals<>: {# (map.sum.transpose) #}
                    tallying←tally_entries<>: broadcast
                    tallies<>: concat
```

Observe que as partículas são agora distribuídas a partir do agrupamento de portas **choice** *particles* da unidade $prob\_def$. As portas desse agrupamento estão conectadas as portas *particles* de cada unidade *tracking* induzida na replicação. Dessa forma, as partículas são requisitadas sob demanda pelas unidades *tracking*, efetivando um balanceamento automático de carga.

Outra fonte de paralelismo que pode ser explorado nesta aplicação leva em consideração o fato de que as computações de cada uma das diferentes estatísticas realizadas sobre os dados recebidos pela unidade *statistics* podem ser realizadas em paralelo por instâncias replicadas desta unidade. Em um esquema semelhante ao anterior, adiciona-se um parâmetro estático m a configuração, indicando o número de cópias a serem criadas para a unidade *statistics*, como a seguir:

```
component MCP< n, m > with
⋮
replicate m statistics # (avg_e, recip, totals, tallies,tally_entries) → ()
        connections avg_e<>: broadcast
                    recip<>: broadcast
                    totals<>: {# (map.sum.transpose) #}
                    tally_entries<>: distribute
                    tallies<>: broadcast
```

Na Figura 4.23, ilustramos a topologia resultante, assumindo-se os valores 4 e 2 como parâmetros atuais para n e m, respectivamente. Figuras de desempenho de MCP-Haskell$_{\#}$ são apresentadas na Figura 4.24, em versões sobre 2, 4 e 8 processadores de um *cluster*, descrito no Capítulo 6. O código # de MCP-Haskell$_{\#}$, incluindo seus módulos funcionais, é apresentado no apêndice C.

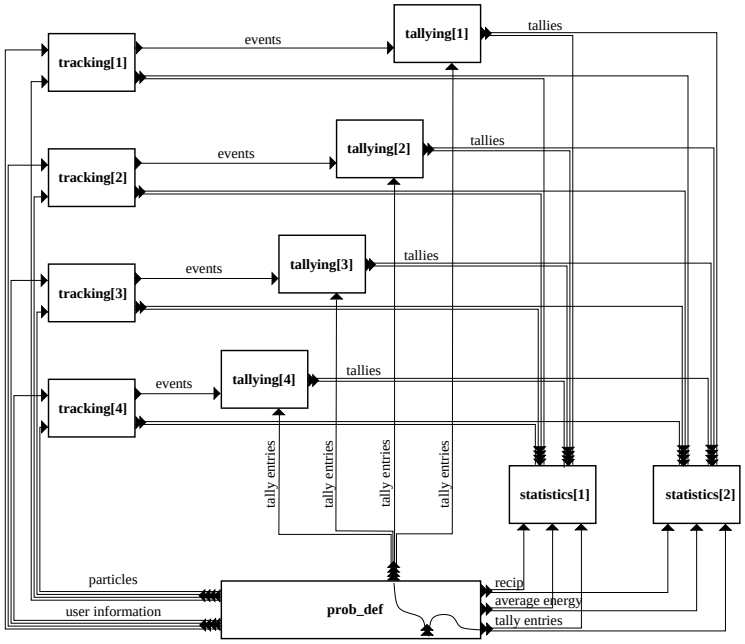


Figura 4.23. Rede de Processos de MCP-$Haskell_{\#}$

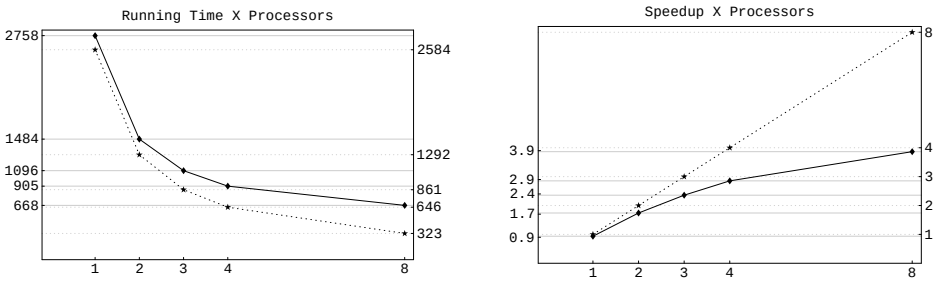


Figura 4.24. Tempo de Execução e *Speedup* obtido para MCP-$Haskell_{\#}$

4.5 DERIVANDO PROGRAMAS # A PARTIR DE PROGRAMAS MPI SPMD

Atualmente, a maior parte dos programas que implementam aplicações científicas paralelas de alto desempenho encontram-se implementados nas linguagens C e Fortran, utilizando bibliotecas de passagem de mensagens (como PVM ou MPI) para o gerenciamento do paralelismo, obedecendo ao estilo SPMD (*Single Program, Multiple Data*) de programação paralela. Com a consolidação das arquiteturas de memória distribuída, a disseminação do uso de *clusters* de baixo custo e a recente interconexão de grandes centros de supercomputação, em especial nos EUA, por meio de tecnologias de comunicação a longa distância de alto desempenho, os chamados *grids*, essa tendência tende a se solidificar. Esse é um dos fatores que motivaram o desenvolvimento do modelo #, como discutido no Capítulo 1. A partir dessa constatação, advém a necessidade de discutir-se a portabilidades de programas paralelos SPMD pré-existentes para o modelo #. Com essa finalidade, foi importante a experiência adquirida ao portar-se os programas NPB e CSM para o modelo #, uma vez que estes podem ser considerados representativos da classe de programas paralelos em questão. CSM é um caso especial, por se tratar de uma aplicação MPMD (*Multiple Program Multiple Data*).

Como consequência da hierarquia de processos, a técnica apresentada a seguir concentra-se na especificação do código # da aplicação. Suposições a cerca da programação em nível de computação são assim desnecessárias. Os passos da técnica de construção do modelo # de um programa SPMD MPI são enumerados e explicados nas seções que se seguem. O kernel IS será usado para exemplificar os passos descritos. Entretanto algumas observações são necessárias:

- A técnica descrita é facilmente extensível para outras bibliotecas de passagem de mensagens, alternativas à MPI;
- A técnica é também extensível para programas MPMD, como foi demonstrado a portarmos a aplicação CSM;
- O programador não precisa entender a computação realizada pelo programa para aplicar os passos descritos. Pode-se abstrair-se completamente funcionalidade da aplicação, uma vez que o nível de programação em questão é o de coordenação.

4.5.1 Identificando Pontos de Sincronzação

A identificação dos pontos de sincronização do programa consiste, a grosso modo, na inspeção do código imperativo em busca de chamadas à rotinas de comunicação, as quais devem ser catalogadas e documentadas. Como exemplo, consideraremos o caso do kernel IS. Neste, são identificados os pontos de sincronização ilustrados na Figura 4.25, em negrito. Observe-se a existência de chamadas à rotinas de comunicação coletiva de MPI. Isto sugerirá adiante o emprego de esqueletos MPI para construção da topologia de processos desta aplicação.

```
main( argc, argv )
      .
      .
      .
      /* This is the main iteration */
      for( iteration=1; iteration<=MAX_ITERATIONS; iteration++ )
      {
        if( my_rank == 0 && CLASS != 'S' ) printf( "
        rank( iteration );
      }
      .
      .
      .
      full_verify();

      MPI_Reduce(&itemp,&passed_verification,
                 1,MPI_INT,MPI_SUM,MPI_COMM_WORLD );

void rank( int iteration )
      .
      .
      .
      MPI_Allreduce( bucket_size,bucket_size_totals,
                     NUM_BUCKETS+TEST_ARRAY_SIZE,
                     MPI_INT,MPI_SUM,MPI_COMM_WORLD );
      .
      .
      .
      MPI_Alltoall(send_count,1,MPI_INT,
                   recv_count,1,MPI_INT,MPI_COMM_WORLD );
      .
      .
      .
       MPI_Alltoallv(key_buff1,send_count,send_displ,MPI_INT,
                     key_buff2,recv_count,recv_displ,MPI_INT,
                     MPI_COMM_WORLD );
      .
      .
      .

void full_verify()
      .
      .
      .
      if( my_rank > 0 )
        MPI_Irecv(&k,1,MPI_INT,my_rank-1,1000,MPI_COMM_WORLD,&request );
      if(my_rank < comm_size-1 )
        MPI_Send(&key_array[total_local_keys-1],1,MPI_INT,my_rank+1,1000
                 MPI_COMM_WORLD );
      if(my_rank > 0 )
        MPI_Wait( &request, &status );
```

Figura 4.25. Inspecionando Pontos de Sincronização e Fluxo de Controle do *kernel* IS

4.5.2 Identificando os Argumentos e Pontos de Retorno do Componente Simples

O conjunto de argumentos e pontos de retorno do componente simples, ponto de partida para inferir-se a interface das unidades que compõem a aplicação, pode ser induzido analisando-se a semântica associada às operações de comunicação catalogadas pela etapa anterior. É comum que a cada chamada a uma primitiva de comunicação seja necessário associar um único argumento ou ponto de retorno, dependendo da natureza da operação. Em alguns casos especiais, porém, pode ser necessário associar um argumento e um ponto de retorno, como ocorre com algumas primitivas onde dados são simultaneamente recebidos e enviados pelo processo. IS (Figura 4.25) é um exemplo onde essa regra se aplica. É ainda possível que várias ocorrências distintas de chamadas à rotinas de comunicação, as quais em conjunto possuem uma mesma finalidade, sejam associadas a um mesmo argumento ou ponto de retorno, como ocorre no kernel CG, onde existem várias chamadas à MPI ao longo do código onde o valor de uma mesma variável *rho* é trocada entre os processos e somada em cada um. Por opção do programador, visando simplicidade ou melhor desempenho, cada operação pode ser implementada por uma sequência de chamadas às primitivas MPI_Irecv e MPI_Send, equivalente a uma única chamada à primitiva $MPI\_AllReduce$. As diversas chamadas correspondem a existência de um único argumento rho_g e um único ponto de retorno rho_l. Isso sugere a necessidade de uma regra que ajude na identificação de grupos de chamadas que correspondem a um mesmo argumento/ponto de retorno. Dessa forma, estabelece-se que duas chamadas a primitivas de comunicação correspondem a um mesmo argumento/ponto de retorno sempre que:

- A informação que está sendo transmitida, a qual captura o significado do dado sendo transmitido no contexto da aplicação, é a mesma;
- As chamadas envolvem comunicação entre os mesmos processos, obedecendo uma certa topologia.

Considerando o caso de IS, a primeira chamada (*MPI_Reduce*), a qual constitui uma comunicação coletiva do tipo *todos-para-um*, induz a existência dos argumentos e pontos de retorno respectivamente denominados pv_g e pv_l em um dos processos instanciados, denominado *root*. Os demais, nomeados *peer*, necessitariam apenas do ponto de saída pv_l. Com a finalidade de manter a simetria dos processos e evitar a necessidade de criar vários componentes simples com interfaces ligeiramente diferentes porém implementando a mesma funcionalidade, ao invés de considerar uma chamada à *MPI_Reduce*, consideraremos uma chamada à *MPI_Allreduce*, de forma que todos os processos receberão o valor que antes era recebido somente pelo processo *root*. Analogamente, a segunda chamada ($MPI\_AllReduce$), também coletiva, induz a existência dos pontos de entrada *bst* e de saída *bs*.

Um caso particular ocorre na terceira e quarta chamadas (*MPI_Alltoall* e *MPI_AllToAllv*). No código IS, estas se tratam na realidade da transmissão de uma mesma informação. A primeira chamada é de controle, apenas para os programas informarem-se sobre os espaços que serão usados nos *buffers* de recebimento por ocasião da transmissão dos dados em si, efetivada na segunda chamada. Por se tratar de uma troca coletiva de dados

do tipo *todos-para-todos* pode ser modelado na linguagem # apenas com um argumentos e um ponto de retorno, indentificados respectivamente por $kb2$ e $kb1$. Entretanto, vale ressaltar, que seria possível considerar duas chamadas independentes e induzir portas separadamente para cada uma. Entretanto, dessa forma, limitações inerentes à MPI seriam levadas ao meio de coordenação do programa #, o qual admite suposições mais flexíveis.

Por fim, as duas chamadas realizadas no contexto da função $full\_verify$ induzem, respectivamente, um ponto de saída k_r e uma ponto de entrada k_l. O teste verifica se o processo instanciado é o mais à esquerda ou o mais à direita. Para estes, respectivamente não seriam necessárias as portas k_l e k_r, respectivamente. Entretanto, pelo mesmo motivo exposto para o primeiro caso, assume-se a existência de ambas.

4.5.3 Construindo a Interface dos Processos

Nesta etapa, inicia-se a modelagem das unidades que constituirão a rede de processos da aplicação. Pelo fato de estarmos tratando de programas SPMD, existe um único componente simples, ao qual são associadas as unidades (processos) que constituem a topologia. Deve-se então definir a interface comum a estas unidades, o que será detalhado nos próximos parágrafos.

4.5.3.1 Identificando Portas de Comunicação As portas de entrada e saída que constituem a interface são induzidas a partir dos argumentos e pontos de retorno, respectivamente, do componente simples associado à unidade. Replicação de portas deve ser empregada sempre que um valor de um ponto de retorno deva ser enviado a vários processos ou valores recebidos de vários processos devam ser reduzidos a um único argumento. Entretanto, a necessidade de replicação de portas somente torna-se óbvia adiante, quando é tratada a interconexão das portas entre os processos.

4.5.3.2 Inferindo o Comportamento da Interface O comportamento da interface é inferido analisando-se o fluxo de controle do programa original, com a finalidade de deduzir-se a ordem em que são realizadas as chamadas às rotinas de comunicação. Analisando o fluxo de controle de IS na Figura 4.25, observa-se que a função *rank* é chamada dentro do escopo de uma iteração por um número fixo de repetições ($MAX\_ITERATIONS$). No corpo desta função, são efetuadas as chamadas correspondentes às ativações das portas *bst*, *bs*, $kb2$ e $kb1$. Ao final de uma iteração, ocorre a chamada à rotina $full\_verify$, onde ocorrem as chamadas correspondentes às ativações das portas k_r e k_l. Posteriormente, efetiva-se a chamada correspondente às portas pv_g e pv_l. Assim, podemos definir o comportamento da interface das unidades de IS (IIS) da seguinte forma:

```
⋮
interface IIS # (bst,kb2,k_r,pv_g) → (bs,kb1,k_l,pv_l)
        behavior: seq { repeat seq {bs!; bst?; kb1! kb2?} counter MAX_ITERATIONS;
                        k_r!; k_l?;
                        pv_l!; pv_g?
                      }
⋮
```

A iteração é modelada pelo combinador **repeat**, sendo as portas ativadas em seu contexto (*bst*, *bs*, *kb*2 e *kb*1) consideradas portas *stream*. Quando ocorrem chamadas em laços aninhados, devem-se empregar *streams* aninhadas ativadas no contexto de combinadores **repeat** aninhados. Esse caso ocorre no kernel CG.

4.5.3.3 Declarando as Unidades A seguir, é exemplificado o código que declara as unidades do kernel IS, uma vez conhecidas suas interfaces e o componente simples associado:

```
⋮
[/ unit is_unit[i] # IIS groups bst<>:sum, kb2's<>:combine_keys, bs<>:broadcast, kb1<>:distribute_keys)
        as IS (Params PROBLEM_CLASS NUM_PROCS MAX_KEY_LOG2
                      NUM_BUCKETS_LOG2 TOTAL_KEYS_LOG2
                      MAX_ITERATIONS MAX_PROCS TEST_ARRAY_SIZE,
               bst, kb2, k_l) → (bs, kb1, k_r)
/]
⋮
```

4.5.4 Definição da Topologia da Rede de Unidades

Conhecida a interface das unidades da rede, deve-se analisar a interconectividade induzida pelas chamadas às primitivas de comunicação. Evidentemente, a análise é diferente de acordo com a natureza da comunicação efetivada pela primitiva, a qual pode ser *ponto-a-ponto* ou *coletiva*. Como discutiremos nos próximos parágrafos, não é suficiente observar-se isoladamente cada chamada, devendo-se analisar a inter-relação entre estas, de forma a identificar casos onde podem ser agrupadas em um mesmo esqueleto apropriado.

Em primitivas ponto-a-ponto, deve-se observar os campos que indicam o emissor/receptor da mensagem e o *tag*, este último geralmente empregado para distingüir mensagens que trafegam entre dois nós determinados. Muitas vezes, como ocorre em CG, cálculos nem sempre triviais são empregados para computar-se o valor destas informações, com o fim de determinar a topologia dos processos em tempo de execução, o que deve ser analisado cuidadosamente para uma tradução correta para a liguagem #.

O uso de primitivas de comunicação coletiva, quando não ocorre sobre o comunicador padrão `MPI_COMM_WORLD`, o qual envolve todos os processos, causa a necessidade de analisar-se a ocorrência de comunicadores de grupos definidos pelo usuário, identificando os grupos de processos participantes em cada operação.

Um caso especial ocorre no kernel CG, ilustrando que o programador # deve levar em consideração a relação entre as chamadas às primitivas de comunicação. Nesse *kernel*, uma comunicação coletiva do tipo *todos-para-todos* ocorre entre os processos localizados em uma mesma linha com a finalidade de somar os valores que estes armazenam de uma variável *rho*. Embora isso pudesse ser resolvido pelo emprego da primitiva de comunicação coletiva *MPI_AllReduce* sobre grupos de processos pertencentes a uma mesma linha, configurados pelo usuário, o programador opta por realizar esta operação por várias chamadas às primitivas ponto-a-ponto *MPI_Irecv* e *MPI_Send*, implementando um algoritmo *butterfly* de disseminação para que todos os processos conheçam a soma dos *rho*'s respectivamente armazenados por cada processo. Na modelagem da configuração # desta aplicação, ao invés de utilizarmos o mesmo recurso, utilizamos o esqueleto `AllReduce` diretamente, uma abordagem bem mais concisa e elegante.

No kernel IS, são resultados desta etapa os padrões topológicos de comunicação induzidos pelas chamadas às primitivas de comunicação coletiva associadas aos pares de portas *bst*/*bs* (*MPI_AllReduce*), *kb*2/*kb*1 (*MPI_AllToAllv*) e *pv_g*/*pv_l* (*MPI_AllReduce*), todos pertencentes ao padrão topológico primitivo *todos-para-todos* de comunicação coletiva. Uma padrão especial é induzido pelas chamadas associadas às portas *k_r*/*k_l*, onde os processos, organizados em série, enviam informação para o seu vizinho à direita e recebem do seu vizinho à esquerda. Notadamente, os processos localizados nas extremidades à direita e à esquerda não possuem porta *k_l* e *k_r*, respectivamente.

4.5.5 Expondo e Modularizando Padrões Topológicos com Esqueletos

Nesta etapa, são identificados os possíveis esqueletos que descrevem a topologia da rede de unidades da aplicação em questão. Esqueletos surgem comumente em duas situações:

i) Reusar padrões topológicos pré-definidos, possivelmente capazes de informar ao compilador informações que podem ser usadas na geração de código mais eficiente e alocação otimizada de processos aos processadores (balancemento de carga estático);

ii) Modularização da configuração, separando padrões topológicos de comunicação que se encontram sobrepostos mas que dizem respeito a preocupações diferentes na aplicação.

Neste sentido, o uso de esqueletos MPI para representar os padrões topológicos induzidos pelas ocorrências de primitivas de comunicação coletiva no código da aplicação, pode ser considerada uma instância do primeiro caso. Este recurso é extensivamente utilizado na implementação # dos *kernels* e aplicações de NPB. No kernel IS, por exemplo, são criados aglomerados abstratos associados aos esqueletos AllReduce, AllToAllv, para implementar os padrões topológicos, identificados na etapa anterior, envolvendo os pares de portas *bst*/*bs*, *kb*2/*kb*1 e *pv_g*/*pv_l*:

```
⋮
unit pv_comm as AllReduce<num_procs, MPI_SUM, MPI_INTEGER>
unit bs_comm as AllReduce<num_procs, MPI_SUM, MPI_INTEGER>
unit kb_comm as AllToAllv<num_procs>
⋮
```

Adicionalmente, é necessário redefinir a interface *IIS* de forma a que esta seja composta a partir das interfaces de unidades dos esqueletos MPI correspondentes:

```
⋮
interface IIS # bs* like IAllReduce (UArray Int Int)
              # kb* like IAllToAllv (Int, Ptr Int)
              # pv  like IAllReduce (UArray Int Int)
              # k   like RShift     Int
              behavior: seq {repeat seq {do bs; do kb} until <bs.in & kb.in>;
                             do pv; do k}
⋮
```

No kernel CG, a identificação do esqueleto Transpose, o qual estabelece um padrão peculiar de topologia que se repete entre subconjuntos de canais que compõem a rede, constitui uma instância do segundo caso. O mesmo se pode dizer com respeito ao esqueleto RShift, em IS, o qual implementa o padrão topológico descrito no último parágrafo da seção anterior, envolvendo as portas *k_r* e *k_l* presentes nas unidades que a compõem. Outros exemplos são os esqueletos X_Solve, Y_Solve, Z_Solve e Copy_Faces, os quais ocorrem nas aplicações BT e SP, e os esqueletos Exchange_1b, Exchange_3b, Exchange_4, Exchange_5 e Exchange_6, cuja sobreposição constitui a rede de processos da aplicação LU, também surgidos por motivações de modularidade.

4.5.6 Compondo Esqueletos para Formação de uma Topologia Virtual

Reconhecidos os esqueletos que compõem os padrões topológicos induzidos pelas chamadas de comunicação no programa, é necessário utilizar a técnica de sobreposição de esqueletos com a finalidade de compor-se uma topologia de unidades virtuais que corresponde a topologia abstrata da aplicação. Isso pode ser realizado utilizando-se a operação de unificação, sobre as unidades virtuais que constituem os esqueletos que definem as partes que compõem a topologia. No caso do kernel IS, isso é efetivado pela seguinte declaração:

```
⋮
[/ unify bs_comm.p[i] # bs,
         kb_comm.p[i] # kb,
         pv_comm.p[i] # pv,
         k_comm.p[i]  # k
         to virtual_is_unit[i] # IIS
/]
⋮
```

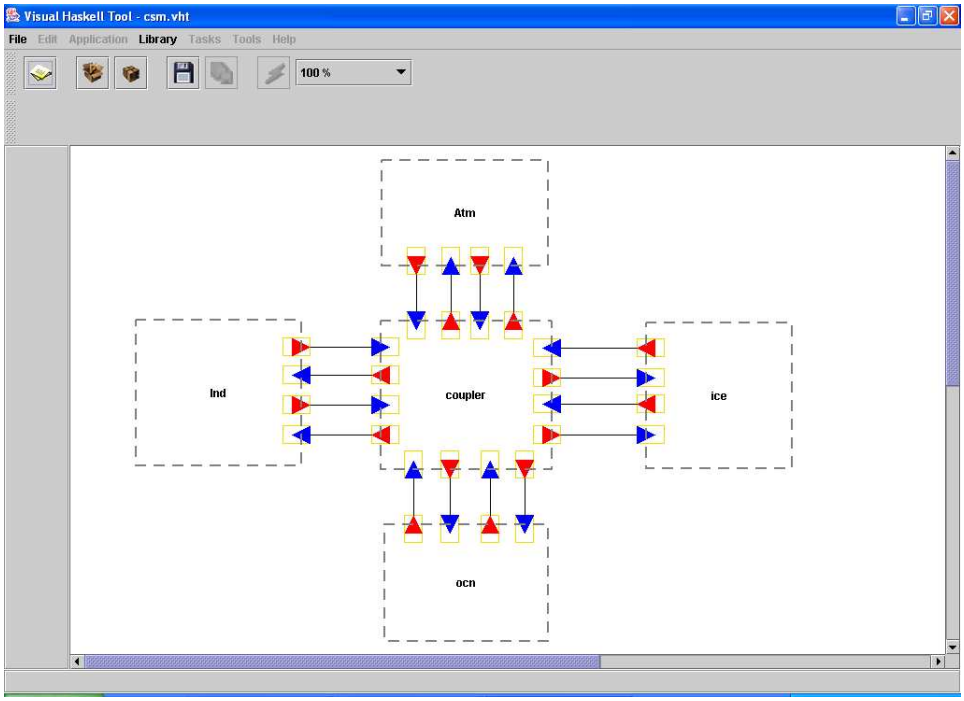


Figura 4.26. Visual # Tool

4.5.7 Instanciação das Unidades Virtuais

Às unidades virtuais resultantes do processo de sobreposição, são nomeadas as unidades não-virtuais anteriormente definidas, as quais caracterizam a funcionalidade descrita para as unidades. O código a seguir ilustra esta operação realizada no *kernel* IS:

```
⋮
[/ assign is_unit[i] to virtual_is_unit[i] /]
⋮
```

4.6 VISUAL # TOOL (VHT)

Como produto deste trabalho, foi desenvolvido o primeiro protótipo para uma ferramenta voltada ao desenvolvimento de programas paralelos baseada no modelo #. A ferramenta VHT (Visual # Tool) propõe-se a oferecer suporte ao desenvolvimento de programas paralelos de larga escala sobre *clusters*, sob as premissas do modelo #. Para isso, os construtores da linguagem de configuração # são suportados por meio de abstrações visuais, ao modo daquelas introduzidas nos diagramas apresentados nas figuras do capítulo 3. Entende-se que o uso de abstrações visuais, a qual é facilitada devido ao uso do modelo #, tende a facilitar a tarefa de desenvolvimento de programas, sobretudo para programadores pouco adaptados a esse estilo de programação, uma vez que ferramentas de programação visual têm sido bastante difundidas desde a última década. Na

Figura 4.5.7, é apresentada a tela principal do ambiente VHT, onde a configuração da aplicação CSM encontra-se configurada visualmente.

Sob influência e inspiração das vantagens advindas da modularidade fornecida pelo modelo #, VHT favorece e estimula o reuso de componentes em todas as fases do desenvolvimento de um programa paralelo. É fornecido ao programador o suporte à criação de bibliotecas de componentes, simples e compostos, comumente usados na construção de programas #, incluindo esqueletos e aspectos. É possível ainda a manipulação de bibliotecas de interfaces reusáveis.

VHT é integrada ao compilador #, de forma a permitir a compilação de programas para execução no ambiente LAM-MPI, o qual encontra-se integrado à VHT. É possível então a inicialização do ambiente LAM e disparo de programas # a partir do ambiente de desenvolvimento. A definição dos nós que compõem a arquitetura (arquivo *lamhosts*), bem como a especificação de esquemas de mapeamento dos processos que compõem o programa aos nós, são suportados por VHT. Pretende-se incluir ainda a integração à ferramentas de análise em tempo de execução de programas MPI, como XMPI, de modo que seja possível acompanhar a execução de programas.

O suporte à análise formal de propriedades e avaliação estática do desempenho de programas paralelos tem sido oferecido pela integração a ferramentas consagradas de análise de redes de Petri, como PEP [36] e INA [194], e avaliação de desempenho usando redes de Petri estocásticas, como TimeNET [234]. Com essa finalidade, compiladores capazes de traduzir configurações # para PNML (*Petri Net Markup Language*) [229] e SPNL (*Stochastic Petri Net Language*) [101] têm sido desenvolvidos, uma vez que estes formatos são suportados por muitas das ferramentas para análise de redes de Petri.

Motivado por requisitos de portabilidade, a linguagem Java foi escolhida para implementar-se VHT. Ainda pelas mesmas motivações, adotou-se a linguagem XML (*Extended Markup Language*) como formato de intercâmbio e armazenamento de especificações visuais de configurações #. O uso de XML tem facilitado a tradução de especificações visuais para a linguagem de configuração # e vice-versa.

CAPÍTULO 5

MODELANDO PROGRAMAS # COM REDES DE PETRI

O íntimo relacionamento com o formalismo de redes de Petri constitui uma das características que distingüem o modelo #, tornando possível a análise de propriedades e simulação de programas por meio do emprego deste formalismo. Redes de Petri constituem um importante formalismo surgido em meados da década de 60 para modelagem e análise de propriedades de sistemas concorrentes[188]. Desde então, o aprimoramento de seu poder expressivo tem sido perseguido, de forma a permitir ou facilitar sua aplicação sobre problemas em diversas áreas de aplicação, onde a necessidade de modelagem se faz presente. Sua disseminação, simplicidade e disponibilidade de uma extensa quantidade de ferramentas, acadêmicas e comerciais, para suporte à animação e prova de propriedades motivaram a escolha desse formalismo para tratar a parte concorrente de programas #. Originalmente, tal arcabouço foi usado nos trabalhos originais dentro do grupo de pesquisa onde surgiu esta tese, com a tradução dos construtores da linguagem OCCAM para redes de Petri[159]. O presente trabalho acrescenta aos trabalhos originais a modelagem de aspectos concernentes à transmissão de *lazy streams* e aos modos de comunicação suportados, baseados em MPI. Discute-se ainda como esqueletos podem ser usados para simplificar a rede de Petri gerada por um programa #, sendo os esqueletos MPI definidos no Capítulo 4 utilizados como estudo de caso. A notação funcional segundo a qual as regras de tradução encontram-se formalizadas serve como especificação para a construção de um compilador capaz de gerar descrições de redes de Petri em formatos padrão suportados por ferramentas de análise e avaliação de desempenho pré-existentes.

Em um programa #, é necessário modelar o *comportamento* individual de cada processo e a sincronização de operações de comunicação entre estes. O comportamento de um processo é definido como a ordem em que suas portas de entrada e saída são ativadas, especificada por meio de uma expressão regular controlada com semáforos, formalismo equivalente em poder expressivo às redes de Petri rotuladas (Seção 3.2.1).

Em [149], a tradução da primeira versão da linguagem Haskell$_{\#}$, a qual serviu como ponto de partida para a concepção do modelo #, encontra-se especificada. Entretanto, o poder expressivo suportado por esta versão da linguagem somente permitia a especificação de padrões muito simples de interação concorrente, restringindo a classe de programas paralelos que podiam ser programados por meio desta linguagem. O comportamento dos processos obedecia o padrão R-S, segundo [133], onde estes executavam a seguinte sequência de tarefas:

i) Leitura das portas de entrada, sequencialmente e na ordem em que se encontram declaradas no programa de configuração;

ii) Avaliação da função *main*, a qual recebe, através de seus argumentos, os valores lidos através das portas de entrada. Após a computação, um conjunto de valores produzidos são retornados em uma tupla;

iii) Transmissão dos valores produzidos devido a avaliação da função *main* através das portas de saída, ativadas em uma ordem sequencial e definida pela ordem de declaração destas no programa de configuração.

A avaliação da função *main* não poderia ser interrompida para efetivação de uma operação de comunicação. Assim, ao contrário do que ocorre em programas MPI, por exemplo, não era possível a intercalação entre computação e comunicação na execução paralela de processos. Processos repetitivos foram introduzidos com a finalidade de permitir a modelagem de programas reativos. Estes executam repetidamente a sequência de ações definida anteriormente, sem alcançar um estado de terminação, porém são incapazes de manter estado entre repetições. A natureza síncrona da comunicação não permitia que duas portas de um mesmo processo estivessem conectadas. Um processo intermediário (*buffer*) deveria ser introduzido com essa finalidade. Estudos posteriores concluíram que tais suposições e restrições não eram realistas para expressão de grande parte dos programas paralelos de interesse em computação científica de alto desempenho.

As limitações expressivas detectadas na versão anterior de Haskell$_{\#}$ motivaram o interesse em estudar maneiras de superá-las sem comprometimento significativo em termos de eficiência, hierarquia de processos e capacidade de raciocínio sobre programas, o que conduziu ao desenvolvimento do modelo #. A introdução de *lazy* streams[49] tornou possível expressar padrões universais de comportamento de processos, possibilitando a intercalação entre comunicação e computação sem comprometimento da hierarquia de processos. Ao contrário, novos mecanismos de abstração introduzidos permitiram o fortalecimento desta propriedade, tornando o programador de módulos funcionais mais livre de suposições a respeito da rede de processos da aplicação [50].

A primeira tradução para redes de Petri de uma versão de Haskell$_{\#}$ com suporte a *lazy streams* foi apresentada em [52]. Expressões regulares embutidas no código HCL (*Haskell$_{\#}$ Configuration Language*) foram empregadas para expressar o comportamento de processos, podendo facilmente serem traduzidas para redes de Petri. Nenhuma restrição mostrou-se necessária para a programação de módulos funcionais no nível de computação. É considerada responsabilidade do programador garantir que o código dos módulos funcionais seja compatível com o padrão regular de ativação das portas de entrada e saída, em nível de coordenação, para os processos associados a estes. Entretanto, algumas questões emergiram ao avaliar-se esta nova abordagem:

- É decidível inferir o comportamento de um processo, em nível de coordenação, pela análise do módulo funcional que implementa a demanda pelos valores de suas portas de entrada e saída em nível de computação ?

- É possível traduzir para redes de Petri quaisquer comportamentos que poderiam ser descritos por meio de um programa Haskell ? Se a resposta for sim, seria possível provar que um compilador poderia ser hábil para analisar o módulo funcional e automaticamente gerar uma rede de Petri que descreva o seu comportamento ?

Estas questões expressam uma preocupação em acoplar aquilo que é descrito nos níveis de computação e coordenação com respeito ao comportamento de processos, libertando os programadores de sua preocupação em garantir que aquilo que foi descrito, sobre um processo, em nível de coordenação (interface) é compatível com o que foi descrito a nível de computação (módulo funcional).

O comportamento de processos foi então formalizado segundo o arcabouço provido pela teoria de linguagens formais[117], uma vez que sequências de ativação de portas em processos descrevem linguagens formais. Devido ao fato de ser Haskell um motor de computação universal, comparável ao $\lambda - calculus$, o comportamento de um processo poderia descrever uma linguagem recursivamente enumerável qualquer. Entretanto, não é decidível verificar se uma dada linguagem recursivamente enumerável é regular. Além disso, redes de Petri não são modelos computacionais universais, não sendo portanto capazes de expressar quaisquer linguagens recursivamente enumeráveis. Redes de Petri rotuladas são capazes de expressar um sub-conjunto próprio das linguagens sensíveis ao contexto[103], diferente do conjunto das linguagens livres do contexto. Portanto, redes de Petri são capazes de expressar uma classe mais rica de padrões de iteração em relação às expressões regulares empregadas até então para expressar comportamentos de processos em Haskell$_{\#}$. A despeito do fato de que padrões cíclicos e regulares podem ser considerados suposições razoáveis para a maioria dos programas paralelos [177, 28, 168, 186], decidiu-se extender a habilidade de Haskell$_{\#}$ em expressar padrões de iteração, alcançando o poder expressivo de redes de Petri rotuladas.

Uma classe de formalismos denominados *expressões concorrentes sincronizadas*, extensão às expressões regulares proposta no início da década de 80 para expressar mecanismos universais de concorrência e sincronização[126], mostraram-se de especial interesse neste trabalho. São conhecidos resultados que estabelecem a equivalência entre *expressões regulares sincronizadas com semáforos*, uma sub-classe das expressões concorrentes, com redes de Petri rotuladas[126].

5.1 NOTAÇÕES, DEFINIÇÕES, E RESULTADOS IMPORTANTES

As próximas seções introduzem as notações, definições e resultados usados ao longo do restante deste capítulo.

5.1.1 Linguagens Formais

O conceito de linguagens formais[117] será usado como arcabouço formal para investigação da expressividade do modelo # em relação aos padrões universais de interação entre processos, suportados pelos meios convencionais de programação via passagem de mensagens. A seguir, são introduzidas as principais definições concernentes:

Definição 5.1 (Alfabeto) *Um alfabeto é qualquer conjunto finito de símbolos indivisíveis, denotados por* Σ.

Definição 5.2 (Palavra) *Uma palavra é uma sequência finita de símbolos pertencentes a um certo alfabeto* Σ. *O símbolo* ϵ *é usado para denotar a palavra vazia, de tamanho 0.*

Definição 5.3 (Fecho de um Alfabeto) *Seja Σ uma alfabeto. O fecho do alfabeto Σ, denotado por Σ^*, é definido como a seguir:*

$$\Sigma^* = \{w \mid \ w \text{ é uma palavra no alfabeto } \Sigma\}$$

Portanto, qualquer palavra no alfabeto Σ, incluindo ϵ, pertence ao conjunto Σ^*. É ainda comum definir o conjunto Σ^+ como:

$$\Sigma^+ = \Sigma^* - \{\epsilon\}$$

Definição 5.4 *Seja Σ um alfabeto. Uma linguagem formal L sobre Σ é um conjunto de palavras tais que:*

$$L \subset \Sigma^*$$

Linguagens formais podem ser usadas para modelar a ordem de ativação de portas de um processo # (comportamento), de forma a estudarmos o poder expressivo do modelo # em relação ao formalismo de redes de Petri. Nesta abordagem, para cada porta é associado um símbolo do alfabeto Σ. Dessa forma, uma sequência de ativação de portas representa assim uma linguagem formal em Σ.

5.1.2 Redes de Petri

Esta seção apresenta os conceitos básicos de redes de Petri, baseadas em [159].

Definição 5.5 (Redes de Petri Lugar/Transição) *Uma rede de Petri lugar/transição pode ser formalizada como uma quádrupla (P, T, A, M_0), onde:*

i) P é um conjunto finito de símbolos, os quais pode armazenar um número ilimitado de marcas;

ii) T é um conjunto finito de transições.

iii) $P \cap T = \emptyset$.

iv) A define um conjunto de arcos, de tal forma que $A \subseteq (P \times T \cup T \times P) \times Naturais$. Isso indica que um arco pode conectar uma transição a um lugar ou um lugar a uma transição, mas nunca dois lugares ou duas transições. O número mapeado ao arco indica sua valoração, ou peso. Por simplicidade, assume-se que se o peso de uma arco é omitido, seu valor é 1. ($(p, t) \equiv (p, t, 1)$);

v) A relação $M_0 \subset P \times N$ define a marcação inicial *da rede, o qual define o número de marcas que estão armazenadas em cada lugar na sua configuração inicial;*

A partir da marcação inicial, uma rede de Petri define um conjunto de marcações alcançáveis. Uma marcação é alcançável se pode ser obtida a partir da marcação inicial pelo disparo de alguma sequência de transições, de acordo com as regras de disparo. As definições abaixo formalizam tais conceitos:

Definição 5.6 (Transição Habilitada) *Sejam $PN = (P, T, A, M_O)$ uma rede de Petri e t, $t \in T$, uma transição de PN:*

$$t \text{ está habilitada} \Leftrightarrow \forall p \in P : (p, t, m) \in A \wedge (p, n) \in M : n \geq m$$

A definição acima indica que uma transição t está habilitada se o número de marcas em cada um de seus lugares de entrada é maior ou igual que o peso do respectivo arco que o liga a transição.

Definição 5.7 (Regra de Disparo em Marcações Alcançáveis) *Seja $PN = (P, T, A, M_0)$ uma rede de Petri e M uma marcação de PN. A partir de uma marcação M, o disparo de uma transição $t \in T$ produz uma nova marcação M', definida como se segue:*

$$M \xrightarrow{t} M' \Rightarrow t \text{ está habilitada}$$
$$\text{onde:}$$
$$M' = \bar{M} \cup M_I \cup M_O$$
$$M_I = \{(p, n - m) \mid p \in P \wedge (p, t, m) \in A \wedge (p, n) \in M\}$$
$$M_O = \{(p, n + m) \mid p \in P \wedge (t, p, m) \in A \wedge (p, n) \in M\}$$
$$\bar{M} = \{(p, n) \mid p \in P \wedge (p, n) \in M \wedge (p, t, m_1) \notin A \wedge (t, p, m_2) \notin A\}$$

A sub-marcação M_I indica a nova marcação para os lugares de entrada de t, enquanto M_O indica a nova marcação para seus lugares de saída. A transição pode ser disparada quando habilitada e o efeito da transição é a remoção de marcas do lugares de entrada de t e acréscimo de marcas nos lugares de saída de t, de acordo com a valoração dos arcos que os ligam a t.

É necessário generalizar esta definição de forma a suportar-se o conceito de marcação alcançável a partir do disparo de uma sequência de transições. Uma marcação M_n é alcançável a partir de uma marcação M se existe uma sequência de disparos a partir da marcação M a qual resulta na marcação M_n.

$$M \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} M_n$$

Esta notação pode ser abreviada como se segue:

$$M \xrightarrow{\sigma} M_n, \text{ onde } \sigma = t_1 t_2 \cdots t_n$$

O símbolo σ denota uma sequência de disparos de transições.

5.1.2.1 Redes de Petri rotuladas

Rede de Petri rotuladas extendem redes de Petri lugar/transição de forma a que estas sejam capazes de gerar classes de linguagens formais.

Definição 5.8 (Redes de Petri rotuladas) *Seja Σ um alfabeto. Uma rede de Petri rotulada é uma 6-upla $(P, T, A, M_0, \Sigma, \rho)$, onde (P, T, A, M_0) é uma rede de Petri lugar/transição e $\rho : T \rightarrow \Sigma \cup \{\lambda\}$ é uma função que associa transições a um símbolo pertencente ao conjunto Σ. Transições rotuladas com o símbolo especial λ são denominadas* transições silenciosas*, cujo disparo não afeta a linguagem gerada pela rede de Petri.*

Denotaremos por $\Re$ o conjunto que inclui todas as redes de Petri rotuladas, como definidas em 5.8. Redes de Petri rotuladas induzem duas classes de linguagens formais, formalizadas a seguir.

Definição 5.9 (Linguagem Gerada por rede de Petri) *Seja $N = (P, T, A, M_0, \rho)$ uma rede de Petri rotulada. A linguagem gerada por N, $L(N, M_0)$, é definida da seguinte forma:*

$$L(N, M_0) = \{\rho(\sigma) \mid M_0 \xrightarrow{\sigma} M, \text{ para alguma marcação alcançável } M\}.$$

Definição 5.10 (Linguagem Terminal Gerada por uma rede de Petri) *Seja uma rede de Petri $N = (P, T, A, M_0, \rho)$ e uma marcação alcançável M_f, denominada* marcação terminal. *A linguagem terminal definida por N, com respeito a M_f, $T(N, M_0, M_f)$, define-se da seguinte forma:*

$$T(N, M_0, M_f) = \{\rho(\sigma) \mid M_0 \xrightarrow{\sigma} M_f\}$$

Em resumo, seja uma rede de Petri rotulada N, a linguagem gerada por N é definida a partir de todas as sequências possíveis de disparos de transições a partir de sua marcação inicial. A linguagem terminal gerada por PN difere de PN por considerar somente aquelas sequências de disparos capazes de alcançar a marcação M_f.

Definição 5.11 (Classes de Linguagens de Redes de Petri) *Denotamos a classe de todas as linguagens de redes de Petri como ${\ell_\lambda}^0$, e a classe de todas as linguagens terminais de redes de Petri como ${\ell_\lambda}^1$. Sabe-se que ${\ell_\lambda}^0 \subset {\ell_\lambda}^1$.*

5.1.3 Expressões Regulares

Expressões regulares[141] constituem um simples e conhecido formalismo usado para expressar linguagens regulares. Estas formam a classe mais simples de linguagens formais, com aplicações importantes em ciência da computação, capazes de serem reconhecidas por meio de autômatos finitos.

Definição 5.12 (Expressão Regular) *Uma expressão regular E sobre um alfabeto Σ é indutivamente definido da seguinte forma:*

i) $a \in \Sigma$ *é uma expressão regular;*

ii) ϵ *é uma expressão regular;*

iii) $\emptyset$ *é uma expressão regular;*

iv) Sejam S_1 e S_2 expressões regulares, então (S_1), $S_1 \cdot S_2$, $S_1 + S_2$ e, ${S_1}^*$ são expressões regulares;

v) Somente expressões da forma definida em i, ii, iii e iv constituem expressões regulares.

Definição 5.13 (Linguagem Regular) *Seja E uma expressão regular sobre o alfabeto Σ. A linguagem formal gerada por E,$L_{\mathsf{RE}}(E)$, é definida da seguinte forma:*

i) $L_{\mathsf{RE}}(\emptyset) = \emptyset$

ii) $L_{\mathsf{RE}}(\lambda) = \{\lambda\}$

iii) $L_{\mathsf{RE}}(a) = \{a\},\ para\ a \in \Sigma,$

iv) $L_{\mathsf{RE}}((E)) = L_{\mathsf{RE}}(E)$

v) $L_{\mathsf{RE}}(E_1.E_2) = \{xy \mid x \in L_{\mathsf{RE}}(E_1) \wedge y \in L_{\mathsf{RE}}(E_2)\}$

vi) $L_{\mathsf{RE}}(E_1 + E_2) = \{x \mid x \in L_{\mathsf{RE}}(E_1) \vee x \in L_{\mathsf{RE}}(S_2)\}$

vii) $L_{\mathsf{RE}}(E^*) = \bigcup_{i>0} L_{\mathsf{RE}}(E_i)$

onde:

i) $E^0 = \lambda;$

ii) $E^i = E^{i-1}E, i > 0$

A classe das linguagens regulares é denotada por RE.

A tradução anterior de Haskell$_\#$ para redes de Petri[52] assumia que o comportamento dos processos era expresso por meio de expressões regulares. Contudo, redes de Petri são mais expressivas que expressões regulares do ponto de vista da geração de linguagens formais[223, 233].

5.1.4 Expressões concorrentes

Investigamos como acrescentar poder expressivo à linguagem # de forma a que esta pudesse expressar quaisquer padrões de comportamento que pudessem ser representados com redes de Petri, incluindo aqueles não-regulares. Para isso, foi estudada a teoria desenvolvida nos anos 80 sobre expressões concorrentes[126], uma generalização de expressões regulares concebida com a intenção de expressar padrões universais de concorrência e sincronização. Expressões concorrentes são formalizadas a seguir.

Definição 5.14 (Expressões Concorrentes) *Seja Σ um alfabeto. Uma expressão concorrente S sobre o alfabeto Σ é definida indutivamente como se segue:*

i) Se S é uma expressão regular sobre Σ, então S é uma expressão concorrente;

ii) Se S_1 e S_2 são expressões concorrentes, então $S_1 \odot S_2$, e ${S_1}^{\otimes}$ são expressões concorrentes;

A presença dos operadores $\odot$ e $\otimes$ diferencia sintaticamente expressões concorrentes de expressões regulares (intercalação[1] e seu fecho, respectivamente). A linguagem formal gerada por expressões concorrentes é definida a seguir.

Definição 5.15 (Linguagem Gerada por uma Expressão Concorrente) *Seja S uma expressão concorrente sobre Σ. A linguagem formal gerada por S, $L_{CE}(S)$, é definida de acordo com as seguintes regras:*

i) $L_{\mathsf{CE}}(S) = L_{\mathsf{RE}}(S)$, *se S é uma expressão regular bem formada;*

ii) $L_{\mathsf{CE}}(S_1 \odot S_2) = \{x_1y_1x_2y_2\cdots x_ky_k \mid x_1x_2\cdots x_k \in L_{\mathsf{CE}}(S_1) \wedge y_1y_2\cdots y_k \in L_{\mathsf{CE}}(S_2)\}$,

iii) $L_{\mathsf{CE}}(S^{\otimes}) = \bigcup L_{\mathsf{CE}}(S^{\odot i})$.

onde:

i) $S^{\odot 0} = \lambda$

ii) $S^{\odot i} = S^{\odot i-1} \odot S, i > 0$.

A classe das expressões concorrentes é denotada por CE. Claramente, $\mathsf{CE} \supset \mathsf{RE}$. Entretanto, existe um resultado mais forte e importante que relaciona linguagens concorrentes e regulares.

Teorema 5.1 ([200], Relação Entre Expressões Concorrentes e Regulares) *Se S é uma expressão concorrente que não faz uso do operador $\otimes$, então $L_{\mathsf{CE}}(S)$ é regular.*

Outro importante resultado para os propósitos deste trabalho caracteriza os limites de expressividade das expressões concorrentes.

Teorema 5.2 ([103], Limite Expressivo das Expressões Concorrentes) *Se S é uma expressão concorrente, então $L_{\mathsf{CE}}(S)$ é uma linguagem sensível ao contexto.*

Com o objetivo de aumentar o poder expressivo de expressões concorrentes, permitindo a estas expressar linguagens recursivamente enumeráveis, introduziu-se o conceito de *expressões concorrentes sincronizadas*[126]. Estas extendem às expressões concorrentes com mecanismos de sincronização, tendo sido empregada na década de 80 para analisar a expressividade de mecanismos de sincronização concorrentes, especialmente aqueles baseados em semáforos. A seguir formalizamos este conceito.

Definição 5.16 (Expressões Concorrentes Sincronizadas) *Seja Σ um alfabeto e Ω um conjunto finito de símbolos que denotam primitivas de sincronização, tal que Σ e Ω constituem conjuntos disjuntos. Uma expressão concorrente E sobre $\Sigma \cup \Omega$ é chamada expressão concorrente sincronizada e a linguagem ($L_S(\Omega)$) é dita ser um mecanismo de sincronização sobre E. Uma expressão concorrente sincronizada sobre $\Sigma \cup \Omega$, adotando o mecanismo de sincronização $\boldsymbol{K}$, onde $\boldsymbol{K}$= $L_S(\Omega)$, será denotada por $(E, \Sigma, \Omega, \mathbf{K})$.*

[1] operador *shuffle* ou *interleaving*, na língua inglesa.

A classe de expressões concorrentes sincronizadas é denotada por SCE. A próxima definição formaliza a linguagem gerada por uma expressão concorrente sincronizada, a qual fornece significado a um mecanismo de sincronização.

Definição 5.17 (Linguagem Gerada por Expressão Concorrente Sincronizada) *Seja* $(S, \Sigma, \Omega, \boldsymbol{K})$ *uma expressão concorrente sincronizada, onde* $\boldsymbol{K} = L_S(\Omega)$*. A linguagem gerada por* S*,* $L_{SCE}(S)$*, é definida como se segue:*

$$L_{SCE}(S) = \{h(x) \mid x \in L_{CE}(x), \overline{h}(x) \in \mathbf{K}\},$$

onde os homomorfismos h *and* $\overline{h}$ *são definidos da seguinte forma:*

$$h(a) = \begin{cases} a & a \in \Sigma \\ \lambda & a \in \Omega \end{cases}$$

$$\overline{h}(a) = \begin{cases} \lambda & a \in \Sigma \\ a & a \in \Omega \end{cases}$$

A classe das linguagens concorrentes sincronizadas que usam um mecanismo de sincronização $\mathbf{K}$ é denotada por SCE^K.

Esta definição parametriza o mecanismo de sincronização adotado. Os mais comuns são aqueles baseados em semáforos, cujos mais significativos exemplares são apresentados a seguir.

Seja Ω_n um conjunto de primitivas de sincronização de cardinalidade n, onde:

$$\Omega_n = \{\omega_i, \sigma_i \mid 1 \leq i \leq n\}$$

Os seguintes mecanismos de sincronização baseados em semáforos pode ser definidos sobre Ω_n[126].

i) Semáforos Contadores: $L_S(\Omega_n) = C(n)$, onde:
$C(n) = L_{\mathsf{CE}}((\sigma_1 \cdot \omega_1 + \sigma_1))^{\otimes} \odot (\sigma_2 \cdot \omega_2 + sigma_2))^{\otimes} \odot \cdots \odot (\sigma_n \cdot \omega_n + \sigma_n))^{\otimes})$
$\mathbf{C} = \{C(n) \mid n \geq 0, ondeC(0) = \lambda\}$

ii) Semáforos [0]-contador: $L_S(\Omega_n) = C_0(n)$, onde:
$C_0(n) = L_{\mathsf{CE}}((\sigma_1 \cdot \omega_1))^{\otimes} \odot (\sigma_2 \cdot \omega_2))^{\otimes} \odot \cdots \odot (\sigma_n \cdot \omega_n))^{\otimes})$
$\mathbf{C}_0 = \{C_0(n) \mid n \geq 0, ondeC_0(0) = \lambda\}$

iii) Semáforo Binário: $L_S(\Omega_n) = B(n)$, onde:
$B(n) = L_{\mathsf{CE}}((\sigma_1 \cdot \omega_1 + \sigma_1))^{*} \odot (\sigma_2 \cdot \omega_2 + sigma_2))^{*} \odot \cdots \odot (\sigma_n \cdot \omega_n + \sigma_n))^{*})$
$\mathbf{B} = \{B(n) \mid n \geq 0, ondeB(0) = \lambda\}$

iv) Semáforo [0]-binário: $L_S(\Omega_n) = B_0(n)$, onde:
$B_0(n) = L_{\mathsf{CE}}((\sigma_1 \cdot \omega_1))^* \odot (\sigma_2 \cdot \omega_2))^* \odot \cdots \odot (\sigma_n \cdot \omega_n))^*)$
$\mathbf{B_0} = \{B_0(n) \mid n \geq 0, onde B_0(0) = \lambda\}$

$\mathbf{C}$, $\mathbf{C_0}$, $\mathbf{B}$ e $\mathbf{B_0}$ denotam os conjuntos de primitivas de sincronização com qualquer quantidade de primitivas.

Esta notação pode ser generalizada para qualquer mecanismo de sincronização que empregue semáforos. Seja $\mathbf{CE}(n)$ a família de mecanismos de sincronização com n primitivas. Então, $\mathbf{CE} = \bigcup_{n=0}^{n} \mathbf{CE}(i)$. Dessa forma, $\mathbf{C}(n)$, $\mathbf{C_0}(n)$, $\mathbf{B}(n)$, $\mathbf{B_0}(n)$ constituem casos particulares de $\mathbf{CE}(2n)$. Uma expressão concorrente controladea por um sistema baseado em semáforos é dito uma *expressão concorrente sincronizada por semáforo*.

Muitos resultados importantes a respeito do poder expressivo das expressões concorrentes sincronizadas foram apresentados[126]. Neste trabalho, merecem especial atenção aqueles que comparam o poder expressivo deste formalismo ao formalismo de redes de Petri. Um destes estabelece a equivalência entre redes de Petri rotuladas terminais como *expressões regulares sincronizadas por semáforos do tipo [0]-contador*.

Definição 5.18 (Expressões Regulares Sincronizadas) *Expressões regulares sincronizadas podem ser entendidas como expressões concorrentes sincronizadas que não fazem uso do operador* $\otimes$.

A classe das expressões regulares sincronizadas é denotada por SRE. A classe das expressões regulares sincronizadas com um certo mecanismo $\mathbf{S}$ é denotada por $\mathsf{SRE_S}$

Teorema 5.3 (Expressões Regulares Sincronizadas e Redes de Petri) *A classe das linguagens geradas por expressões regulares sincronizadas por semáforos 0-contador é igual a* $\ell_\lambda{}^1$.

O Teorema 5.3 é de fundamental importância para o propósito de tornar o poder expressivo da linguagem # equivalente ao poder expressivo do formalismo de redes de Petri. O uso de expressões regulares sincronizadas evita o uso do operador $\otimes$, o qual não possui interpretação semântica na linguagem #. Shaw[200], ao introduzir o conceito de expressões de fluxo, formalismo precursor das expressões concorrentes, forneceu duas interpretações para o operador $\otimes$: um construtor para iterações paralelas ou a chamada *ad hoc* à primitiva (*fork*) no contexto de uma iteração com número não conhecido de repetições. Nem uma destas possui equivalente prático no modelo #.

Dessa forma, é necessário somente a introdução das primitivas de controle de semáforos (*wait* e *signal*), bem como do operador $\odot$, o qual expressa a execução concorrente de ações. De acordo com os Teoremas 5.1 and 5.3, entretanto, somente o sistema de semáforos é suficiente para levar o poder expressivo de expressões regulares ao patamar expressivo das redes de Petri.

O uso de semáforos [0]-contador impõe a seguinte restrição a execução de processos em um programa #: o valor de um semáforo no início e no final deve ser o mesmo. Entretanto, é indecidível verificar esta restrição em uma expressão arbitrária.

A seguir é introduzida a classe das expressões de fluxo, introduzidas por Shaw em 1978, e algums resultados sobre a equivalência de algumas sub-classes de redes de Petri induzidas por este formalismo.

5.1.4.1 Expressões de Fluxo Anteriormente ao aparecimento das expressões concorrentes no início da década de 80, em 1978, Shaws introduziu o conceito de *flow expressions*, uma tentativa de definir um formalismo para especificação e análise de propriedades de sistemas. Essencialmente, expressões de fluxo são expressões concorrentes sincronizadas da forma $(E, \Sigma, \Omega, \mathbf{B})$, enriquecidas com um mecanismo de sincronização de exclusão mútua e o operador ∞ operator (repetição infinita), capaz de modelar sistemas reativos. Este operador não aparece em expressões concorrentes, uma vez que a teoria de linguagens formais, sobre a qual este formalismo adere, não trata de sequência infinitas de símbolos. Os operadores de exclusão mútua pode ser facilmente simulados usando semáforos. Dessa forma, podem ser omitidos sem prejuízo ao poder descritivo das expressões de fluxo. De acordo com as considerações acima, expressões concorrentes sincronizadas da forma $(E, \Sigma, \Omega, \mathbf{B})$ serão referidas como *finite flow expressions*.

No contexto deste trabalho, o resultado mais importante concernente à teoria das expressões de fluxo relaciona seu poder expressivo ao poder expressivo de redes de Petri rotuladas terminais. Este resultado será formalizado nos próximos parágrafos.

Definição 5.19 *Seja FE^r a classe das expressões de fluxo que não contém uma das seguintes formas: $(S_1 S_2{}^{\otimes} S_3)^*$ e $(S_1 S_2{}^{\otimes} S_3)^{\otimes}$.*

Definição 5.20 *Seja FE^n a classe das expressões de fluxo que não contém uma das seguintes formas: $(S_1 S_2{}^* S_3)^*$, $(S_1 S_2{}^* S_3)^{\otimes}$, $(S_1 S_2{}^{\otimes} S_3)^*$ e $(S_1 S_2{}^{\otimes} S_3)^{\otimes}$.*

As classe de linguagens de fluxo geradas por expressões em FE^r and FE^n são, respectivamente, FE^r e FE^n.

Teorema 5.4 ([7]) $\mathsf{FE}^r = \mathsf{FE}^n = \ell_\lambda{}^1$

Em [7], é mostrado como expressões de fluxo em FE^r e FE^n podem ser traduzidas para redes de Petri e vice-versa.

A referência [200] mostra a correspondência entre expressões de fluxo e construtores práticos empregados comumente em linguagens de programação sob diversos paradigmas.

5.1.5 Processos # e Expressões Regulares Sincronizadas por Semáforos

Como discutido anteriormente, o comportamento de um processo #, sob o ponto de vista do meio de coordenação, é descrito pela ordem em que as portas de comunicação de sua interface são ativadas. A ativação de uma porta causa a realização de uma operação de comunicação sobre esta, de acordo com a semântica do modo de comunicação do canal conectado à porta. A ordem de ativação de portas pode ser modelada por meio de uma linguagem formal, sendo seu alfabeto constituído de rótulos associados a cada uma das portas. Por exemplo, considere um processo $MERGE$ com duas portas de entrada, rotuladas in_1 e in_2, e uma porta de saída, rotulada out. A partir de in_1 e in_2,

o processo recebe duas listas de valores de mesmo tamanho, as quais são intercaladas e enviadas pela porta *out*. Um valor marcador (EOS) deve ser enviado para marcar o final da *stream*. Usando uma expressão regular, podemos descrever seu comportamento da seguinte forma:

$$in_1 \cdot in_2 \cdot ((out \cdot (in_1 + in_2)))^* \cdot out$$

A linguagem # usa este tipo de abordagem para descrever o comportamento de processos em nível de coordenação. A despeito do fato de que para programas paralelos em geral, comportamentos regulares de comunicação são uma suposição razoável[168, 186], um formalismo mais expressivo, equivalente às redes de Petri, pode ser usado em alternativa às expressões regulares. Expressões regulares sincronizadas por semáforos do tipo 0-contador ($SRE_{C[0]}$'s) são então empregadas para este propósito no modelo #. Estas são equivalentes às redes de Petri rotuladas em poder expressivo (Teorema 5.3).

Uma linguagem para descrição de comportamento de processos foi embutida na linguagem de configuração #, como mostrado na Seção 3.2.1. Como discutido, seus combinadores são equivalentes aqueles suportados por expressões regulares sincronizadas por semáforos, inspirando-se nos combinadores da linguagem OCCAM[124]. Sendo assim, é possível traduzir a especificação do comportamento de um processo # em uma rede de Petri equivalente, o que será demonstrado na próxima seção.

5.2 ESQUEMA DE TRADUÇÃO

Esta seção formaliza um esquema de tradução de programas # para redes de Petri rotuladas.

5.2.1 Notação Empregada

A linguagem cuja sintaxe é apresentada abaixo, denominada *abstract #*, captura somente as informações consideradas essenciais ao procedimento de tradução de um programa # para redes de Petri, sendo empregada adiante para formalização deste esquema de tradução. É fácil perceber que os construtores de *abstract #* possuem correspondência trivial aos construtores da linguagem #:

<Component> → **component** ({u_1:<Unit>, . . ., u_k:<unit>},
{c_1:<Channel>, . . .,c_k:<Channel>})
{i_1:<Interface_Point>, . . .,i_k:<Interface_Point>}

<Interface_Point> → **interface_point** (id, pid^{port}, <IPoint_type>)

<IPoint_type> → **entry** | **exit**

<Unit> → **unit** (id, {p_1:<Port>, . . ., p_k:<Port>}, <Type$_u$>, <UnitKind>)

<UnitKind> → **process** c:<Behavior> | **cluster** (c:<Component>, {(id_1, id_1^{port}), . . ., (id_k, id_k^{port})})

<Type$_u$> → **repetitive** | **non-repetitive**

<Behavior> → **behavior** ({$id_1^{sem}, \ldots, id_n^{sem}$}, <Action>)

<Action> → **skip**
| **seq** {a_1:<Action>; . . .; a_k:<Action>}
| **par** {a_1:<Action>; . . .; a_k:<Action>}
| **alt** {g_1:<Action>; . . .; g_k:<Action>}
| **repeat_until** (<Action>, C)
| **repeat_counter** (<Action>, N)
| **repeat_forever** <Action>

| **signal** id
| **wait** id
| **activate** id

<PORT> → **port** (id,<DIRECTION>,<MULTIPLICITY>,<TYPE$_p$>,$nesting\_factor$)

<MULTIPLICITY> → **single** | **group** (<TYPE$_g$>, {p_1:<PORT>, ..., p_n:<PORT>})

<DIRECTION> → **input** | **output**

<TYPE$_p$> → **stream** | **non-stream**

<TYPE$_g$> → **any** | **all**

<CHANNEL> → **connect** (id^{port}, id^{port}, <CHANMODE>)

<CHANMODE> → **synchronous** | **buffered** | **ready**

A notação funcional empregada na formalização do processo de tradução facilita sua implementação em uma linguagem funcional. Por esse motivo, a notação empregada na definição de *abstract #* inspira-se na definição de tipos algébricos em programas escritos na linguagem Haskell. Os identificadores em negrito são construtores, possivelmente seguidos por uma tupla que armazena as informações sobre o objeto o qual descreve. Assume-se que as portas das unidades possuem identificadores distintos.

O esquema de tradução é definido indutivamente sobre a estrutura sintática de *abstract #*, gerando uma rede de Petri segundo a representação relacional descrita na Definição 5.5. Notações usuais em teoria dos conjuntos, como compreensão de conjuntos, são empregadas extensivamente.

5.2.1.1 Qualificadores: A noção de *qualificador* é introduzida com a finalidade de facilitar a distinção em uma rede de Petri de sub-redes que se encontram sobrepostas. Esse artifício permite modularizar a descrição de funcionalidades presentes na rede resultante que se encontram entrelaçadas na sua descrição, sendo assim útil nas definições adiante. A cada elemento da rede (lugar ou transição) é associado um *qualificador*, o qual é recuperado por meio da função $qualifier$. Um qualificador é indutivamente definido da seguinte forma:

1. a é um *elemento qualificador primitivo*;
2. Se $a_1, \ldots, a_n$ são *elementos qualificadores primitivos*, então $(a_1, \ldots, a_n)$ é um *elemento qualificador composto*;
3. Se $t_1, \ldots, t_n$ são *elementos qualificadores*, primitivos ou compostos, então $\{t_1, \ldots, t_n\}$ é um *qualificador*;
4. Nada além do que pode ser formado a partir das regras 1, 2 e 3 constitui um qualificador.

Dessa forma, um qualificador é definido como uma lista de *elementos qualificadores*, simples ou compostos. Por simplicidade, uma lista com um único elemento é representado sem os delimitadores de lista ({, }). Não há uma restrição sobre o que pode ser usado como elemento qualificador primitivo. Nas definições adiante, por simplicidade, o qualificador de um elemento da rede de Petri (lugar ou transição) é único quando não explicitamente definido. O qualificador especial **unique** pode ser aplicado sempre que seja necessário definir um qualificador único explicitamente. A definição a seguir ensina

como qualificadores podem ser usados para identificar, em uma rede de Petri, elementos (lugares ou transições) equivalentes.

Definição 5.21 (Identificação de Vértices na Rede de Petri Resultante) *Seja E o conjunto de vértices (lugares e transições) de uma rede de Petri. Considere um subconjunto arbitrário $\overline{E}$ ($\overline{E} \subset E$) de vértices de mesma natureza (todos são lugares ou todos são transições). Assume-se que o qualificador de um vértice $e \in E$ é definido pela função de qualificação $qualifier(e)$. Define-se que:*

$$\overline{E} \textit{constituem os mesmos vértices} \Leftrightarrow \exists e \in \overline{E}\ (\forall \overline{e} \in \overline{E}, qualifier(e) \cap qualifier(\overline{e}) \neq \emptyset)$$

Pela Definição 5.21, vértices que possuem o mesmo qualificador, ou cuja interseção dos vértices qualificadores não é vazio, constituem o mesmo vértice da rede e podem ser unificados na rede de Petri resultante. Adicionalmente, mesmo que dois vértices possuam qualificadores distintos ou não possuam vértices qualificadores em comum, se existe um terceiro qualificador que satisfaz essa propriedade com relação a ambos, simultaneamente, então os três vértices devem ser unificados.

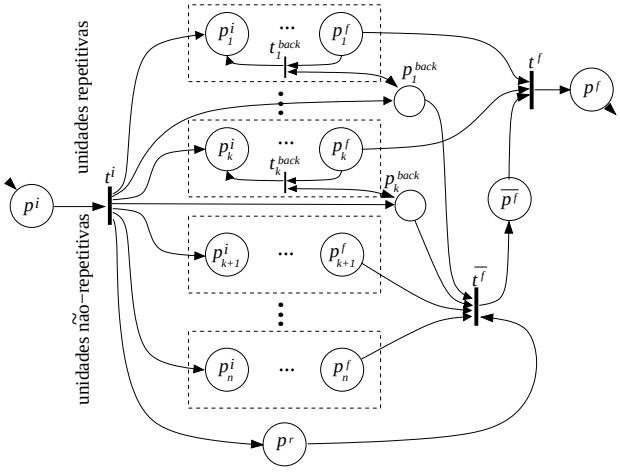


Figura 5.1. Traduzindo um Programa # (Regra 5.1)

5.2.2 Traduzindo um Programa # para Redes de Petri

Denotaremos por $\Pi_{\#}$ o conjunto de todos programas que podem ser descritos por *abstract #*. A função $\Upsilon^{\#}$, total e injetora, traduz um programa # para sua representação equivalente em redes de Petri:

$$\Upsilon^{\#} : \Pi_{\#} \rightarrow \Re$$

Na descrição que se segue, a palavra processo é usada como sinônimo de unidade. O tradutor de programas # para redes de Petri desenvolvido neste trabalho atua sobre um

componente, gerando a rede de Petri que modela o seu comportamento. Caso alguma unidade que constitui o componente seja um aglomerado, recursivamente são geradas as redes de Petri que descrevem os comportamentos dos componentes associados a essas unidades.

$$\Upsilon^{\#}(\textbf{component}\ (Units, Channels, InterfacePoints)) = PN_3$$

onde:

$$PN_1 = fold\ (\Upsilon^{channel}, Channels, (P, T, A^+ \cup A^r, \rho, p^i, p^f))$$

$$PN_2 = fold\ (\lambda(\pi, PN_k) \rightarrow \Upsilon^{end\_sync}(\pi, Channels, PN_k), \Pi, PN)$$

$$PN_3 = fold\ (\Upsilon^{end\_order}, \Pi, PN_2)$$

$$\Pi = \bigcup_{k=1}^{n} \{\Pi_k \mid \textbf{unit}\ (*, \Pi_k, *, *) \simeq u_k\}$$

$$\{(id_1, pid_1), \ldots, (id_m, pid_m)\} = \{(id, pid) \mid \textbf{interface\_point}\ (id, pid, *) \in InterfacePoints\}$$

$$\{u_1, \ldots, u_r\} = \{u \mid u \in Units \wedge type\_u(u) = \textbf{repetitive}\}$$
$$\{u_{r+1}, \ldots, u_n\} = \{u \mid u \in Units \wedge type\_u(u) = \textbf{non-repetitive}\}$$

$$P = \{p^i, \overline{p^f}, p^f, p^r\} \cup \left(\bigcup_{k=1}^{n} P_k\right) \cup \left(\bigcup_{j=1}^{m} \{p_j, \overline{p_j}\}\right) \cup \left(\bigcup_{k=1}^{r} \{p_k^{back}\}\right)$$

$$T = \{t^i, \overline{t^f}, t^f\} \cup \{t_k^{ok} \mid 1 \leq k \leq s\} \cup \left(\bigcup_{k=1}^{n} T_k\right) \tag{5.1}$$

$$A^+ = \left\{ \begin{array}{c} (p^i, t^i), \\ (\overline{t^f}, \overline{p^f}), \\ (\overline{p^f}, t^f), \\ (t^f, p^f) \end{array} \right\} \cup \left(\bigcup_{k=1}^{r} \left\{ \begin{array}{c} (t^i, p_k^i), \\ (t^i, p_k^{back}) \\ (p_k^{back}, \overline{t^f}), \\ (p_k^{back}, t_k^{back}), \\ (t_k^{back}, p_k^{back}), \\ (p_k^f, t^f) \end{array} \right\} \right) \cup \left(\bigcup_{k=r+1}^{n} \left\{ \begin{array}{c} (t^i, p_k^i), \\ (p_k^f, \overline{t^f}) \end{array} \right\} \right) \cup \left(\bigcup_{k=1}^{n} A_k \right)$$

$$\{t_k^{back}\} = \{t \mid \{(p_k^f, t), (t, p_k^i)\} \subset A_k\},\ \forall k : 1 \leq k \leq r$$

$$(P_k, T_k, A_k, \rho_k, p_k^i, p_k^f) = \Upsilon^{unit}(u_k, Mappings),\ \forall k : 1 \leq k \leq n$$

$$A^r = \left\{ \begin{array}{ll} \{(p^r, \overline{t^f})\} & , \text{se } r = n \text{ (não há unidade não-repetitiva)} \\ \{(t^i, p^r), (p^r, \overline{t^f})\} & , \text{caso contrário} \end{array} \right.$$

$$\rho(t) = \left\{ \begin{array}{ll} \rho_k(t) & , \text{se } t \in T_k \\ \lambda & , \text{caso contrário} \end{array} \right.$$

$$qualifier(p_j) = \{(id_j, \text{POK}), (pid_j, \text{POK})\},\ \forall j : 1 \leq j \leq m$$
$$qualifier(\overline{p_j}) = \{(id_j, \overline{\text{POK}}), (pid_j, \overline{\text{POK}})\},\ \forall j : 1 \leq j \leq m$$

A Regra 5.1, ilustrada na Figura 5.1, define a função $\Upsilon^{\#}$. Os lugares p^i e p^f caracterizam os estados inicial e final da aplicação. Assim, o disparo da transição t^i caracteriza a inicialização do programa, enquanto o disparo da transição t^f caracteriza a sua finalização. Seja n o número de processos do programa # em questão. Para cada processo, a rede de Petri que modela seu comportamento é induzida pela aplicação da função Υ^{unit} sobre sua descrição. As redes de Petri induzidas para cada processo são conectadas de forma a descrever sua execução paralela. Com este fim, são criados arcos que ligam a transição t^i aos lugares iniciais $(p_j^i, 1 \leq j \leq n)$ de cada rede induzida por um processo. A terminação do programa ocorre quando todos os processos não-repetitivos em sua constituição terminam, estado modelado pela marcação onde uma marca encontra-se depositada no lugar $\overline{p^f}$, após o disparo da transição $\overline{t^f}$. O disparo da transição $\overline{t^f}$ causa a retirada das marcas contidas nos lugares p_j^{back}, $1 \leq j \leq k$, impedindo que os k processos repetitivos executem

outra vez (disparo de t_j^{back}) após atingir sua marcação final (lugar p_j^f). O lugar p^r é usado para indicar a existência ou não de processos não-repetitivos. Caso exista pelo menos um processo desta natureza, é depositada uma marca nesse lugar, devido ao disparo da transição t^i. Caso contrário, esse lugar permanece não marcado, pela não existência do arco a partir da transição t^i. Neste caso, a transição $\overline{t^f}$ permanece bloqueada (transição morta), não podendo assim o programa atingir um estado final. Essa modelagem está de acordo com a semântica definida informalmente no Capítulo 3, onde foi estabelecido que um programa # que só contenha processos repetitivos não termina.

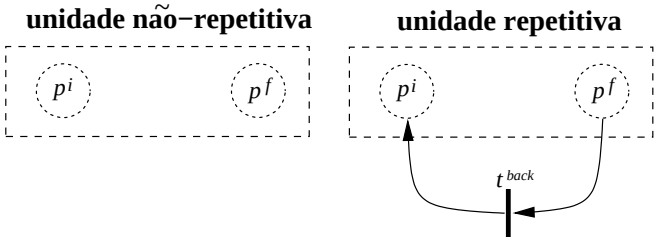


Figura 5.2. Processos (Regra 5.2)

As funções Υ^{end_sync} e Υ^{end_order} são responsáveis por introduzir na rede protocolos para sincronização da natureza do valor transmitido devido a ativação de portas, assunto que será discutido detalhadamente nas Seções 5.2.12 e 5.2.13. A função $\Upsilon^{channel}$ é encarregada de inserir na rede a informação de sincronização entre os processos contida na definição dos canais de comunicação. Esta é aplicada a cada canal (c_k) individualmente, gerando uma nova rede com as informações deste incluídas. A função $\Upsilon^{channel}$ será definida na Seção 5.2.14.

$$
\begin{gathered}
\Upsilon^{unit}(\textbf{unit } (uid, \Pi, type_u, \textbf{process}(\textbf{behavior } (S, action))) = (P, T \cup T^r, A \cup A^r, \rho, p^i, p^f) \\
\textbf{onde:} \\
(P, T, A, \rho, p^i, p^f) = \Upsilon^{action}(action, \Pi, 0) \\
(T^r, A^r) = \begin{cases} (\{t^{back}\}, \{(p^f, t^{back}), (t^{back}, p^i)\}) & \text{, se } type_u = \textbf{repetitive} \\ (\emptyset, \emptyset) & \text{, se } type_u = \textbf{non-repetitive} \end{cases} \\
\rho(t) = \begin{cases} \overline{\rho}(t) & \text{, se } t \in \overline{T},\ 1 \leq k \leq n \\ \lambda & \text{, caso contrário} \end{cases}
\end{gathered}
\tag{5.2}
$$

$$
\begin{gathered}
\Upsilon^{unit}(\textbf{unit } (uid, \Pi, type_u, \textbf{cluster}(c, a)) = (P \cup P^+, T, A, \rho, p^i, p^f) \\
\textbf{onde:} \\
(P, T, A, \rho, p^i, p^f) = \Upsilon^{\#}(c) \\
\{(ip_1, pid_1,), \ldots, (ip_n, pid_n)\} = a \\
P^+ = \bigcup_{k=1}^{n} \{p_k, \overline{p_k}\} \\
qualifier(p_k) = \{(ip_k, \text{POK}), (pid_k, \text{POK})\},\ \forall k : 1 \leq k \leq n \\
qualifier(\overline{p_k}) = \{(ip_k, \overline{\text{POK}}), (pid_k, \overline{\text{POK}})\},\ \forall k : 1 \leq k \leq n
\end{gathered}
\tag{5.3}
$$

As Regras 5.2 e 5.3 definem a função Υ^{unit}, a qual constrói a rede de Petri que modela o comportamento de uma unidade, aplicando-se respectivamente a *processos* e *aglomerados*. Na modelagem de processos, ilustrado na Figura 5.2, os lugares p^i e p^f

modelam respectivamente os estados inicial e final da unidade. Caso o processo seja repetitivo, é criada uma transição t^{back}, com lugar de saída p^i e lugar de entrada p^f, a qual permite que a unidade retorne ao seu estado inicial após atingir seu estado final. Na modelagem de aglomerados, é induzida a rede que descreve o comportamento do componente ao qual a unidade está associada. Os lugares p_j e $\overline{p_j}$, $1 \leq j \leq n$ são usados para conectar as portas da unidade aos pontos de entrada e saída do componente, utilizando identificação de lugares equivalentes por meio de qualificadores.

A seções que se seguem definem a função Υ^{action}, indutivamente sobre os combinadores que definem a descrição comportamental do processo. O número inteiro correspondente ao terceiro argumento dessa função indica a profundidade de aninhamento da ação no contexto de combinadores **repeat**. Essa informação será importante para modelagem da condição de terminação envolvida na semântica deste combinador.

$$\Upsilon^{action}(\mathbf{skip}, *, *) = (\{p\}, \emptyset, \emptyset, p, p) \tag{5.4}$$

Figura 5.3. "Ação nula" Modelada com Redes de Petri (Regra 5.4)

5.2.3 A Ação Nula(skip)

O combinador **skip** não possui qualquer efeito(*ação nula*). Sua tradução para redes de Petri é formalizada na Regra 5.4 e ilustrada na Figura 5.3, com interpretação óbvia.

5.2.4 A Ação de Ativação de Portas (? e !)

O conceito de *porta de comunicação* foi introduzido no Capítulo 3. Estas podem ser *individuais* ou *agrupamentos*. A princípio, um agrupamento de portas é constituído por um conjunto de portas individuais. Entretanto, como efeito da aplicação de operações sobre unidades (Seção 3.2.9), um agrupamento pode ainda admitir em sua constituição outros agrupamentos, ditos *agrupamentos aninhados*, recursivamente. Um agrupamento pode ainda ser caracterizado como *choice* ou não-*choice*, dependendo de como um valor a ser transmitido é distribuído entre as portas (ou agrupamentos) aninhadas ao agrupamento. Nas discussões ao longo do texto, o termo porta é aplicado indistintamente à agrupamentos de portas ou portas individuais.

$$
\begin{gathered}
\Upsilon^{action}(\textbf{activate}\ pid, \Pi, \_) = (P \cup P^{+}, T \cup T^{+}, A \cup A^{+}, \rho, p^{i}, p^{f}) \\
\textbf{onde:} \\
(P, T, A, \rho, \overline{p^{i}}, p^{f}, *) = \Upsilon^{port}(\pi) \\
\pi \in \Pi \wedge (\pi \simeq \textbf{port}(pid, *) \vee \pi \simeq \textbf{group}(pid, *, *, *)) \\
\{pid_1, \ldots, pid_n\} = nested\_groups\_pids\ (\pi) \\
P^{+} = \{p^{i}, p\} \cup \bigcup_{k=1}^{n} \{\overline{p^{ok}}_k\} \\
T^{+} = \{t^{i}, \overline{t^{i}}\} \\
A^{+} = \{(p^{i}, t^{i}), (t^{i}, p), (p, \overline{t^{i}}), (\overline{t^{i}}, \overline{p^{i}})\} \cup \bigcup_{k=1}^{n} \{(t^{i}, \overline{p^{ok}}_k)\} \\
qualifier(\overline{p^{ok}}_k) = (pid_k, \overline{\text{POK}}),\ \forall k : 1 \leq k \leq n \\
qualifier(t^{i}) = (pid, \text{BEGIN\_ACTIVATION}, \textbf{unique}) \\
qualifier(\overline{t^{i}}) = (pid, \overline{\text{BEGIN\_ACTIVATION}}, \textbf{unique})
\end{gathered}
\tag{5.5}
$$

Na descrição do comportamento da interface de um processo, as primitivas de ativação de portas de comunicação, ! e ?, correspondem respectivamente às primitivas de *passagem de mensagens* empregadas em programação concorrente sobre arquiteturas distribuídas. A definição da função Υ^{action} sobre estas primitivas, as quais são representadas em *abstract #* pela primitiva **activate**, é apresentada na Regra 5.5. A função *nested_groups_pids* computa os identificadores de portas dos agrupamentos aninhados ao agrupamento fornecido como argumento. Os lugares $\overline{p^{ok}}_k$, cujo significado será explicado nos parágrafos que se seguem estão associados a cada agrupamento aninhado à porta ativada, caso exista algum. Estes devem ser inicializados com uma marca no início da ativação do grupo, justificando a existência da transição t^{i}.

A função Υ^{port}, diretamente empregada na definição de Υ^{action}, modela a semântica de ativação de uma porta. Encontra-se definida atrabés das Regras 5.6, 5.7, 5.8 e 5.9 por indução sobre a estrutura da porta em questão, aplicando-se a agrupamentos de portas sob qualquer nível de aninhamento. A rede induzida pela ativação de uma porta define um *lugar destacado*, no qual a presença de uma marca caracteriza o fato de que a porta está pronta para se comunicar (habilitada). Por definição, como visto no Capítulo 3, uma porta individual (incluindo aquelas aninhadas à agrupamentos) encontra-se habilitada se ela própria e seu par de comunicação encontram-se ativadas. Isso será modelado com a sobreposição das redes geradas para cada processo devido a modelagem de canais de comunicação (Seção 5.2.14), onde os lugares destacados de pares de portas individuais conectadas constituirão-se um protocolo de sincronização capaz de caracterizar o modo de comunicação suportado pelo canal. Na rede induzida pela ativação de um agrupamento *choice*, o lugar destacado conterá uma marca sempre que pelo menos um dos lugares destacados das redes induzidas pela ativação de suas portas diretamente aninhadas contiver uma marca. Na ativação de um agrupamento não-*choice*, o lugar destacado contém uma marca sempre que os lugares destacados das redes induzidas por todas as suas portas diretamente aninhadas contiverem uma marca. Assumindo-se o fato de que agrupamentos formam estruturas indutivamente definidas, a presença de uma marca no lugar destacado de um agrupamento de portas indica que este está pronto para completar uma operação

de comunicação, envolvendo suas portas aninhadas. O lugar destacado é definido por um lugar adicional (p^{ok}) que compõe a estrutura da rede de Petri induzida por Υ^{port}.

De forma a obedecer as restrições mencionadas no parágrafo anterior, a função Υ^{port} é definida de forma distinta para cada um dos seguintes tipos de porta: portas individuais não-*stream* (Equação 5.6), portas individuais *stream* (Equação 5.7), agrupamentos não-*choice* (Equação 5.8) e agrupamentos *choice* (Equação 5.9).

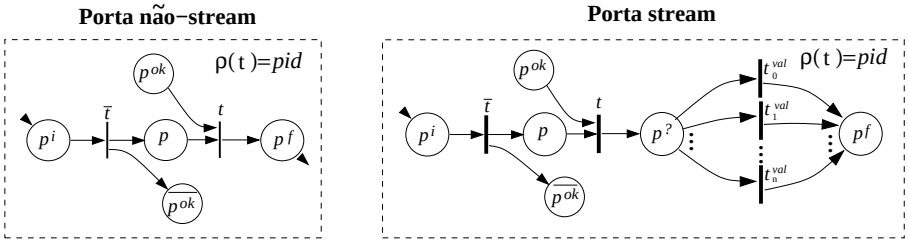


Figura 5.4. Ativação de Portas (?, !) modelada como redes de Petri (Regras 5.6 e 5.7)

$$\Upsilon^{port}(\textbf{port}\ (pid, *, \textbf{single}, \textbf{non-stream}), *) = (P, T, A, \rho, p^i, p^f, p^{ok})$$

$$\textbf{onde:}$$

$$P = \{p^i, p, \overline{p^{ok}}, p^{ok}, p^f\}$$

$$\begin{array}{c} qualifier(\overline{p^{ok}}) = (pid, \overline{\text{POK}}) \\ qualifier(p^{ok}) = (pid, \text{POK}) \end{array} \tag{5.6}$$

$$T = \{t, \overline{t}\}$$

$$A = \{(p^i, \overline{t}), (\overline{t}, p), (\overline{t}, \overline{p^{ok}}), (p, t), (p^{ok}, t), (t, p^f)\}$$

$$\begin{array}{c} \rho(t) = pid \\ \rho(\overline{t}) = \lambda \end{array}$$

$$\Upsilon^{port}(\textbf{port}\ (pid, *, \textbf{single}, \textbf{stream}, *)) = (P, T, A, \rho, p^i, p^f, p^{ok})$$

$$\textbf{onde:}$$

$$P = \{p^i, p, \overline{p^{ok}}, p^{ok}, p^?, p^f\}$$

$$T = \{\overline{t}, t\} \cup \left(\bigcup_{k=1}^{n+1} \{t_k^{val}\} \right)$$

$$\begin{array}{c} qualifier(t_k^{val}) = (pid, \text{SYNC\_END}, k, \textbf{unique}),\ \forall k : 0 \leq k \leq n \\ qualifier(p^{ok}) = (pid, \text{POK}) \end{array} \tag{5.7}$$

$$A = \left\{ \begin{array}{c} (p^i, \overline{t}), (\overline{t}, p), (p, t), \\ (\overline{t}, \overline{p^{ok}}), (p^{ok}, t), (t, p^?) \end{array} \right\} \cup \left(\bigcup_{k=0}^{n} \{(p^?, t_k^{val}), (t_k^{val}, p^f)\} \right)$$

$$\begin{array}{c} \rho(t) = pid \\ \rho(\overline{t}) = \lambda \end{array}$$

Considere-se a ativação de portas individuais (Regras 5.6 e 5.7), ilustrada na Figura 5.4. A transição $\overline{t}$ modela a ativação da porta. Assim, a presença de uma marca no lugar p modela o estado onde a porta encontra-se ativada. A efetivação da comunicação é modelada pelo disparo da transição t. Quando o canal sobre o qual a porta está conectada é *síncrono*, a ativação de t pode ser atrasada até a ativação de seu par na comunicação, estado caracterizado pela presença de uma marca no lugar destacado p^{ok}, o qual é retornado por Υ^{port}.

Especificamente na Regra 5.7 (portas *stream*), um conflito em um lugar $p^?$ é empregado para modelar o teste que indica a natureza do valor transmitido na ativação. Como discutido na Seção 3.2.1, em Haskell$_\#$, *streams* são capazes de transmitir listas aninhadas de um processo emissor para um processo receptor, sendo possível recuperar neste a estrutura de aninhamento original da lista transmitida. Para que isso seja possível, um valor especial é transmitido na *stream* sempre que o final de uma lista aninhada é alcançado. Dessa forma, seja p uma porta *stream* e n o nível de aninhamento da lista correspondente à *stream* transmitida pela porta. A natureza do valor transmitido por p indica se este constitui um valor normal de dados ou um marcador de final de uma lista que ocorre em um certo aninhamento i. Portanto, a natureza de um valor transmitido por uma porta é uma dentre $n + 1$ possíveis, incluindo valor de dados e indicadores de final de lista para cada nível de aninhamento. Por convenção, a transmissão de um valor marcador de final de lista em um aninhamento i é modelado pelo disparo da transição t^{v_i}, $0 \leq i \leq n - 1$, enquanto o disparo da transição t^{v_n} modela a transmissão de um item de dados. O nível de aninhamento da *stream* transmitida por uma porta corresponde ao número de símbolos "*" que ocorrem após o idenficador da porta na declaração da interface que a caracteriza. A necessidade em modelar-se a natureza do último valor transmitido decorre do uso desta informação para modelagem da condição de terminação de ocorrências do combinador **repeat**, um refinamento que pode ser útil para caracterizar o comportamento de um programa # de forma mais detalhada.

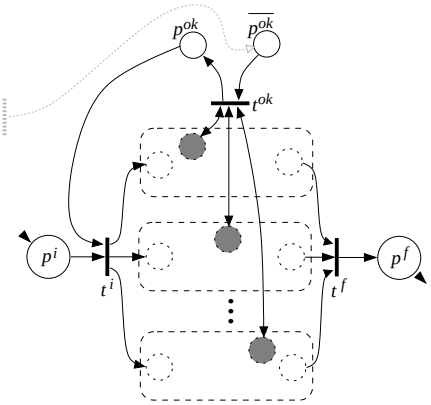


Figura 5.5. Ativação de um Agrupamento de Portas não-*choice*(?, !) (Regra 5.8)

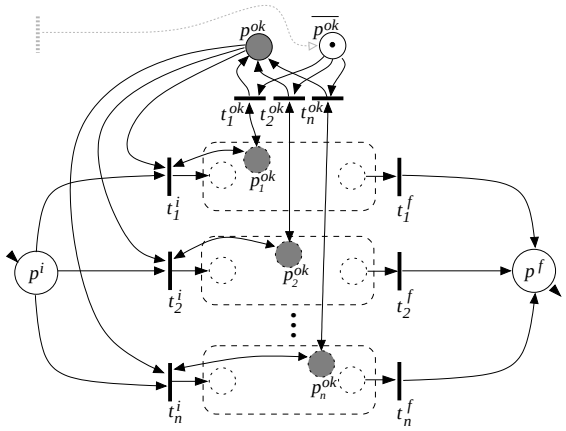


Figura 5.6. Ativação de um Agrupamento de Portas *choice*(?, !) (Regra 5.9)

$$\Upsilon^{port}(\textbf{port}\ (pid, \_, \textbf{group}\ (*, \textbf{all}, N), *)) = (P, T, A, \rho, p^i, p^f, p^{ok})$$

onde:

$$\{p_1, p_2, \ldots, p_n\} = N$$

$$(P_k, T_k, A_k, \rho_k, p_k^i, p_k^f, p_k^{ok}) = \Upsilon^{port}(p_k),\ \forall k : 1 \leq k \leq n$$

$$\begin{gathered} P = \{p^i, p^f, p^{ok}, \overline{p^{ok}}\} \cup (\bigcup_{k=1}^{n} P_k) \\ T = \{t^i, t^f, t^{ok}\} \cup (\bigcup_{k=1}^{n} T_k) \\ A = \left\{ \begin{array}{c} (p^i, t^i), (p^{ok}, t^i) \\ (t^f, p^f), (t^{ok}, p^{ok}), (\overline{p^{ok}}, t^{ok}) \end{array} \right\} \cup \left(\bigcup_{k=1}^{n} \left(\left\{ \begin{array}{c} (t^i, p_k^i), (p_k^f, t^f), \\ (p_k^{ok}, t^{ok}), (t^{ok}, p_k^{ok}) \end{array} \right\} \cup A_k \right) \right) \end{gathered} \tag{5.8}$$

$$\begin{gathered} qualifier(p^{ok}) = (pid, \text{POK}) \\ qualifier(\overline{p^{ok}}) = (pid, \overline{\text{POK}}) \end{gathered}$$

$$qualifier(t^{ok}) = (pid, \text{TOK})$$

$$
\begin{gathered}
\Upsilon^{port}(\textbf{port}\ (*,*,\textbf{group}\ (*,\textbf{any},N),*)) = (P,T,A,\rho,p^i,p^f,p^{ok}) \\
\textbf{onde:} \\
\{p_1,p_2,\ldots,p_n\} = N \\
(P_k,T_k,A_k,\rho_k,p_k^i,p_k^f,p_k^{ok}) = \Upsilon^{port}(p_k),\ \forall k : 1 \leq k \leq n \\
P = \{p^i,p^f,p^{ok},\overline{p^{ok}}\} \cup (\bigcup_{k=1}^{n} P_k) \\
T = \bigcup_{k=1}^{n}\{t_k^i,t_k^f,t_k^{ok}\} \cup (\bigcup_{k=1}^{n} T_k) \\
A = \bigcup_{k=1}^{n}\left(\left\{\begin{array}{c}(p^i,t_k^i),(t_k^i,p_k^i),(p_k^f,t_k^f),\\(t_k^f,p^f),(t_k^{ok},p^{ok}),(\overline{p^{ok}},t_k^{ok}),\\(p_k^{ok},t_k^{ok}),(t_k^{ok},p_k^{ok}),\\(p_k^{ok},t_k^i),(t_k^i,p_k^{ok})\end{array}\right\} \cup A_k\right) \\
qualifier(p^{ok}) = (pid,\text{POK}) \\
qualifier(\overline{p^{ok}}) = (pid,\overline{\text{POK}}) \\
qualifier(t_k^{ok}) = (pid,k,\text{TOK}),\ \forall k : 1 \leq k \leq n
\end{gathered}
\tag{5.9}
$$

As regras referentes a ativação de agrupamentos *choice* (5.9) e não-*choice* (5.8) são ilustradas respectivamente nas Figuras 5.5 e 5.6. O lugar p^{ok} é definido como o lugar destacado, sendo retornado por Υ^{port}. Por construção, garante-se o atendimento às restrições impostas ao lugar destacado definidas anteriormente. O lugar dual de p^{ok}, denominado $\overline{p^{ok}}$, é usado para evitar que a ativação repetida das transições t^{ok}, em grupos não-*choice*, ou t_k^{ok}, $1 \leq k \leq n$, em grupos *choice*, causem o depósito de mais de uma marca no lugar p^{ok}, em uma certa ativação.

Na Seção 5.2.12, definiremos a função Υ^{end_sync}, responsável por inserir na estrutura da sub-rede induzida pelas ativações de portas *stream* um protocolo de sincronização da natureza de valores transmitidos entre estas, admitindo uma conjunto de restrições que serão explicadas na referida seção.

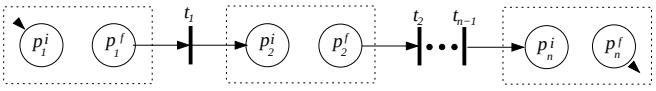


Figura 5.7. Sequência de Ações (**seq**) (Regra 5.10)

$$\Upsilon^{action}\left(\begin{array}{ll}\textbf{seq}\{ & a_1;\\ & a_2;\\ & \vdots\\ & a_n\}\end{array}, \Pi, d\right) = \left(P, T, A, \rho, p_1^i, p_n^f\right),$$

onde:

$$(P_k, T_k, A_k, \rho_k, p_k^i, p_k^f) = \Upsilon^{action}(a_k, \Pi, d),\ \forall k : 1 \leq k \leq n$$

$$P = \bigcup_{k=1}^{n} P_k$$

$$T = \left(\bigcup_{k=1}^{n-1}\{t_k\}\right) \cup \left(\bigcup_{k=1}^{n} T_k\right) \tag{5.10}$$

$$A = \left(\bigcup_{k=1}^{n-1}\{(p_k^f, t_k), (t_k, p_{k+1}^i)\}\right) \cup \left(\bigcup_{k=1}^{n} A_k\right)$$

$$\rho(t) = \left\{\begin{array}{ll}\rho_k(t) & \text{, se } t \in T_k,\ 1 \leq k \leq n\\ \lambda & \text{, caso contrário}\end{array}\right. ,\ t \in T$$

5.2.5 Sequenciamento de Ações (seq)

O combinador **seq** descreve a execução sequencial de um coleção de ações, aqui referenciadas como $a_1, a_2, \ldots, a_n$. Pode ser modelado pela composição sequencial das redes de Petri induzidas por cada ação, como ilustrado na Figura 5.7 e formalizada na Regra 5.10.

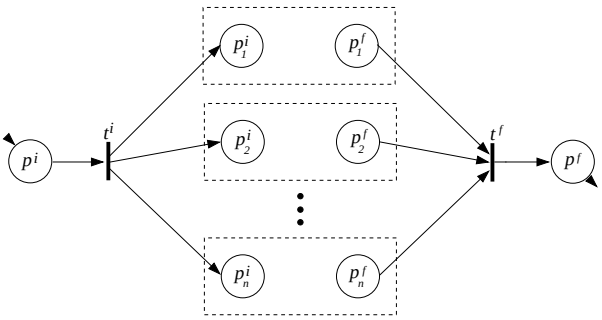


Figura 5.8. Concorrência de Ações (**par**) (Regra 5.11)

$$\Upsilon\left(\begin{array}{ll}\textbf{par}\{ & a_1;\\ & a_2;\\ & \vdots\\ & a_n\}\end{array}, \Pi, d\right) = \left(P, T, A, \rho, p^i, p^f\right),$$

onde:

$$\begin{gathered}(P_k, T_k, A_k, \rho_k, p_k^i, p_k^f) = \Upsilon^{action}(a_k, \Pi, d),\ \forall k : 1 \leq k \leq n\\ P = \{p^i, p^f\} \cup \left(\bigcup_{k=1}^{n} P_k\right)\\ T = \{t^i, t^f\} \cup \left(\bigcup_{k=1}^{n} T_k\right)\\ A = \{(p^i, t^i), (t^f, p^f)\} \cup \left(\bigcup_{k=1}^{n} \left(\{(t^i, p_k^i), (p_k^f, t^f)\} \cup A_k\right)\right)\\ \rho(t) = \begin{cases} \rho_k(t) & \text{, se } t \in T_k,\ 1 \leq k \leq n\\ \lambda & \text{, caso contrário}\end{cases}\ ,\ t \in T\end{gathered} \tag{5.11}$$

5.2.6 Concorrência de Ações (par)

O combinador **par** descreve a execução concorrente (intercalada) de n ações, aqui referenciadas como $a_1, a_2, \ldots, a_n$. Modela-se pela composição paralela das redes de Petri induzidas por cada ação, como ilustrado na Figura 5.8 e formalizada na Regra 5.11.

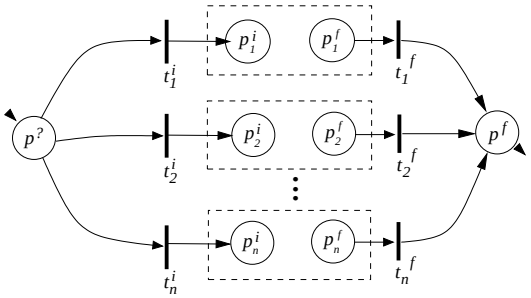


Figura 5.9. Execução Alternativa de Ações (**alt**) (Regra 5.12)

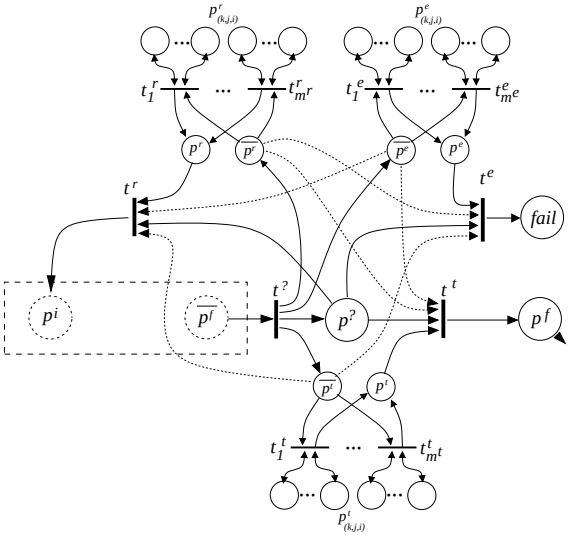


Figura 5.10. Ações Repetidas (**repeat** ... **until**) Modeladas como redes de Petri

$$\Upsilon\left(\begin{array}{ll}\textbf{alt}\{ & a_1;\\ & a_2;\\ & \quad .\\ & \quad .\\ & a_n\}\end{array}, \Pi, d\right) = \left(P, T, A, \rho, p^?, p^f\right),$$

onde:

$$(P_k, T_k, A_k, \rho_k, p_k^i, p_k^f) = \Upsilon^{action}(a_k, \Pi, d),\ \forall k : 1 \leq k \leq n$$

$$P = \{p^?, p^f\} \cup \left(\bigcup_{k=1}^{n} P_k\right)$$

$$T = \left(\bigcup_{k=1}^{n} \{t_k^i, t_k^f\}\right) \cup \left(\bigcup_{k=1}^{n} T_k\right) \tag{5.12}$$

$$A = \bigcup_{k=1}^{n} \left(A_k \cup \overline{A_k} \cup \{(p^?, t_k^i), (t_k^i, p_k^i), (p_k^f, t_k^f), (t_k^f, p^f)\}\right)$$

$$\rho(t) = \begin{cases} \rho_k(t) & \text{, se } t \in T_k,\ 1 \leq k \leq n \\ \lambda & \text{, caso contrário} \end{cases}$$

5.2.7 Escolha entre Ações (alt)

O combinador **alt** descreve a execução de uma ação escolhida dentre uma coleção de ações, aqui referenciadas como $a_1, a_2 \ldots a_n$. A modelagem do combinador **alt** com redes de Petri é ilustrada na Figura 5.9 e formalizada na Regra 5.12.

A escolha da ação é modelada em redes de Petri pelo conflito no lugar $p^?$. A partir da marcação onde uma marca encontra-se depositada neste lugar, é disparada uma das transições t_k^i, $1 \leq k \leq n$, as quais modelam a escolha de uma certa ação, iniciando sua execução.

$$\Upsilon^{action}(\textbf{repeat\_until}\ (a, C), \Pi, d) = (P \cup P^+, T \cup T^+, A \cup A^+, \rho^+, p^i, p^f)$$

onde:

$$\begin{array}{l} (id^t_{(1,1)}\ \&\ \ldots\ \&\ id^t_{(1,n^t_1)}) \mid \ldots \mid (id^t_{(m^t,1)}\ \&\ \ldots\ \&\ id^t_{(m^t,n^t_{m^t})}) = C^f \\ (id^r_{(1,1)}\ \&\ \ldots\ \&\ id^r_{(1,n^r_1)}) \mid \ldots \mid (id^r_{(m^r,1)}\ \&\ \ldots\ \&\ id^r_{(m^r,n^r_{m^r})}) = C^r \\ (id^e_{(1,1)}\ \&\ \ldots\ \&\ id^e_{(1,n^e_1)}) \mid \ldots \mid (id^e_{(m^e,1)}\ \&\ \ldots\ \&\ id^e_{(m^e,n^e_{m^e})}) = C^e \end{array}$$

$$(P, T, A, \rho, p^i, \overline{p^f}) = \Upsilon^{action}(a, \Pi, d+1)$$

a^t_k é o aninhamento da porta id^t_k
a^r_k é o aninhamento da porta id^r_k
a^e_k é o aninhamento da porta id^e_k

$$P^+ = \left\{ \begin{array}{c} \overline{p^?}, fail, p^f, p^t, \\ \overline{p^t}, p^r, \overline{p^r}, p^e, \overline{p^e} \end{array} \right\} \cup \left(\bigcup_{k=1}^{m^t} \bigcup_{j=1}^{n^t_k} \bigcup_{i=0}^{a^t_j} \{p^t_{(k,j,i)}\} \right) \cup \left(\bigcup_{k=1}^{m^r} \bigcup_{j=1}^{n^r_k} \bigcup_{i=0}^{a^r_j} \{p^r_{(k,j,i)}\} \right) \cup \left(\bigcup_{k=1}^{m^e} \bigcup_{j=1}^{n^e_k} \bigcup_{i=0}^{a^e_j} \{p^e_{(k,j,i)}\} \right)$$

$$T^+ = \{t^?, t^t, t^r, t^e\} \cup \bigcup_{k=1}^{m^t} \{t^t_k\} \cup \bigcup_{k=1}^{m^r} \{t^r_k\} \cup \bigcup_{k=1}^{m^e} \{t^e_k\}$$

$$A^+ = \{(t^?, p^?)\} \cup A^t \cup A^r \cup A^e$$

$$A^t = \left\{ \begin{array}{c} (t^?, \overline{p^t}), (p^?, t^t), \\ (p^t, t^t), (t^t, p^f), \\ (\overline{p^t}, t^r), (\overline{p^t}, t^e) \end{array} \right\} \cup \left(\bigcup_{k=1}^{m^t} \left\{ \begin{array}{c} (\overline{p^t}, t^t_k), \\ (t^t_k, p^t) \end{array} \right\} \cup \left(\bigcup_{j=1}^{n^t_k} \bigcup_{i=1}^{d-1} \left\{ \begin{array}{c} (\overline{p^t_{(k,j,i)}}, t^t_k), \\ (t^t_k, \overline{p^t_{(k,j,i)}}) \end{array} \right\} \right) \right)$$

$$A^r = \left\{ \begin{array}{c} (t^?, \overline{p^r}), (p^?, t^r), \\ (p^r, t^r), (t^r, p^i), \\ (\overline{p^r}, t^t), (\overline{p^r}, t^e) \end{array} \right\} \cup \left(\bigcup_{k=1}^{m^r} \left\{ \begin{array}{c} (\overline{p^r}, t^r_k), \\ (t^r_k, p^r) \end{array} \right\} \cup \left(\bigcup_{j=1}^{n^r_k} \bigcup_{i=d}^{a^r_{(k,j,i)}} \left\{ \begin{array}{c} (\overline{p^r_{(k,j,i)}}, t^r_k), \\ (t^r_k, \overline{p^r_{(k,j,i)}}) \end{array} \right\} \right) \right)$$

$$A^e = \left\{ \begin{array}{c} (t^?, \overline{p^e}), (t^e, fail) \\ (p^?, t^e), (p^e, t^e), \\ (\overline{p^e}, t^t), (\overline{p^e}, t^r) \end{array} \right\} \cup \left(\bigcup_{k=1}^{m^e} \left\{ \begin{array}{c} (\overline{p^e}, t^e_k), \\ (t^e_k, p^e) \end{array} \right\} \cup \left(\bigcup_{j=1}^{n^e_k} \bigcup_{i=1}^{d-1} \left\{ \begin{array}{c} (\overline{p^e_{(k,j,i)}}, t^e_k), \\ (t^e_k, \overline{p^e_{(k,j,i)}}) \end{array} \right\} \right) \right) \tag{5.13}$$

$$qualifier(fail) = \textsc{Fail}$$

$$qualifier(\overline{p^t_{(k,j,i)}}) = (\overline{\textsc{Loop\_Flags}}, id^t_{(k,j,i)}, \left\{ \begin{array}{l} \forall k : 1 \le k \le m^t \\ \forall j : 1 \le j \le n^t_k \\ \forall i : 1 \le i \le a^t_{(k,j,i)} \end{array} \right.$$

$$qualifier(\overline{p^r_{(k,j,s,i)}}) = (\overline{\textsc{Loop\_Flags}}, id^r_{(k,j,i)}, \left\{ \begin{array}{l} \forall k : 1 \le k \le m^r \\ \forall j : 1 \le j \le n^r_k \\ \forall i : 1 \le i \le a^r_{(k,j,i)} \end{array} \right.$$

$$qualifier(\overline{p^e_{(k,j,s,i)}}) = (\overline{\textsc{Loop\_Flags}}, id^e_{(k,j,i)}, \left\{ \begin{array}{l} \forall k : 1 \le k \le m^e \\ \forall j : 1 \le j \le n^e_k \\ \forall i : 1 \le i \le a^e_{(k,j,i)} \end{array} \right.$$

5.2.8 Execução Repetida de Ações (repeat ... until)

O combinador **repeat** modela a execução repetida de uma ação. Sua semântica foi discutida na Seção 3.2.1. A tradução do combinador **repeat**, com condição de terminação especificada pela cláusula **until**, é ilustrada na Figura 5.10 e formalizada na Regra 5.13. O argumento d, na função Υ^{action}, indica a profundidade de aninhamento da ocorrência do combinador **repeat** em relação a uma ocorrência mais externa deste combinador. Caso esta não exista (o próprio é o mais externo), o valor de d é 1.

Em ocorrências do combinador **repeat**, a verificação da condição de terminação especificada por meio da cláusula **until** exige a modelagem da caracterização da natureza do último valor transmitido pelas portas *stream* referenciadas na condição. Com essa finalidade, a rede de Petri que modela a semântica de um programa # inclui para cada porta desse tipo, identificada por i na discussão que se segue, um conjunto de lugares $\textsc{Loop\_Flags}_i$, satisfazendo a seguinte restrição:

$$\text{LOOP\_FLAGS}_i = \{flag_k \mid 0 \leq k \leq a_i\}, \tag{5.14}$$

onde a_i corresponde ao fator de aninhamento da i-ésima porta. Assim, para cada porta de aninhamento a_i, existem $a_i + 1$ lugares, tais quais a presença de uma marca caracteriza o recebimento de um valor de cada uma das naturezas possíveis na sua mais recente ativação. Formalmente, deve ainda ser obedecida a seguinte restrição:

$$\forall i : \sum_{p \in \text{LOOP\_FLAGS}_i} p = 1 \tag{5.15}$$

Ou seja, os lugares em LOOP_FLAGS_i podem ser interpretados como semáforos divididos modelados com redes de Petri. Faz-se também necessário um conjunto de lugares $\overline{\text{LOOP\_FLAGS}_i}$, duais e mutuamente exclusivos aos lugares correspondentes no conjunto LOOP_FLAGS_i, os quais serão também usados para modelagem de condições de uma repetição. A seguinte restrição caracteriza o conjunto de lugares $\overline{\text{LOOP\_FLAGS}_i}$, relacionando-o com o conjunto LOOP_FLAGS_i:

$$\forall i, \forall k, 1 \leq k \leq a_i : flag_k + \overline{flag_k} = 1 \mid flag_k \in \text{LOOP\_FLAGS}_i \wedge \overline{flag_k} \in \overline{\text{LOOP\_FLAGS}_i} \tag{5.16}$$

As restrições expressas pelas Equações 5.15 e 5.16 são introduzidas na rede de Petri que modela o programa # pela função Υ^{end_sync}, a qual será definida adiante. Os próximos parágrafos descrevem como os lugares em $\overline{\text{LOOP\_FLAGS}_i}$ são usados para modelar condições de terminação em ocorrências do combinador **repeat**.

O conflito no lugar $p^?$ modela a decisão a cerca da finalização da repetição ou execução de uma nova iteração. Os disparo das transições t^r, t^t e t^e modelam, respectivamente, a execução de uma nova iteração, a finalização do laço e a ocorrência de um erro por falha de sincronia das *streams* transmitidas, caracterizada pela semântica dos delimitadores $<$ e $>$ quando empregados na condição de terminação. Obviamente, deve-se garantir que o disparo destas seja mutuamente exclusivo.

Seja i uma das portas que aparecem na condição da repetição. Para calcular o valor do predicado após uma iteração, foi mostrado na Seção 3.2.1 como caracterizar os valores lógicos (*falso* ou *verdadeiro*) que devem ser associados a cada porta. Baseado no que foi definido e supondo d o aninhamento da ocorrência de **repeat** em questão, o valor verdadeiro pode ser verificado na rede de Petri pela presença de marcas em todos os lugares $\overline{flag_k}$ em $\overline{\text{LOOP\_FLAGS}_i}$, tais que $d \leq k \leq n$. De acordo com as Equações 5.15 e 5.16, esse estado garante que um valor marcador de final de lista com aninhamento maior ou igual a d foi transmitido na mais recente ativação da porta. Por dualidade da negação, o valor *falso* pode ser verificado pela ocorrência de uma marcação onde todos os lugares $\overline{flag_k}$, tais que $1 \leq k \leq d-1$, possuem marcas.

Na Seção 3.2.1 foi ainda discutido a caracterização da condição para execução de uma nova iteração ($\neg\overline{C}$, onde $\overline{C}$ corresponde a negação de C asssumindo-se a semântica dos delimitadores $<$ e $>$ definida na Equação 3.6) e para a ocorrência de um erro de sincronização ($\neg C \wedge \overline{C}$). Estas causam o disparo das transições t^r e t^e, respectivamente. Por construção, tais condições, bem como a condição C original, são mutuamente exclusivas, o que exige o disparo mutuamente exclusivo das transições t^r, t^t e t^e.

5.2.8.1 Habilitação de $\mathbf{t^t}$ Para $1 \leq k \leq m, 1 \leq j \leq n_k, d \leq i \leq a_{(i,j)}$, a sub-rede formada pelos lugares p^t, $\overline{p^t}$ e $p^t_{(k,j,i)}$ e pelas transições t^t_j pode ser usada para verificar o predicado que caracteriza a condição de terminação da repetição (C). Como o predicado encontra-se em sua forma normal disjuntiva (FND), deve-se garantir que pelo menos uma das condições que constituem as disjunções seja verdadeiro. Isso é caracterizado pelo depósito de uma marca no lugar p^t. O lugar $\overline{p^t}$ evita que mais de uma marca seja depositada neste lugar, uma vez que a condição precisa ser testada uma única vez.

5.2.8.2 Habilitação de $\mathbf{t^r}$ Para $1 \leq k \leq m, 1 \leq j \leq n_k, 0 \leq i < d$, a sub-rede formada pelos lugares p^r, $\overline{p^r}$ e $p^r_{(k,j,i)}$ e pelas transições t^r_j é capaz de verificar o predicado que caracteriza a condição de terminação da repetição ($\neg C$), em sua forma normal disjuntiva, obedecendo o que foi descrito na Seção 3.2.1, em especial no que concerne ao suporte à semântica dos delimitadores $<$ e $>$. Como o predicado encontra-se em sua forma normal disjuntiva (FND), deve-se garantir que pelo menos uma das condições que constituem as disjunções seja verdadeiro. Isso é caracterizado pelo depósito de uma marca no lugar p^r. O lugar $\overline{p^r}$ evita que mais de uma marca seja depositada neste lugar, uma vez que a condição precisa ser testada uma única vez.

5.2.8.3 Habilitação de $\mathbf{t^e}$ A condição de disparo da transição t^e surge a partir da percepção de que é suficiente existir pelo menos uma marca em algum lugar $flag_k$, $d \leq k \leq n$ para que, com base na Equação 5.16, garanta-se que a transição t^f não seja habilitada.

Observe que os lugares $p_{(k,j,i)}$ somente serão mapeados aos lugares em $\overline{\text{LOOP\_FLAGS}_{(k,j)}}$ quando na aplicação da função Υ^{end_sync}, empregando o artifício de qualificação para este propósito.

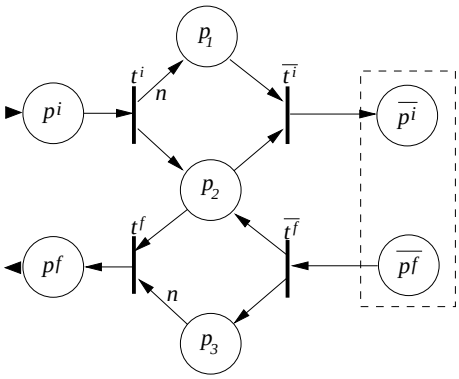


Figura 5.11. Ações Repetidas por Número Fixo de Iterações (**repeat counter**)

$$
\begin{gathered}
\Upsilon^{action}(\textbf{repeat\_counter}\ (a,n),\Pi,d) = (P \cup P^{+}, T \cup T^{+}, A \cup A^{+}, \rho^{+}, p^{i}, p^{f}) \\
\textbf{onde:} \\
(P,T,A,\rho,\overline{p^{i}},\overline{p^{f}}) = \Upsilon^{action}(a,\Pi,d+1) \\
P^{+} = \{p^{i}, p^{f}, p_1, p_2, p_3\} \\
T^{+} = \{t^{i}, t^{f}, \overline{t^{i}}, \overline{t^{f}}\} \\
A^{+} = \left\{ \begin{array}{c} (p^{i},t^{i}), (t^{i},p_1,n), (t^{i},p_2), (p_1,\overline{t^{i}}), (p^{2},\overline{t^{i}}), (\overline{t^{i}},\overline{p^{i}}) \\ (\overline{p^{f}},\overline{t^{f}}), (\overline{t^{f}},p_2), (\overline{t^{f}},p_3), (p_2,t^{f}), (p_3,t^{f},n), (t^{f},p^{f}) \end{array} \right\} \\
\rho^{+}(t) = \left\{ \begin{array}{ll} \rho(t) & , \text{se } t \in T \\ \lambda & , \text{se } t \in T^{+} \end{array} \right.
\end{gathered}
\tag{5.17}
$$

5.2.9 Repetição por Quantidade Fixa de Iterações (repeat ... counter N)

A tradução do combinador **repeat**, com condição de terminação especificada pela cláusula **counter**, é ilustrada na Figura 5.11 e formalizada na Regra 5.17. A cláusula **counter** especifica um número de fixo de iterações (n) para execução repetida de uma ação a.

Na rede da Figura 5.11, os arcos (t^i, p_1) e (p_3, t^f) possuem peso n, o qual define o número de vezes que a rede induzida pela ação a será ativada, com o depósito de uma marca em seu lugar inicial $\overline{p^i}$.

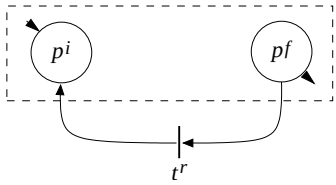


Figura 5.12. Repetição Infinita (**repeat**)

$$
\begin{gathered}
\Upsilon^{action}(\textbf{repeat\_forever}\ a,\Pi,d) = (P, T \cup T^{+}, A \cup A^{+}, \rho^{+}, p^{i}, p^{f}) \\
\textbf{onde:} \\
(P,T,A,\rho,p^{i},p^{f}) = \Upsilon^{action}(a,\Pi,d+1) \\
T^{+} = \{t^{r}\} \\
A^{+} = \{(p^{f},t^{r}), (t^{r},p^{i})\} \\
\rho^{+}(t) = \left\{ \begin{array}{ll} \rho(t) & , \text{se } t \in T \\ \lambda & , \text{se } t \in T^{+} \end{array} \right.
\end{gathered}
\tag{5.18}
$$

5.2.10 Execução Repetida Infinita de Ações (repeat)

A tradução do combinador **repeat**, sem uma condição de terminação especificada, é ilustrada na Figura 5.12 e formalizada na Regra 5.18. A ação a é executada repetidamente e sequencialmente, sem que uma condição de terminação seja alcançada.

5.2.11 Primitivas de Semáforos (wait e signal)

As primitivas **wait** e **signal** sobre semáforos contadores são usadas na linguagem # para aumentar seu poder expressivo, tornando essa linguagem capaz de expressar qualquer comportamento modelável por uma rede de Petri rotulada terminal[126]. A semântica de semáforos contadores é definida em uma linguagem concorrente da seguinte forma:

$wait(s)$: <**wait** $s > 0$; $s \rightarrow s - 1$>
$signal(s)$: $< s \rightarrow s + 1 >$

O código acima obedece a sintaxe introduzida em [6] para programação concorrente. Os delimitadores < e > encapsulam operações executadas atomicamente (exclusão mútua), enquanto o construtor **wait** representa uma primitiva de sincronização condicional preemptiva. Em um sistema concorrente controlado por semáforos 0-contador, o valor do semáforo no final da execução deve ser o mesmo valor inicial. Assim, em qualquer seqüência gerada por uma expressão concorrente sincronizada com semáforos 0-contador, para cada n ocorrências da ação *wait*, devem existir uma sequência correspondente de n ocorrências da primitiva *signal*. Dessa forma, pode ser dito que para cada ocorrência de *wait*, deve existir uma ocorrência de *signal* que a precede.

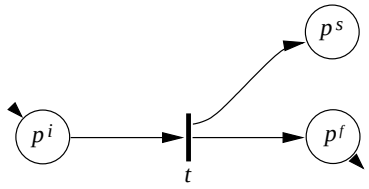


Figura 5.13. Primitiva **signal** (Regra 5.19)

$$\begin{gathered}\Upsilon^{action}(\textbf{signal}\ s, \_, \_) = (P, T, A, \rho, p^i, p^f) \\ \textbf{onde:} \\ P = \{p^i, p^f, p^s\} \\ T = \{t\} \\ A = \{(p^i, t), (t, s), (t, p^t)\} \\ \rho(t) = \lambda \\ qualifier(p^s) = (\textsc{Sem}, s)\end{gathered} \tag{5.19}$$

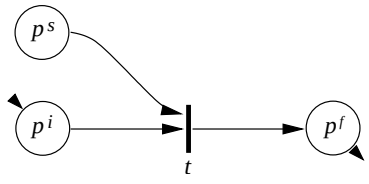


Figura 5.14. Primitiva **wait** (Regra 5.20)

$$\begin{gathered}\Upsilon^{action}(\textbf{wait } s, \_, \_) = (P, T, A, \rho, p^i, p^f) \\ \textbf{onde:} \\ P = \{p^i, p^f, p^s\} \\ T = \{t\} \\ A = \{(p^i, t), (s, t), (t, p^f)\} \\ \rho(t) = \lambda \\ qualifier(p^s) = (\textsc{Sem}, s)\end{gathered} \tag{5.20}$$

A modelagem com redes de Petri das primitivas **signal** e **wait** é formalizada através das Regras 5.19 e 5.20, sendo ilustradas nos diagramas 5.13 e 5.14, respectivamente. Seja na ocorrência da primitiva **signal** ou da primitiva **wait**, o lugar p^s modela o semáforo em questão. O artifício de qualificação é empregado para unificar os lugares p^s induzidos por cada ocorrência destas primitivas sobre um mesmo semáforo.

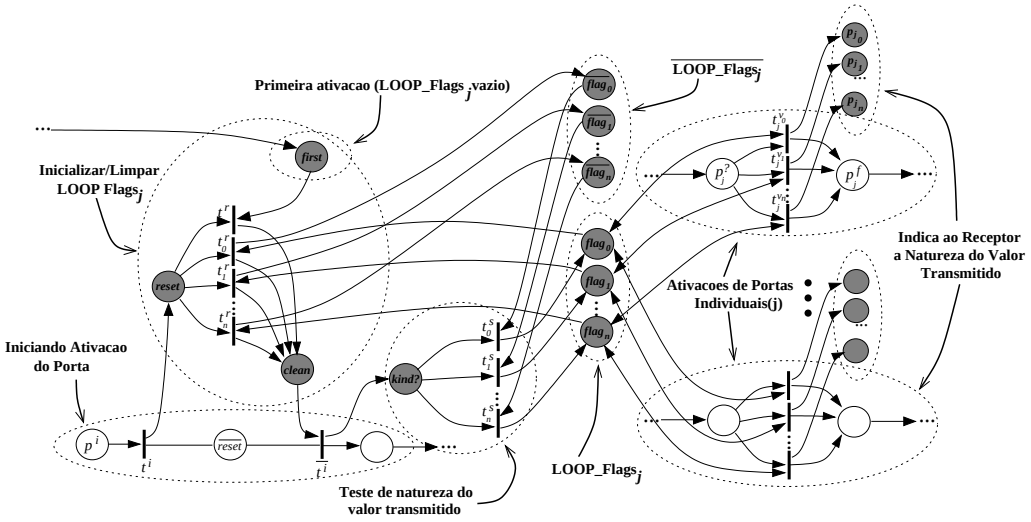


Figura 5.15. Sincronização de Natureza de Valor Transmitido em Portas de Saída

5.2.12 O Protocolo de Controle da Terminação Sincronizada de Streams

Na definição da função $\Upsilon^{program}$ (Regra 5.1), a função Υ^{end_sync} é aplicada sobre a rede de Petri que caracteriza o comportamento do programa com a finalidade de inserir nesta um protocolo para sincronização da natureza do valor transmitido como efeito de ativações de portas. Este se faz necessário para possibilitar a modelagem da semântica de terminação de ocorrências do combinador **repeat**, como descrito na Seção 5.2.8, atendendo ainda a restrições impostas pela semântica de programas #, relatadas a seguir:

i) Na transmissão de valores entre portas *stream*, a natureza do valor enviado pela porta emissora deve coincidir com a natureza do valor recebido pela porta receptora. Portanto, observando-se a rede induzida pela ativação de portas individuais (Figura 5.4), supondo-se n o aninhamento da porta, deve-se garantir que se a transição t^{v_k}, $0 \leq k \leq n$, é disparada na ativação da porta de saída, como resultado do

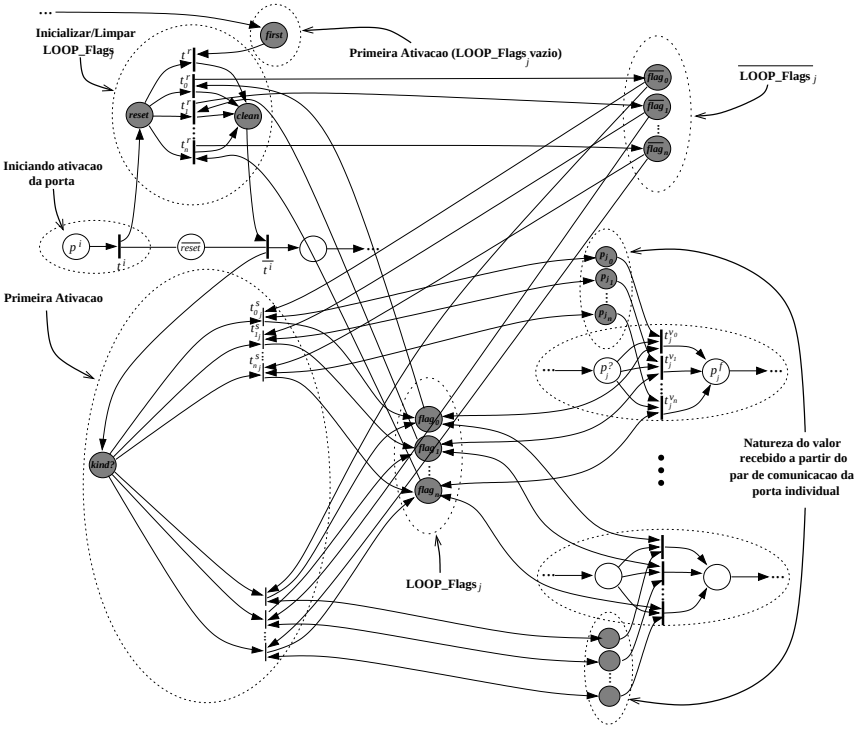


Figura 5.16. Sincronização de Natureza de Valor Transmitido em Portas de Entrada

conflito em $p^?$, então, na ativação da porta de entrada que constitui seu par na comunicação, a transição t^{v_k}, correspondente, deve também ser disparada. Como discutido anteriormente, convenciona-se que o disparo das transições de t^{v_k}, $0 \leq k \leq n-1$, indica a transmissão de valores indicadores de final de lista em aninhamento k, enquanto o disparo de t^{v_n} indica a transmissão de um item de dados;

ii) Todas as portas pertencentes a um mesmo agrupamento devem transmitir (enviar ou receber) valores de mesma natureza em uma certa ativação;

iii) A natureza do valor transmitido é guiado pela porta de saída, uma vez que é o módulo funcional do processo emissor que produz o valor transmitido. Portanto, em um programa # é possível que na ativação de um agrupamento de portas de entrada, algumas portas individuais aninhadas leiam valores de natureza diferente, estado que caracteriza um erro em tempo de execução. O mesmo não ocorre em portas de saída, pelo motivo inicialmente exposto;

As Equações 5.22 e 5.23 definem a função Υ^{end_sync}, quando aplicada a ativações de portas de saída e de entrada, respectivamente. A constituição da rede que define o protocolo difere ligeiramente para cada caso, devido ao atendimento à restrição (iii), como pode ser observado nas Figuras 5.15 e 5.16, respectivamente. Indiferente do caso considerado, para cada porta i devem existir os conjuntos de lugares LOOP_FLAGS_i e $\overline{\text{LOOP\_FLAGS}_i}$, como descrito na Seção 5.2.8. Estes são usados para lembrar a natureza

do valor transmitido na mais recente ativação da porta i, sendo portanto o atendimento às suas restrições (Equações 5.15 e 5.16) a razão da existência do protocolo descrito a seguir.

No início da ativação de uma porta *stream* i, com aninhamento n, é necessário restaurar a marcação dos lugares LOOP_FLAGS_i e $\overline{\text{LOOP\_FLAGS}_i}$ ao seu estado inicial, o qual atende a seguinte restrição, assumindo a obediência à propriedade de exclusão mútua especificada na Equação 5.16:

$$\sum_{k=0}^{n} flag_k = 0 \wedge \sum_{k=0}^{n} \overline{flag_k} = n \tag{5.21}$$

$$\begin{gathered}
\Upsilon^{end_sync}(\pi, Channels, PN) = (P \cup P^+, T \cup T^+, A \cup A^+, \rho^+, p^i, p^f), \\
\textbf{onde:} \\
\textbf{port}\ (pid, \textbf{output}, *, *, n) \simeq \pi \\
\{pid_pair\} = \{i \mid \textbf{connect}\ (pid, i, *) \in Channels\} \\
\{pid_1, \ldots, pid_m\} = nested_groups_pids\ (\pi) \\
(P, T, A, \rho, p^i, p^f) = PN \\
T^v_{j\ k} = \{t \mid t \in T \wedge qualifier(t) \simeq (pid_j, k, \text{SYNC_END}, *)\},\ \forall k, j : 1 \leq k \leq n, j : 1 \leq j \leq m \\
\begin{array}{l} T^i = \{t \mid t \in T \wedge qualifier(t) \simeq (pid, \text{BEGIN_ACTIVATION}, *)\} \\ \overline{T^i} = \{t \mid t \in T \wedge qualifier(t) \simeq (pid, \overline{\text{BEGIN_ACTIVATION}}, *)\} \end{array} \\
P^+ = \{first, reset, \overline{reset}, clean, kind?\} \cup \left(\bigcup_{k=0}^{n} \{flag_k, \overline{flag_k}\} \cup \left(\bigcup_{k=0}^{n} \bigcup_{j=1}^{m} \{p_{j\,k}\} \right) \right) \\
T^+ = \{t^r\} \cup \left(\bigcup_{k=0}^{n} \{t^r_k, t^s_k\} \right)
\end{gathered} \tag{5.22}$$

$$A^+ = \left\{ \begin{array}{l} (t^i, reset), \\ (t^i, \overline{reset}), \\ (\overline{reset}, \overline{t^i}), \\ (reset, t^r), \\ (first, t^r), \\ (clean, \overline{t^i}), \\ (\overline{t^i}, kind?) \\ (t^r, clean) \end{array} \quad \mid (t^i, \overline{t^i}) \in T^i \times \overline{T^i} \right\} \cup \left(\bigcup_{k=0}^{n} \left\{ \begin{array}{l} (reset, t^r_k), \\ (t^r_k, clean), \\ (kind?, t^s_k), \\ (flag_k, t^r_k), \\ (t^r_k, \overline{flag_k}), \\ (\overline{flag_k}, t^s_k), \\ (t^s_k, flag_k) \end{array} \right\} \cup \left(\bigcup_{j=1}^{m} \left\{ \begin{array}{l} (flag_k, t), \\ (t, flag_k), \\ (t, p_{j\,k}) \end{array} \quad \mid t \in T^v_{j\ k} \right\} \right) \right)$$

$$\rho^+(t) = \left\{ \begin{array}{ll} \rho(t) & \text{se } t \in T \\ \lambda & \text{se } t \in T^+ \end{array} \right.$$

$$\begin{gathered}
qualifier(t^s_k) = (pid, k, \text{TCLEAN}),\ \forall k : 0 \leq k \leq n \\
qualifier(t^r_k) = (pid, k, \text{TSET}),\ \forall k : 0 \leq k \leq n
\end{gathered}$$

$$\begin{gathered}
qualifier(\underline{flag_k}) = (\text{LOOP_FLAGS}, pid, k),\ \forall k : 0 \leq k \leq n \\
qualifier(\overline{flag_k}) = (\overline{\text{LOOP_FLAGS}}, pid, k),\ \forall k : 0 \leq k \leq n
\end{gathered}$$

$$qualifier(p_{j\,k}) = (pid, pid\_pair)$$

A restrição indica que todos os lugares em $\overline{\text{LOOP\_FLAGS}_i}$ devem possuir uma única marca, enquanto todos os lugares em LOOP_FLAGS_i devem conter nenhuma marca. O reajuste inicia com o depósito de uma marca no lugar *reset*. A transição $\overline{t^i} \in \overline{T^i}$ permanece inabilitada até que um lugar seja depositado no lugar *clean*, quando a restrição imposta pela Equação 5.21 encontra-se satisfeita. O lugar $first$ e a transição t^r são necessários na primeira ativação, quando LOOP_FLAGS_i não contém marca alguma. Uma transição t^r_k, $1 \leq k \leq n$, é disparada quando existe uma marca no lugar $flag_k$, surtindo o efeito da

retirada desta marca e o depósito de uma outra no lugar $\overline{flag_k}$ correspondente, o qual, pela garantia à restrição na Equação 5.16, deve estar vazio.

A diferença entre as duas redes se caracteriza quando analisamos o momento em que a natureza do valor transmitido é verificado pela porta para que a marcação de LOOP_FLAGS$_i$ seja atualizada.

$$
\begin{gathered}
\Upsilon^{end\_sync}(\pi, Channels, PN) = (P \cup P^+, T \cup T^+, A \cup A^+, \rho^+, p^i, p^f), \\
\textbf{onde:} \\
\textbf{port}\ (pid, \textbf{input}, *, *, n) \simeq \pi \\
\{pid\_pair\} = \{i \mid \textbf{connect}\ (i, pid, *) \in Channels\} \\
\{pid_1, \ldots, pid_m\} = nested\_groups\_pids\ (\pi) \\
(P, T, A, \rho, p^i, p^f) = PN \\
T^v_{j\ k} = \{t \mid t \in T \wedge qualifier(t) \simeq (pid_j, k, \textsc{Sync\_End}, *)\},\ \forall k : 1 \leq k \leq n \\
\begin{array}{c} T^i = \{t \mid t \in T \wedge qualifier(t) \simeq (pid, \textsc{Begin\_Activation}, *)\} \\ \overline{T^i} = \{t \mid t \in T \wedge qualifier(t) \simeq (pid, \overline{\textsc{Begin\_Activation}}, *)\} \end{array} \\
P^+ = \{first, reset, \overline{reset}, clean, kind?\} \cup \left(\bigcup_{k=0}^{n} \{flag_k, \overline{flag_k}\} \right) \cup \left(\bigcup_{k=0}^{n} \bigcup_{j=1}^{m} \{p_{k\,j}\} \right) \\
T^+ = \{t^r, \overline{t^i}\} \cup \left(\bigcup_{k=0}^{n} \{t^r_k\} \right) \cup \left(\bigcup_{k=0}^{n} \bigcup_{j=1}^{m} \{t^s_{k_j}\} \right) \\
A^+ = \left\{ \begin{array}{c} (t^i, reset), \\ (t^i, \overline{reset}), \\ (\overline{reset}, \overline{t^i}), \\ (reset, t^r), \\ (first, t^r), \\ (clean, \overline{t^i}), \\ (\overline{t^i}, kind?), \} \\ (t^r, clean) \end{array} \mid (t^i, \overline{t^i}) \in T^i \times \overline{T^i} \right\} \cup \left(\bigcup_{k=0}^{n} \left\{ \begin{array}{c} (reset, t^r_k), \\ (t^r_k, clean), \\ (kind?, t^s_k), \\ (flag_k, t^r_k), \\ (t^r_k, \overline{flag_k}) \end{array} \right\} \cup \left(\bigcup_{j=1}^{m} \left\{ \begin{array}{c} (\overline{flag_k}, t^s_{k_j}), \\ (t^s_{k_j}, flag_k), \\ (t^s_{k_j}, p_{j\,k}), \\ (p_{j\,k}, t^s_{k_j}), \\ (kind?, t^s_{k_j}) \end{array} \right\} \cup \left\{ \begin{array}{c} (p_{j\,k}, t), \\ (flag_k, t), \\ (t, flag_k) \end{array} \mid t \in T^v_{j\,k} \right\} \right) \right) \\
\rho^+(t) = \left\{ \begin{array}{ll} \rho(t) & \text{se } t \in T \\ \lambda & \text{se } t \in T^+ \end{array} \right. \\
qualifier(flag\_k) = (\textsc{Loop\_Flags}, pid, k),\ \forall k : 0 \leq k \leq n \\
qualifier(\overline{flag\_k}) = (\overline{\textsc{Loop\_Flags}}, pid, k),\ \forall k : 0 \leq k \leq n \\
qualifier(p_{j\,k}) = (pid\_pair, pid)
\end{gathered}
\tag{5.23}
$$

Na ativação de portas de saída, a natureza do valor que será transmitido é decidida antes do envio. Caso a porta seja um agrupamento, todas as portas individuais a esta aninhadas devem transmitir um valor de mesma natureza. A escolha da natureza do valor, neste caso, ocorre pelo conflito no lugar $kind?$. O disparo da transição t^s_k, $0 \leq k \leq n$, causa o depósito de uma marca no lugar $flag_k$ e a retirada da marca contida no lugar $\overline{flag_k}$. Uma vez que as n+1 transições t^s_k são disparadas de forma mutuamente exclusiva, garante-se o atendimento às restrições 5.15 e 5.16. Posteriormente, na ativação de qualquer porta individual aninhada ao grupo em qualquer nível, é disparada uma das transições em T^k_j correspondente. Assim garante-se que todas as portas do grupo compartilhem a mesma natureza para o valor transmitido. Com a finalidade de obrigar que a porta receptora receba um valor de mesma natureza, uma marca é depositada no lugar p_{jk} após o disparo de uma das transições em T^k_j. Após o final da ativação da porta (agrupamento ou não), a natureza do valor transmitido permanece salva em LOOP_FLAGS$_i$ até que sua marcação inicial seja restaurada no início da próxima ativação. Dessa forma, é possível testar

a natureza do último valor transmitido por uma porta ao verificar a condição de uma repetição que faz referência a esta, por exemplo.

No caso de uma porta individual de entrada, a natureza do valor transmitido é verificada de acordo com a marcação dos lugares p_{j_k}, $0 \leq k \leq n$, de seu par na comunicação. Para um agrupamento, deve-se considerar os lugares p_{j_k} de cada porta individual aninhada. Seja qual for o caso, cada lugar p_{j_k} deve estar ligado por dois arcos (ida e volta) com a transição de t_k^s, na rede induzida pela ativação em questão. Além disso, deve haver um arco de p_{j_k} para as transições em T_j^k, na sub-rede que indica a ativação de uma porta individual. Com isso um *deadlock* ocorre caso portas individuais de um mesmo agrupamento recebam valores de natureza diferente, evento cujo acontecimento é possível durante a execução de um programa #, porém causando um erro em tempo de execução. Entretanto, é facilmente possível extender esta rede de forma a testar se o valor recebido pelas portas tem natureza distinta, assumindo-se neste caso uma marcação que caracterize um erro em tempo de execução.

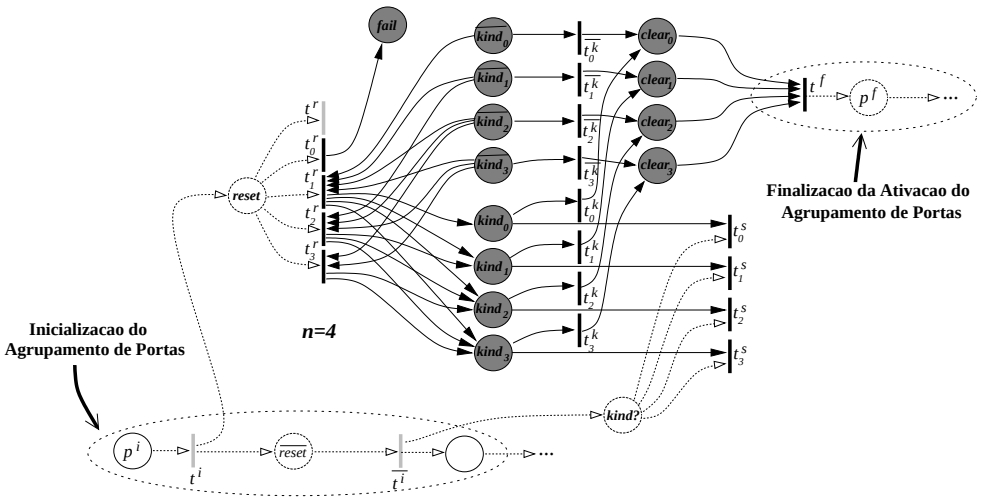


Figura 5.17. Restrição de Ordem para a Natureza do Valor Transmitido na Ativação de uma Porta

5.2.13 Modelando a Ordem da Natureza dos Valores Transmitidos

Considere uma lista aninhada Haskell de inteiros, com fator de aninhamento 4 (tipo `[[[[Int]]]]`):

```
[[[[1],[5,6]],[[2,3]]],[],[[[[4,5,7],[8,9]]],[[6],[7,9]]]]
```

Uma *stream* de fator de aninhamento 4 associada a lista acima transmitirá os seguintes valores:

```
{1,Eos 3, 5, 6,Eos 3,Eos 1, 2, 3,Eos 3,Eos 2,Eos 1,Eos 1, 4, 5, 7,
 Eos 3, 8, 9,Eos 3,Eos 2,Eos 1, 6,Eos 3, 7, 9,Eos 3,Eos 2,Eos 1,Eos 0}
```

Observe-se que ao transmitir-se um valor EOS 2, não é possível que na próxima ativação ocorra a transmissão de um valor EOS 0, de maneira que a *stream* em aninhamento 1 a qual pertence a *stream* de aninhamento 2 finalizada não seja também finalizada. Neste exemplo, os valores de dados são considerados de grau de aninhamento 4. Portanto após a transmissão de um valor de dados somente é possível ser transmitido outro valor de dados ou um valor END 3, finalizando a *stream* aninhada.

O exemplo anterior ilustra outro importante aspecto possível de ser modelado com respeito à natureza dos valores transmitidos através de *streams*. Este consiste na seguinte restrição de ordem. Considere uma *stream* s, com fator de aninhamento n. Neste caso, o grau de aninhamento do valor de dados é n. Seja k o grau de aninhamento do valor transmitido na mais recente ativação da porta p que transmite s. Se $k < n$, na próxima ativação somente poderão ser transmitidos valores com grau de aninhamento menores ou iguais a $k + 1$. Se $k = n$, então a ativação da porta deve causar um erro em tempo de execução, uma vez que a *stream* já atingiu seu final e portanto não há mais valor a ser transmitido pela porta. Em outras palavras, essa restrição modela o seguinte fato: quando uma *stream* finaliza no aninhamento k, as *streams* as quais esta encontra-se aninhada devem ser todas finalizadas em sequência e ordenadamente $(k+1, k+2, \ldots, n)$.

A função Υ^{order_end}, apresentada na Regra 5.24, introduz na rede de Petri resultante da aplicação da função Υ^{sync_end} a restrição descrita nos parágrafos anteriores. O diagrama da Figura 5.17 ilustra o efeito da aplicação da função Υ^{order_end}.

$$
\begin{gathered}
\Upsilon^{end\_order}(\mathbf{port}(pid, *, *, *, n), PN) = (P \cup P^+, T \cup T^+, A \cup A^+, \rho^+, p^i, p^f), \\
\mathbf{onde:} \\
(P, T, A, \rho, p^i, p^f) = PN \\
\{t^f\} = \{t \mid t \in T \wedge (t, p^f) \in A\} \\
P^+ = \{fail\} \cup \left(\bigcup_{k=0}^{n} \{kind_k, \overline{kind_k}, clean_k\} \right) \\
T^+ = \bigcup_{k=0}^{n} \{t_k^r, t_k^s, t_k^c\} \\
A^+ = \{(t_r^0, fail)\} \cup \left(\bigcup_{k=0}^{n} \left\{ \begin{array}{l} (kind_k, t_k^s), (kind_k, t_k^c), \\ (t_k^c, clear_k), (\overline{kind_k}, t_k^c), \\ (t_k^c, clear_k), (clear_k, t^f) \end{array} \right\} \right) \cup \left(\bigcup_{k=1}^{n} \bigcup_{j=k-1}^{n} \left\{ \begin{array}{l} (t_k^r, kind_j), \\ (\overline{kind_j}, t_k^r) \end{array} \right\} \right) \\
qualifier(t_k^s) = (pid, k, \text{TCLEAN}), \ \forall k : 0 \leq k \leq n \\
qualifier(t_k^r) = (pid, k, \text{TSET}), \ \forall k : 0 \leq k \leq n \\
qualifier(fail) = \text{FAIL} \\
\rho^+(t) = \left\{ \begin{array}{ll} \rho(t) & \text{se } t \in T,\ 1 \leq k \leq n \\ \lambda & \text{se } t \in T \end{array} \right.
\end{gathered}
\tag{5.24}
$$

Seja j, $0 \leq j \leq n$, a natureza do último valor transmitido por uma porta *stream* arbitrária, denominada p. No início de uma nova ativação de p, a presença de uma marca no lugar *reset* (ver Regra 5.22) habilita o disparo da transição t_j^r, cujo efeito é retirar a marca que deve estar contida no lugar $flag_j$. Na marcação resultante, a soma das marcas no conjunto de lugares LOOP_FLAGS$_j$ é zero, pré-requisito para iniciar-se o protocolo de caracterização da natureza do valor transmitido na ativação em curso. Se $j > 0$, a sub-rede introduzida pela aplicação da função Υ^{end_order} força que o disparo

da transição t_j^r cause ainda o depósito de marcas nos lugares $kind_i$, $j-1 \leq i \leq n$. Caso contrário ($j = 0$), um estado de falha é assumido com o depósito de uma marca no lugar $fail$ (a *stream* já finalizou na ativação anterior). A presença de marcas nos lugares $kind_j$, $0 \leq j \leq n$, caracteriza o conjunto de naturezas que o próximo valor que será transmitido pela porta poderá assumir, em função da natureza do valor transmitido na ativação anterior. Com essa finalidade, esses lugares são usados para guardar as transições t_j^s, $0 \leq j \leq n$, correspondentes, as quais modelam a decisão a cerca a natureza do valor transmitido na ativação. Dessa forma, para o caso em questão, somente uma dentre as transições t_i^s, $j-1 \leq i \leq n$ estarão habilitadas, o que caracteriza o atendimento à restrição que estabelece a ordem das naturezas para os valores transmitidos, descrito no início desta seção. Os lugares $\overline{kind_j}$, $0 \leq j \leq n$ são necessários para permitir o esvaziamento dos lugares $kind_j$ correspondentes ao final da ativação da porta. Ainda com essa finalidade, são empregadas os conjuntos de transições t_j^k, $\overline{t_j^k}$ e lugares $clear_j$.

5.2.14 Modelando a Sincronização de Processos (Comunicação)

A definição da função Υ^{action}, detalhada nas seções anteriores, mostra como o comportamento individual de cada processo pode ser modelado com redes de Petri. É ainda necessário modelar a semântica de sincronização de processos por meio dos canais de comunicação que conectam suas portas, assumindo-se seus três modos suportados: **synchronous**, **buffered** e **ready**, correspondentes aos modos de comunicação suportados por MPI. Nesta seção, mostraremos como modelar a informação de sincronização provida por canais de comunicação sobre a rede de Petri induzida pelo programa.

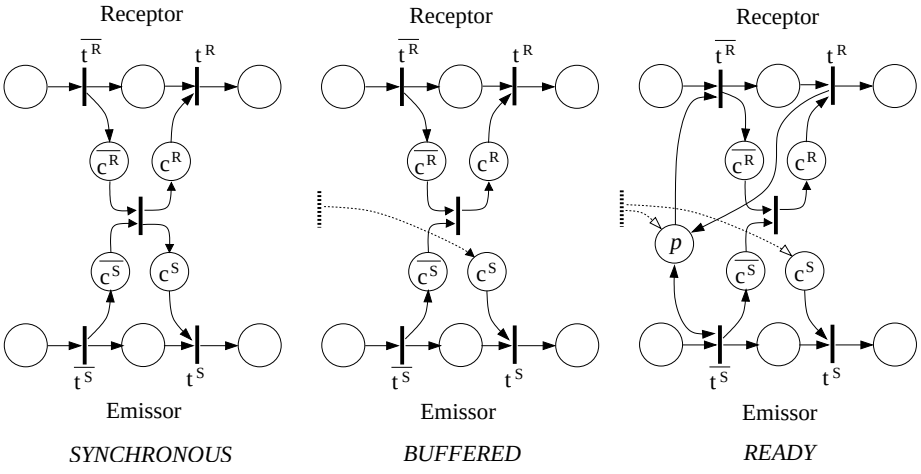


Figura 5.18. Modelagem de Canais de Comunicação

As Regras 5.25, 5.26 e 5.27 definem a função $\Upsilon^{channel}$, aplicada na Regra 5.1 para inserir a informação de sincronização contida na definição de canais de comunicação. O diagrama na Figura 5.18 ilustra essas regras. Respectivamente, as regras modelam canais de cada modo de comunicação suportado: **synchronous**, **buffered** e **ready**.

$$
\begin{gathered}
\Upsilon^{channel}(\textbf{connect}(s, r, \textbf{synchronous}), PN) = (P \cup P^+, T \cup T^+, A \cup A^+, \rho^+, p^i, p^f) \\
\textbf{onde:} \\
(P, T, A, \rho, p^i, p^f) = PN \\
P^+ = \{c^R, \overline{c^R}, c^S, \overline{c^S}\} \\
T^+ = \{t\} \\
A^+ = \{(\overline{c^R}, t), (\overline{c^S}, t), (t, c^R), (t, c^S)\} \\
\begin{array}{l} qualifier(c^R) = (r, \text{POK}) \\ qualifier(c^S) = (s, \text{POK}) \\ qualifier(\overline{c^R}) = (r, \overline{\text{POK}}) \\ qualifier(\overline{c^S}) = (s, \overline{\text{POK}}) \end{array} \\
\rho^+(t) = \left\{ \begin{array}{ll} \rho(t) & \text{se } t \in T \\ \lambda & \text{se } t \in T^+ \end{array} \right.
\end{gathered}
\tag{5.25}
$$

No modo síncrono (*blocking*) (Regra 5.25), o disparo da transição t modela a efetivação de uma operação de comunicação em um certo canal c_i. A presença de uma marca no lugar $\overline{c^S}$ modela o fato de que a porta de saída está ativada, enquanto a presença de uma marca no lugar $\overline{c^R}$ modela o fato de que a porta de entrada está ativada. Logo, a comunicação só poderá completar-se uma marca encontra-se depositada em cada um desses lugares (*rendezvous*), possibilitando o disparo da transição t. Os pares de lugares c^R/c^S e $\overline{c^R}/\overline{c^S}$ correspondem respectivamente a pares de lugares $p^{Ok}/\overline{p^{Ok}}$ introduzidos pela Regra 5.5.

$$
\begin{gathered}
\Upsilon^{channel}(\textbf{connect}(s, r, \textbf{buffered}), PN) = (P \cup P^+, T, A \cup A^+, \rho^+, p^i, p^f) \\
\textbf{onde:} \\
(P, T, A, \rho, p^i, p^f) = PN \\
P^+ = \{c^R, \overline{c^R}, c^S, \overline{c^S}\} \\
T^+ = \{t\} \\
A^+ = \{(\overline{c^R}, t), (\overline{c^S}, t), (t, c^R)\} \\
\begin{array}{l} qualifier(c^R) = (r, \text{POK}) \\ qualifier(c^S) = (s, \text{POK}) \\ qualifier(\overline{c^R}) = (r, \overline{\text{POK}}) \\ qualifier(\overline{c^S}) = (s, \overline{\text{POK}}) \end{array} \\
\rho^+(t) = \left\{ \begin{array}{ll} \rho(t) & \text{se } t \in T \\ \lambda & \text{se } t \in T^+ \end{array} \right.
\end{gathered}
\tag{5.26}
$$

No modo *bufferizado* (Regra 5.26), o lugar c^S possuirá uma marca desde o disparo da transição que caracteriza o início da ativação da porta, ou agrupamento de portas aonde a porta encontra-se aninhada, até o disparo da transição que caracteriza a finalização de sua ativação. Em agrupamentos de portas, por exemplo, estas transições correspondem respectivamente a t^i e t^f nas Regras 5.6 e 5.5. Assim, a porta de saída, quando ativada, pode completar a transmissão de um dado sem necessidade de aguardar a ativação da porta de entrada que constitui seu par na configuração do canal.

$$\Upsilon^{channel}(\mathbf{connect}(s,r,\mathbf{ready}),PN) = (P \cup P^+, T, A \cup A^+, \rho^+, p^i, p^t)$$

$$\mathbf{onde:}$$

$$(P,T,A,\rho,p^i,p^f) = PN$$

$$P^+ = \{c^R, \overline{c^R}, c^S, \overline{c^S}, p\}$$

$$\begin{array}{c} qualifier(c^R) = (r, \text{POK}) \\ qualifier(c^S) = (s, \text{POK}) \\ qualifier(\overline{c^R}) = (r, \overline{\text{POK}}) \\ qualifier(\overline{c^S}) = (s, \overline{\text{POK}}) \end{array}$$

$$T^+ = \{t\}$$

$$A^+ = \{(\overline{c^R}, t), (\overline{c^S}, t), (t, c^R), (t^i, p), (p, t^f)\} \cup \left(\bigcup_{k=1}^{s} \{(t\overline{^S_k}, p), (p, \overline{t^S_k})\} \right) \cup \left(\bigcup_{k=1}^{r} \{(p, \overline{t^R}), (t^R, p)\} \right) \quad (5.27)$$

$$\begin{array}{c} \{t^i\} = \{t \mid (p^i, t) \in A\} \\ \{t^f\} = \{t \mid (t, p^f) \in A\} \end{array}$$

$$\begin{array}{c} \{\overline{t^S}_1, \ldots, \overline{t^S}_s\} = \{t \mid (t, \overline{c^S}) \in A \cup A^+\} \\ \{t^R_1, \ldots, t^R_r\} = \{t \mid (c^R, t) \in A \cup A^+\} \\ \{\overline{t^R}_1, \ldots, \overline{t^R}_r\} = \{t \mid (t, \overline{c^R}) \in A \cup A^+\} \end{array}$$

$$\rho^+(t) = \begin{cases} \rho(t) & \text{se } t \in T \\ \lambda & \text{se } t \in T^+ \end{cases}$$

No modo *ready* (Regra 5.27), um lugar, denominado p, é introduzido, inicializado com uma marca. Este garante que a operação de envio é sempre precedida da operação de recebimento em uma execução correta do sistema. A porta receptora, quando ativada, consome a marca do lugar p, permanecendo bloqueada até que exista uma marca no lugar c^R. Entretando, o receptor permanecerá bloqueado para sempre caso não tenha havido, em um momento anterior, uma ativação da porta emissora, pois a ativação desta só ocorre quando existe pelo menos uma marca no lugar p, estado que só ocorrerá após a efetivação da comunicação na porta receptora. (disparo da transição t_R).

5.3 TRADUZINDO ESQUELETOS MPI PARA REDES DE PETRI

Esqueletos # constituem uma ferramenta útil para simplificar a rede de Petri que modela o comportamento de um programas #, tornando visualmente mais simples e computacionalmente menos custosa a verificação de suas propriedades e avaliação de seu desempenho. Para demonstrar como esqueletos podem efetivamente ser usados com esse propósito, utilizaremos como exemplo os esqueletos MPI descritos na Seção 4.2.4, cuja implementação é apresentada na Seção 6.2.5.

Considere-se um conjunto de unidades e uma operação de comunicação coletiva que envolve portas que constituem suas interfaces. Como descrito no Capítulo 6, para que esqueletos MPI possam ser efetivamente usados, as interfaces dessas unidades devem incluir a interface do esqueleto MPI apropriado em sua composição. As portas herdadas da interface MPI não devem ser referenciadas em operações de ativação na descrição comportamental da interface, porém podem ser referenciadas em condições de terminação de ocorrências do combinador **repeat**. Na descrição do comportamento da interface, o combinador **do** é usado para descrever os pontos onde deve ser efetivada a operação de comunicação coletiva entre as portas das unidades envolvidas. Como discutido na Seção 3.2.1, este combinador encapsula o comportamento de um conjunto de portas herdadas

de uma interface. Uma operação de comunicação se completa quando todas as unidades "executam **do**" simultaneamente sobre as componentes de interface correspondentes em cada unidade. O artifício descrito foi extensivamente utilizado na construção do código # dos programas constituintes do *NAS Parallel Benchmarks*, como pode ser observado no Apêndice B. De maneira simples, o compilador # é capaz de gerar código específico correspondente ao padrão de iteração descrito pelo uso do esqueleto MPI, da forma descrita anteriormente, com chamadas diretas às primitivas de comunicação coletiva correspondentes suportadas por esta biblioteca.

Em uma operação coletiva, todas as portas envolvidas são ativadas simultaneamente pelo combinador **do**. Caso tais portas sejam do tipo *stream*, a natureza do valor transmitido como efeito da ativação deve ser a mesma para todas as portas. Portanto, as portas envolvidas em uma operação coletiva são funcionalmente equivalentes, sendo possível modelá-las por meio de uma única porta em *abstract #*. O combinador **do**, neste contexto, é interpretado como um combinador de ativação para portas dessa natureza. Assim, em *abstract #*, usamos o combinador **activate**, como usualmente, para representar o combinador **do** da linguagem # original. Além disso, qualquer referência a uma porta pertencente a uma interface MPI na condição de terminação de um **repeat** corresponde, em *abstract #*, a uma referência a porta (única) que representa as portas envolvidas na operação coletiva.

Utilizando a interpretação descrita, a ocorrência de uma aplicação de **do** sobre uma interface MPI em # original, corresponde a aplicação do combinador **activate** sobre a porta única que representa as portas desta interface, em *abstract #*. Nenhuma modificação ou extensão é portanto necessária às funções do esquema de tradução que lidam com a descrição comportamental de unidades. Porém, deverá existir uma forma explícita de estabelecer o relacionamento entre as portas envolvidas em uma mesma operação de comunicação coletiva modelada pelo esqueleto MPI. A extensão necessária a *abstract #* diz respeito a forma como portas envolvidas em uma operação coletiva são conectadas. Para isso extende-se o não terminal <CHANNEL> da descrição formal de *abstract #* com um novo construtor, **mpi_collective**, o qual relaciona um conjunto de portas que participam de uma mesma operação de comunicação coletiva descrita por um esqueleto MPI:

$$
\begin{array}{l}
\vdots \\
\text{<CHANNEL>} \rightarrow \textbf{connect}\ (id,\ id,\ \text{m:<CHANMODE>}) \\
\qquad\qquad\qquad\quad \textbf{mpi\_collective}\ \{id_1,\ id_2,\ \ldots,\ id_n\} \\
\vdots
\end{array}
$$

Uma nova equação deve extender a definição da função $\Upsilon^{channel}$ com a finalidade de modelar a semântica do construtor **mpi_collective**. Esta é apresentada na Regra 5.28 e ilustrada na Figura 5.19.

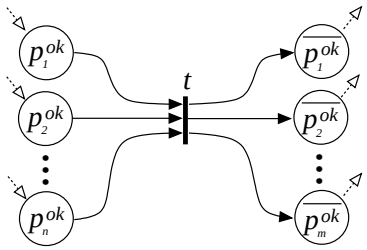


Figura 5.19. Traduzindo Esqueletos MPI para Redes de Petri

$$
\begin{gathered}
\Upsilon^{channel}(\textbf{mpi\_collective}\ (pid_1,\ldots,pid_m),PN) = (P\cup P^+, T\cup T^+, A\cup A^+, \rho^+, p^i, p^f),\\
\textbf{onde:}\\
(P,T,A,\rho,p^i,p^f) = PN\\
P^+ = \bigcup_{k=1}^{m}\{\overline{p_k}, p_k\}\\
T^+ = \{t\}\\
A^+ = \bigcup_{k=1}^{m}\{(\overline{p_k},t),(t,p_k)\}\\
\rho^+(t) = \begin{cases} \rho(t) & \text{se } t\in T\\ \lambda & \text{se } t\in T^+\end{cases}\\
qualifier(p_k) = (pid_k, \text{POK}), \forall k : 1\leq k\leq m\\
qualifier(\overline{p_k}) = (pid_k, \overline{\text{POK}}), \forall k : 1\leq k\leq m
\end{gathered}
\tag{5.28}
$$

É ainda necessário mostrar como o protocolo que controla a natureza do valor transmitido deve ser aplicado às portas conectadas por meio do construtor **mpi_collective**. Inicialmente, ao contrário das portas conectadas por **channel**, não há uma distinção entre portas de entrada e saída neste caso. Assim, é necessário extender *abstract #* para suportar a existência de portas não caracterizáveis como de entrada ou saída:

⋮
<DIRECTION> → **input** | **output** | **indifferent**
⋮

Todas as portas envolvidas em uma comunicação coletiva transmitem, simultaneamente, valores de mesma natureza. Partindo-se desta suposição, a modelagem do protocolo é bastante simplificada. Um mesmo conjunto de lugares LOOP_FLAGS$_j$ e $\overline{\text{LOOP\_FLAGS}_j}$ é compartilhado entre as portas envolvidas de todas as unidades. A escolha do valor transmitido pode ser feita por qualquer uma das portas. A Equação 5.29 define o protocolo de sincronização da natureza do valor transmitido para um conjunto de portas envolvidos em uma operação coletiva. Note que somente uma das portas é conectada a um protocolo semelhante aquele descrito na Seção 5.22. As demais apenas são usadas para determinar o qualificador dos lugares em LOOP_FLAGS$_j$ e $\overline{\text{LOOP\_FLAGS}_j}$, os quais são compartilhados entre as portas.

$$\Upsilon^{end\_sync}(\pi, Channels, PN) = \begin{cases} (P \cup P^+, T \cup T^+, A \cup A^+, \rho^+, p^i, p^f) & , pid = pid^{pivot} \\ (P, T, A, \rho, p^i, p^f) & , \text{caso contrário} \end{cases}$$

onde:

$$\textbf{port}\ (pid, \textbf{indifferent}, *, *, n) = \pi$$

$$\{pid^{pivot}\} = \{pid_1 \mid \textbf{mpi\_collective}\ \{pid_1, \ldots, pid_n\} \in Channels \wedge pid \in \{pid_1, \ldots, pid_n\}\}$$

$$(P, T, A, \rho, p^i, p^f) = PN$$

$${T^v}_k = \{t \mid t \in T \wedge qualifier(t) \simeq (pid_j, k, \textsc{Sync\_End}, *)\},\ \forall k : 1 \leq k \leq n$$

$$\begin{array}{c} T^i = \{t \mid t \in T \wedge qualifier(t) \simeq (pid, \textsc{Begin\_Activation}, *)\} \\ \overline{T^i} = \{t \mid t \in T \wedge qualifier(t) \simeq (pid, \overline{\textsc{Begin\_Activation}}, *)\} \end{array}$$

$$P^+ = \{first, reset, \overline{reset}, clean, kind?\} \cup \left(\bigcup_{k=0}^{n} \{flag_k, \overline{flag_k}\} \right)$$

$$T^+ = \{t^r\} \cup \left(\bigcup_{k=0}^{n} \{t^r_k, t^s_k\} \right) \tag{5.29}$$

$$A^+ = \left\{ \begin{array}{c} (t^i, reset), \\ (t^i, \overline{reset}), \\ (\overline{reset}, \overline{t^i}), \\ (reset, t^r), \\ (first, t^r), \\ (clean, \overline{t^i}), \\ (\overline{t^i}, kind?) \\ (t^r, clean) \end{array} \mid (t^i, \overline{t^i}) \in T^i \times \overline{T^i} \right\} \cup \left(\bigcup_{k=0}^{n} \left\{ \begin{array}{c} (reset, t^r_k), \\ (t^r_k, clean), \\ (kind?, t^s_k), \\ (flag_k, t^s_k), \\ (t^r_k, \overline{flag_k}), \\ (\overline{flag_k}, t^s_k), \\ (t^s_k, flag_k) \end{array} \right\} \cup \left\{ \begin{array}{c} (flag_k, t), \\ (t, flag_k), \\ (t, p_{j\,k}) \end{array} \mid t \in {T^v}_{j\,k} \right\} \right)$$

$$\rho^+(t) = \begin{cases} \rho(t) & \text{se } t \in T \\ \lambda & \text{se } t \in T^+ \end{cases}$$

$$\begin{array}{c} qualifier(t^s_k) = (pid, k, \textsc{TClean}),\ \forall k : 0 \leq k \leq n \\ qualifier(t^r_k) = (pid, k, \textsc{TSet}),\ \forall k : 0 \leq k \leq n \end{array}$$

$$qualifier(flag\_k) = \bigcup_{j=1}^{m} \{(\textsc{Loop\_Flags}, pid_j, k)\},\ \forall k : 0 \leq k \leq n$$

$$qualifier(\overline{flag\_k}) = \bigcup_{j=1}^{m} \{(\overline{\textsc{Loop\_Flags}}, pid_j, k)\},\ \forall k : 0 \leq k \leq n$$

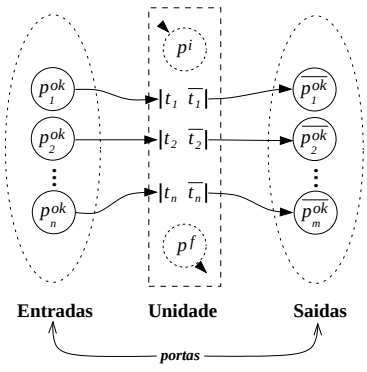


Figura 5.20. Visão Funcional de uma Unidade

5.4 PERSPECTIVA MODULAR DA REDE DE PETRI DE UM PROGRAMA

Uma forma conveniente de visualizar unidades # quando modeladas como redes de Petri, segundo a tradução descrita na seção anterior, é ilustrada na Figura 5.20. O conjunto de lugares p_i^{ok}, $1 \leq i \leq n$, modelam as portas (individuais) de entrada da unidade. A presença de marca em tais lugares indica o recebimento de valores provenientes de outros processos. Por outro lado, os lugares $\overline{p_i^{ok}}$, $1 \leq i \leq m$, modelam as portas (individuais) de saída da unidade. A presença de marcas nesses lugares indica a existência de valores que devem ser transmitidos pela unidade. A ordem em que as marcas nos lugares de entrada são consumidas e em que as marcas nos lugares de saída são produzidas é modelada pela linguagem formal induzida pela rede de Petri rotulada que descreve o comportamento da unidade, onde as transições t_i, $1 \leq i \leq n$, e $\overline{t_i}$, $1 \leq i \leq m$, são rotuladas com os nomes das portas de entrada e saída, respectivamente, cuja ativação cada uma destas modela.

Na Figura 5.21 é ilustrado como um programa #, o qual é descrito pela composição de um conjunto de unidades, pode ser enxergado a partir do esquema funcional de representação da rede de Petri que descreve suas unidades, da forma descrita no parágrafo anterior. Uma vez que canais de comunicação são as entidades que descrevem a interconexão das unidades que compõem o programa, através de suas portas, a transição t da Figura 5.2.14, induzida na tradução de cada canal de comunicação, descreve a conexão de um lugar de saída da unidade enviadora a um lugar de entrada da unidade receptora. Os lugares associados à portas mapeadas a pontos de entrada e saída do componente, por meio da declaração **bind**, constituirão os lugares de entrada e saída de um aglomerado associado ao componente em questão.

A forma modular de visualizar a rede de Petri de um programa # descrita anteriormente é efetivamente usada na geração de código PNML (*Petri Net Markup Language*)[229] na implementação do tradutor de programas # para redes de Petri. PNML tem sido proposta como um formato padrão para intercâmbio de arquivos que descrevem redes de Petri entre ferramentas que oferecem suporte a esse formalismo, suportanto mecanismos de modularização bastante expressivos e úteis para o uso com programas #.

5.5 ANALISANDO PROPRIEDADES DE PROGRAMAS

A presente seção apresenta exemplos que ilustram o uso do esquema de tradução descrito na Seção 5.2 para análise de propriedades de programas # por meio de redes de Petri. Os exemplos escolhidos (*Jantar dos Filósofos* e *Protocolo do Bit Alternado*) não constituem exemplos de programas paralelos reais, no sentido aplicado em processamento de alto desempenho. Entretanto, são apropriados para os propósitos desta seção, ilustrando a expressividade do modelo # e as potencialidades oferecidas pelo mecanismo de análise de propriedades de programas paralelos # por meio de redes de Petri.

5.5.1 Jantar dos Filósofos

O Jantar dos filósofos constitui um problema bem conhecido de sincronização de processos, originalmente proposto por Dijkstra[72] e enunciado da seguinte forma:

> "Cinco filósofos encontram-se ao redor de uma mesa, jantando. Cada um

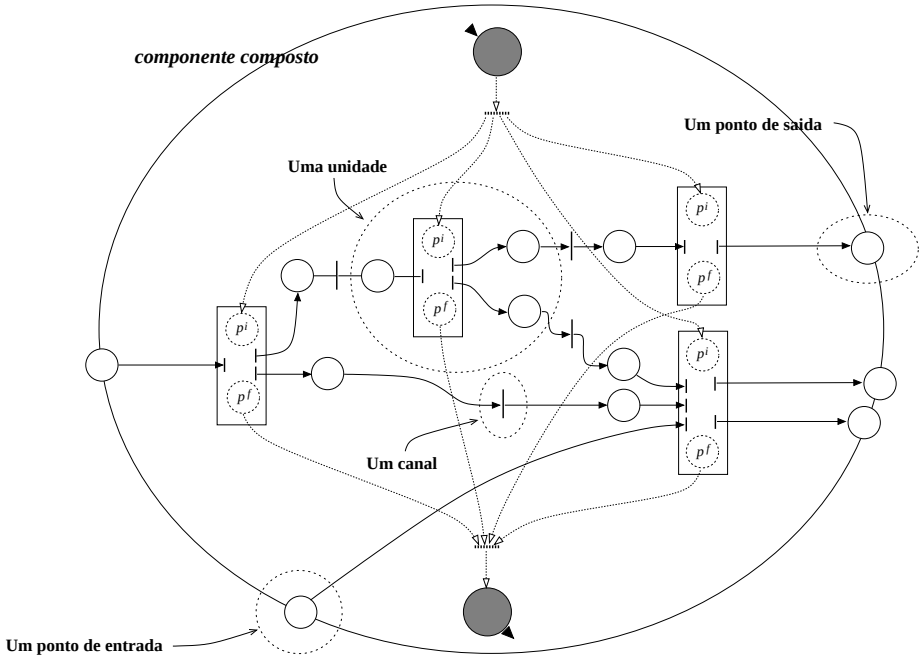


Figura 5.21. Visão Modular da Rede de Petri que Modela um Componente Composto

destes possui dois talheres, à direita e à esquerda. Existem cinco talheres disponíveis, de forma que cada par de filósofos adjacentes compartilha um talher. Para comer, um filósofo precisa estar de posse dos talheres à sua direita e à esquerda. Quando não está de posse dos dois talheres, o filósofo encontra-se pensando."

Embora aparentemente simples, o *jantar dos filósofos* modela um dos principais problemas de sincronização que podem ocorrem em sistemas concorrentes, o *deadlock*. Imaginemos que cada filósofo represente um *processo* e que cada talher represente um *recurso*, compartilhado entre os processos. Recursos são *seções críticas*, sendo assim acessados de maneira mutuamente exclusiva. Nesse contexto, um processo (filósofo) precisa estar de posse de ambos os recursos (talheres), o qual requisita, para realizar uma certa operação (comer) a que se destina. Suponha que todos os processos (filósofos), simultaneamente, peçam e obtenham permissão para usar o recurso (talher) à direita. Isso modela um estado de *deadlock*, uma vez que todos os processos estarão esperando por um recurso (talher à esquerda) que se encontra de posse de outro processo dentro do conjunto. O problema do jantar dos filósofos tem sido vastamente utilizado para comparar e avaliar mecanismos de sincronização e linguagens concorrentes.

5.5.1.1 Um Modelo # para o Jantar dos Filósofos Com a finalidade de prover um primeiro exemplo do emprego de redes de Petri para análise formal de programas #, analisaremos a implementação nesta linguagem de duas soluções para o problema do jantar dos filósofos, com características distintas.

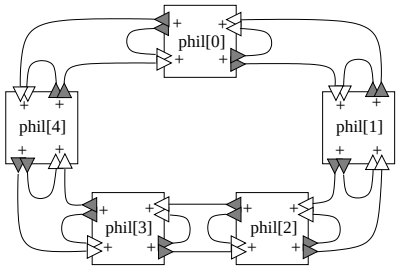


Figura 5.22. Topologia # para o Jantar dos Filósofos (Primeira Solução)

A primeira solução modela o mecanismo mais geral de sincronização entre os filósofos, onde estes literalmente disputam a posse dos talheres segundo um conjunto mínimo de restrições suficientes para manter a coerência com o enunciado original do problema. Na solução enunciada, os filósofos devolvem e obtêm os talheres (à sua direita ou à sua esquerda) sem uma ordem pré-estabelecida. Assume-se de início que cada filósofo detém um talher. Por convenção, a primeira operação realizada por cada filósofo é devolver o talher à sua direita na mesa. A partir de então os filósofos alternam entre tentar obter ambos os talheres, usá-los (comer) e devolvê-los à mesa. A ordem em que os talheres (direita e esquerda) são obtidos ou devolvidos é arbitrária. A única restrição é com relação ao fato de que um filósofo deve esperar que um talher esteja livre (não esteja de posse do filósofo adjacente) para obter a permissão para usá-lo. É importante ressaltar que quando um filósofo devolve um talher à direita ou à esquerda, este pode ser obtido posteriormente tanto pelo próprio filósofo quanto pelo filósofo adjacente, arbitrariamente. Analisando com cuidado a solução acima, esta apresenta dois inconvenientes:

- *possibilidade de deadlock*: Se todos os processos simultaneamente tentam obter o talher à direita (ou à esquerda), todos permanecerão bloqueados para sempre;
- *não garante justiça*: É possível que um seja sempre preterido de obter os dois talheres, caso não seja empregado uma política de escalonamento *fortemente justa*, a qual garanta que, a menos de terminação, se um processo requisita os talheres infinitas vezes na execução do programa este será atendido em um tempo finito. Entretanto, escalonadores fortemente justos não são considerados práticos na maioria dos casos[6].

Na discussão que se segue, considere N o número de filósofos. A topologia de unidades da implementação # para a solução descrita acima encontra-se ilustrada na Figura 5.22, para a instância $N = 5$. Nesta, cada filósofo é modelado por um processo ($phil[i]$, $0 \leq i \leq N-1$). A presença ou ausência de um talher na mesa é modelada pela existência ou não de um valor trafegando em um dos canais bufferizados que conectam as portas de dois processos (filósofos) adjacentes. Tanto à sua direita quanto à sua esquerda, cada processo (filósofo) possui um agrupamento constituído por duas portas de entrada e outro constituído por duas portas de saída, ambos do tipo *choice*. Considere o "lado direito" de um processo filósofo. Os agrupamentos de portas são identificadas como rf_get (entrada)

```
component DiningPhilosophers<N> with

index i range [0,N-1]

interface IPhil # (rf_get*, lf_get*) → (rf_put*, rf_put*)
        behavior: seq {rf_put!;
                       repeat seq {par {lf_get?; rf_get?};
                                   par {lf_put!; rf_put!}}
                       until < lf_get & rf_get & lf_put & rf_put >}

[/ unit phil[i] # IPhil groups rf_get< 2 >: choice, lf_get< 2 >: choice,
                               rf_put< 2 >: choice, lf_put< 2 >: choice /]

[/ connect phil[i]→rf_put[0] to phil[i-1 mod N]←lf_get[0], buffered
   connect phil[i]→rf_put[1] to phil[i]←rf_get[1], buffered
   connect phil[i]→lf_put[0] to phil[i+1 mod N]←rf_get[0], buffered
   connect phil[i]→lf_put[1] to phil[i]←lf_get[1], buffered /]
```

Figura 5.23. Código # para a Implementação da Primeira Solução para o Problema do Jantar dos Filósofos

e rf_put (saída). O agrupamento rf_put é ativado quando o filósofo decide devolver o talher à sua direita à mesa, enquanto o agrupamento rf_get é ativado quando o filósofo deseja obter a permissão para usar este talher. Explicação análoga é válida no que se refere ao "lado esquerdo" de um processo filósofo, constituído dos agrupamentos lf_get e lf_put. Em cada agrupamento, a porta individual de índice 0 é usada quando o filósofo troca (devolve ou recebe) o talher com seu vizinho, enquanto a porta de índice 1 é usada quando o filósofo decide reusar o talher na próxima iteração. O recurso de usar agrupamentos *choice* é necessário para modelar o fato de que um talher ao ser devolvido por um filósofo poderia ser obtido posteriormente pelo seu filósofo vizinho ou pelo próprio.

O código # que implementa a topologia apresentada na Figura 5.22 é apresentada na Figura 5.23. A interface *IPhil* descreve o comportamento das unidades $phil[i]$, $0 \leq i \leq N-1$. Inicialmente, cada filósofo detém um talher, uma vez que no estado inicial do programa não há dados trafegando nos canais que conectam as portas de processos filósofos adjacentes. A primeira operação consiste então em todos os processos (filósofos) ativarem suas respectivas portas rf_put (filósofo devolve talher à direita na mesa). Posteriormente, cada processo (filósofo) repetidamente ativa simultaneamente suas portas de entrada lf_get e rf_get (filósofo tenta obter os talheres à sua direita e à sua esquerda, em qualquer ordem) e, em sequência, ativa simultaneamente suas portas de saída rf_put e lf_put (filósofo devolve os talheres à mesa, após fazer uso destes). É fácil verificar que este programa # entrará em *deadlock* (impasse entre os filósofos) sempre que todos os processos filósofos ativarem, simultaneamente, a mesma porta de entrada de índice 0 (todos os filósofos tentam pegar o talher à direita ou à esquerda a partir de seu vizinho). Além disso, pode ocorrer a situação onde um processo filósofo é sempre preterido de obter os dois talheres, ao competir pela sua posse com outros processo filósofos, sendo necessário uma política de escalonamento *fortemente justa* para que isto não ocorra. Entretanto, a

semântica de ativação de agrupamentos *choice* não implementa uma política justa para habilitação das porta individuais pertencentes ao agrupamento, uma vez que não é garantido que uma porta individual que faz parte de sua constituição será ativada durante a execução mesmo que o agrupamento seja potencialmente ativado infinitas vezes durante uma execução do programa, a menos de terminação. Entretanto, esta possibilidade é pouco provável em programas comuns.

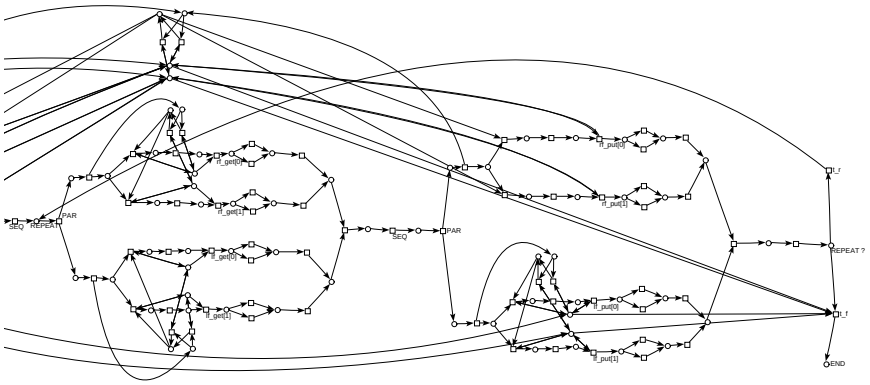


Figura 5.24. Rede de Petri que Modela o Comportamento de um Processo *phil[i]* no Programa # (Versão 1) que Implementa o Problema Jantar dos Filósofos (Ignorando Sincronização de de Finalização de *Streams*)

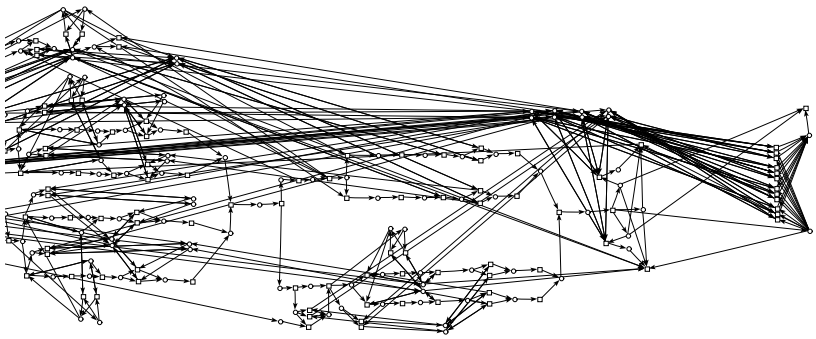


Figura 5.25. Rede de Petri que Modela o Comportamento de um Processo *phil[i]* no Programa # (Versão 1) que Implementa o Problema Jantar dos Filósofos (Incluindo Protocolo de Sincronização de *Streams*)

A partir de sua descrição *abstract #*, aplicando-se o esquema de tradução detalhado na Seção 5.2, traduzimos o programa # que implementa o jantar dos filósofos para redes de Petri, assumindo $N = 5$ (cinco filósofos). A rede de Petri que modela o comportamento de um processo filósofo encontra-se ilustrada na Figura 5.24. Nesta, o protocolo de sincronização da natureza do valor transmitido pelas portas (Seção 5.2.12) ainda não foi introduzido, de forma facilitar seu entendimento pelo leitor. A rede resultante da introdução deste protocolo sobre a rede da Figura 5.24 é apresentada na Figura 5.25.

Percebe-se que a introdução de um grande número de componentes na rede, devido a introdução do protocolo, torna sua compreensão difícil, embora o comportamento do processo seja modelado de maneira mais fiel à realidade e permitindo uma análise mais apurada de suas propriedades. A Figura 5.25 é portanto meramente ilustrativa. A Figura 5.26 ilustra a rede completa, também ilustrativa pelo motivo exposto, incluídos os cinco processos que modelam os filósofos. Os arcos que conectam as redes induzidas por cada processo filósofo, facilmente identificáveis na figura, modelam os canais de comunicação que ligam suas portas e o protocolo de sincronização da natureza do valor transmitido entre portas conectadas por um canal.

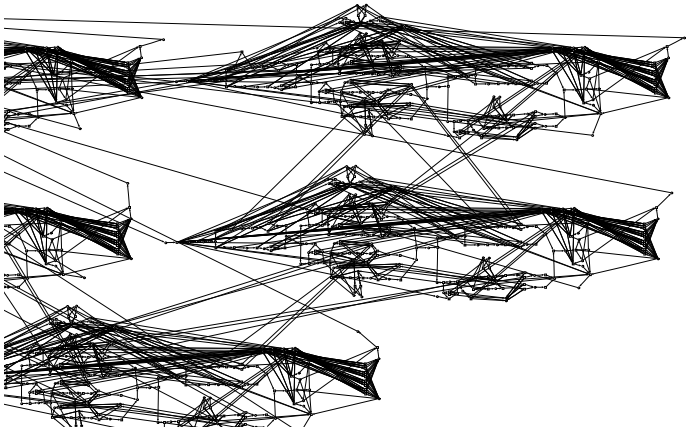


Figura 5.26. Rede de Petri para p Programa # de um Processo (Versão 1) que Implementa o Problema do Jantar dos Filósofos

Utilizando a ferramenta INA (*Integrated Net Analyzer*)[194], é possível verificar propriedades formais sobre o programa # em questão, permitindo assim sua análise. A Tabela 5.1 enumera propriedades estruturais e comportamentais inferidas por INA com respeito à rede de Petri que modela o comportamento do programa em questão.

As informações contidas nestas propriedades são usadas pelo sistema especialista implementado por INA para selecionarem-se os algoritmos apropriados que devem ser aplicados sobre a rede em questão, para efetivação dos diversos tipos de análise suportados pelo ambiente, tais como análise de *invariantes* (lugar e transição), testes de *alcançabililidade* de marcações, testes de simetria, reduções, etc. Por exemplo, o fato de a rede ser limitada (propriedades 20 e 21) permite que seu comportamento dinâmico seja modelável por máquinas de estados finitos. Em termos de programas #, significa que as sequências possíveis de ativação das portas do processos descrevem linguagens regulares, ou padrões regulares de interação. Para redes desse tipo, é possível construir o seu *grafo de estados*, uma ferramenta poderosa para análise das propriedades dinâmicas de uma rede de Petri, em especial no que diz respeito à alcançabilidade de marcações e propriedades relacionadas à *liveness*. Entretanto, em redes não limitadas, é ainda possível computar-se o chamado *grafo de cobertura*, embora esta tarefa seja mais custosa do ponto de vista computacional, ocupando mais recursos da máquina, não seja computável (em sua total-

1	**não**	Livre de conflitos dinâmicos
2	**não**	Pura
3	**sim**	Ordinária
4	**sim**	Homogênea
5	**não**	Conservativa
6	**não**	Subconservativa
7	**não**	Máquina de Estados
8	**não**	Livre Escolha
9	**não**	Livre Escolha Extendida
10	**não**	Simples Extendida
11	**não**	Decomponível por máquina de estado
12	**não**	Alocatável por máquina de estado
13	**não**	*Live*
14	**não**	*Live* e Segura
15	**sim**	Marcada
16	**sim**	Marcada com exatamente um token
17	**sim**	Conectada
18	**não**	Fortemente Conectada
19	**sim**	Limitada
20	**sim**	Estruturalmente limitada
21	**sim**	Coberta por sub-invariantes semi-positivas de transição
22	**sim**	Coberta por sub-invariantes semi-positivas de lugar
23	**não**	Reversível
24	**não**	Propriedade de *deadlock-trap*
25	**não**	Grafo marcado

Tabela 5.1. Propriedades

idade) para qualquer tipo de rede e permita um poder menor de análise que o grafo de estados.

Com a finalidade de analisar-se a existênca de *deadlocks* dinâmicos nesta implementação # do jantar dos filósofos, tornamos inalcançáveis as marcações mortas relacionadas a terminação normal do programa, por finalização de todas as *streams* transmitidas pelos filósofos em uma iteração ou falha de sincronização da terminação destas [2]. Respectivamente, nas marcações que caracterizam estes estados encontra-se depositada pelo menos uma marca nos lugares 619 ou 124 da rede INA, respectivamente correspondentes aos lugares p^f e p^{fail} na figura 5.1. Com essa finalidade, tornamos *mortas a partir da marcação inicial* todas aquelas transições cujo disparo determina a transmissão de um valor marcador de final de lista na ativação de uma porta *stream* de saída (38, 118, 163, 243, 266, 301, 372, 476, 515 e 624), utilizando um lugar (618) inicializado com zero marcas e sem transição de entrada (não pode receber marcas, portanto). Estas correspondem à transições t^s_k, $0 \leq k \leq n-1$, na rede da Figura 5.15. Observa-se que, em cada sub-rede induzida pela ativação de uma porta *stream* de saída, somente a transição t^s_n permanece habilitável, uma vez que o disparo desta caracteriza a transmissão de um valor de dados. Utilizando esse recurso e assumindo-se a não existência de *deadlocks* por impasse entre os filósofos ao tentarem obter os talheres, o programa deveria executar para sempre, uma vez que a satisfação das condições de terminação ou falha ao final de uma iteração (comando **repeat**) é modelada pela alcançabilidade de marcações que resultam do disparo, na última iteração, de uma das transições forçadamente mortas nesta análise.

A existência de marcações mortas na rede que modela o programa, a partir das quais nenhuma outra marcação pode ser alcançada, indica a existência de *deadlocks*. Para computá-las, é necessário construir o *grafo de estados* ou o *grafo de cobertura* da rede em

[2]De acordo com a condição de terminação do **repeat** do programa em questão, ao final de uma iteração, todas as *streams* transmitidas pelas portas de cada filósofo devem finalizar ou não, caracterizando respectivamente as condição de terminação ou execução de uma nova iteração na repetição. Falha ocorre quando, ao final da iteração, co-existem *streams* finalizadas e não finalizadas

questão. Uma vez que a rede induzida por esta implementação do jantar dos filósofos é uma rede *limitada*, é possível computar-se um grafo de estados que represente seu comportamento dinâmico. Entretanto, devido ao grande número de componentes da rede, não é possível computar o grafo completo (foi empregado para esta tarefa um computador Pentium IV, 2 GHz, 256MB RAM). Porém, uma vez que o objetivo é encontrar pelo menos um estado morto, o que caracterizaria a existência de *deadlock* na rede, somente parte do grafo de estados é suficiente. A ferramenta INA permite configurar a profundidade máxima do grafo de estados. Utilizando este recurso, limitamos o grafo à profundidade 180. Após serem computados 351.381 estados, foram encontrados 7 estados (marcações) mortos(as). A seguir, apresentamos a saída gerada pelo INA, gravada no arquivo `SESSION.INA`:

```
Current analysis options are:
     symmetrical reduction
     stubborn reduction
     maximal depth = 180
     do not use a 'bad' predicate

Computation of the reachability graph using symmetrical and
stubborn reduction. States generated: 351381 Arcs generated:
410801

Capacities needed so far: Place   1   2   3   4   5   6   7   8
9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25
26  27  28  29  30  31  32  33  34
    :  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
    :  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
    : 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
    : 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
    : 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
    : 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238
    : 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272
    : 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306
    : 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340
    : 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374
    : 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408
    : 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442
    : 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476
    : 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510
    : 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544
    : 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578
    : 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612
    : 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646
    : 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670
 Cap:   1   1   1   1   1   1   1   1   1   1   0   1   0   1   1   1   1   1   1   0   1   1   1   1   1   1   0   1   1   1   0   1   1
    :   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1
    :   0   1   1   1   1   1   1   0   0   1   0   0   0   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   0   1   0   1   1
    :   1   1   1   0   1   1   0   1   1   1   1   1   0   1   1   1   0   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   0   0   1   0   0   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   0   1   1
    :   1   1   1   0   1   1   0   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   0   0   1   0   0   0   1   1   1   1   1   1
    :   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   0   1   1   1   1   1   1   1   0   1
    :   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   0   1   1   1   1   1   1   1   1   1   1   1   0   1
    :   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   0   1   1   1   1   1   1   1   1   0   1   1   1   1   1   0   1   1   1
    :   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   0   1   1   1   0
    :   1   1   1   0   1   1   1   1   1   0   1   0   0   0   1   1   1   1   1   1   1   1   1   1   0   1   0   1   1   1   1   1   1   1
    :   1   0   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   0
    :   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
Dead states: 132, 866,1005,2092,2981,3106,97005, Number of dead
states found: 7 The net has dead reachable states. The net is not
reversible (resetable). The computed graph is not complete.
```

Os lugares cuja capacidade usada na computação do grafo é zero correspondem a um sub-conjunto daqueles que possuem número zero de marcas em qualquer marcação alcançável da rede. A presença de marcas nestes lugares ocorre como efeito colateral da transmissão de valores marcadores de final de lista em algum momento anterior da

execução. As marcações mortas encontradas são enumeradas a seguir e seu significado é discutido nos parágrafos que se seguem. Os números em cada item indicam os identificadores INA para os lugares que contém uma marca na marcação em questão (a rede é *marcada com exatamente uma marca*, como demonstrado pela propriedade 17 na Tabela 5.1):

i) **Estado 132**: 17, 21, 22, 27, 29, 53, 58, 59, 82, 83, 91, 100, 116, 117, 141, 142, 148, 150, 174, 179, 180, 203, 204, 212, 221, 222, 237, 238, 253, 256, 257, 277, 278, 284, 286, 289, 309, 314, 315, 341, 342, 350, 359, 383, 384, 398, 403, 404, 437, 441, 442, 453, 455, 462, 463, 470, 498, 504, 505, 525, 526, 545, 547, 553, 554, 574, 583, 584, 598, 599, 619, 636, 637, 639, 640, 642, 643, 645, 646, 648, 649;

ii) **Estado 866**: 17, 21, 22, 27, 29, 53, 58, 59, 82, 83, 91, 100, 116, 117, 141, 142, 148, 150, 174, 179, 180, 203, 204, 212, 221, 222, 237, 238, 253, 256, 257, 277, 278, 284, 286, 289, 314, 315, 341, 342, 350, 359, 383, 384, 398, 403, 404, 437, 439, 441, 453, 455, 462, 463, 470, 498, 504, 505, 525, 526, 545, 547, 553, 554, 574, 583, 584, 598, 599, 619, 636, 637, 639, 640, 642, 643, 645, 646, 648, 649;

iii) **Estado 1005**: 17, 19, 21, 27, 29, 58, 59, 82, 83, 91, 100, 116, 117, 139, 141, 142, 148, 150, 174, 179, 180, 203, 204, 212, 221, 222, 237, 238, 253, 256, 257, 277, 278, 284, 286, 289, 314, 315, 341, 342, 350, 359, 383, 384, 398, 403, 404, 437, 439, 441, 453, 455, 462, 463, 470, 498, 504, 505, 525, 526, 545, 547, 553, 554, 574, 583, 584, 598, 599, 619, 636, 637, 639, 640, 642, 643, 645, 646, 648, 649;

iv) **Estado 2092**: 17, 19, 21, 27, 29, 58, 59, 82, 83, 91, 100, 116, 117, 139, 141, 148, 150, 179, 180, 203, 204, 212, 221, 222, 237, 238, 253, 256, 257, 277, 278, 284, 286, 289, 314, 315, 341, 342, 350, 359, 383, 384, 398, 403, 404, 437, 439, 441, 453, 455, 462, 463, 470, 498, 504, 505, 525, 526, 545, 547, 553, 554, 574, 583, 584, 598, 599, 619, 636, 637, 639, 640, 642, 643, 645, 646, 648, 649;

v) **Estado 2981**: 17, 19, 21, 27, 29, 58, 59, 82, 83, 91, 100, 116, 117, 139, 141, 148, 150, 179, 180, 203, 204, 212, 221, 222, 237, 238, 253, 256, 257, 275, 277, 284, 286, 289, 314, 315, 341, 342, 350, 359, 383, 384, 403, 404, 437, 439, 441, 453, 455, 462, 463, 470, 498, 504, 505, 525, 526, 545, 547, 553, 554, 574, 583, 584, 598, 599, 619, 636, 637, 639, 640, 642, 643, 645, 646, 648, 649;

vi) **Estado 3106**: 17, 19, 21, 27, 29, 58, 59, 82, 83, 91, 100, 116, 117, 139, 141, 148, 150, 179, 180, 203, 204, 212, 221, 222, 237, 238, 253, 256, 257, 275, 277, 284, 286, 289, 314, 315, 341, 342, 350, 359, 383, 384, 403, 404, 437, 439, 441, 453, 455, 462, 463, 470, 502, 504, 525, 526, 545, 547, 553, 554, 574, 583, 598, 599, 619, 636, 637, 639, 640, 642, 643, 645, 646, 648, 649;

vii) **Estado 97005**: 17, 19, 21, 29, 30, 47, 58, 75, 82, 83, 91, 116, 117, 139, 141, 150, 151, 168, 179, 196, 203, 204, 212, 222, 237, 238, 244, 253, 256, 257, 275, 277, 286, 287, 289, 314, 334, 341, 342, 350, 383, 384, 392, 403, 436, 437 439, 441, 455, 456, 462, 463, 482, 502, 504, 525, 526, 547, 548, 553, 564, 574, 584, 598, 599, 619, 631, 632, 633, 634, 635, 638, 641, 644, 647, 650.

Os estados de (i) à (vi) correspondem à marcações onde todos os filósofos conseguem permissão para usar o talher à sua esquerda (ocorrência de marcas nos lugares 637, 640, 643, 646 e 649), porém encontram-se ainda tentando obter a permissão para usar o talher à sua direita (ocorrência de marcas nos lugares 636, 639, 642, 645 e 648). Neste estado, como previsto nas discussões dos parágrafos anteriores, os processos filósofos encontram-se em *deadlock*. A marcação (vii) corresponde ao estado dual ao primeiro, onde todos

i	LOOP_FLAGS$_i$	$\overline{\text{LOOP\_FLAGS}_i}$	first
0	83 e 82	21 e 22	53
1	204 e 203	141 e 142	174
2	463 e 462	277 e 278	398
3	599 e 598	504 e 505	498
4	342 e 341	441 e 442	309

Tabela 5.2. Discriminação dos Lugares LOOP_FLAGS$_i$, $\overline{\text{LOOP\_FLAGS}_i}$ e $first$ (Figura 5.15) correspondentes á ativação da porta lf_put em cada processo

os filósofos obtém a permissão para usar o talher à sua direita (ocorrência de marcas nos lugares 638, 641, 644, 647 e 650) mas encontram-se ainda esperando a liberação do talher à sua esquerda (ocorrência de marcas nos lugares 631, 632, 633, 634 e 635) para prosseguir sua execução. Obviamente, pelo fato de não ter sido possível computarem-se todos os estados do grafo, os estados mortos computados não constituem os únicos estados mortos da rede. Entretanto, são suficientes para mostrar a existência de *deadlocks* nesta implementação e provêem evidências de que os únicos *deadlocks* existentes referem-se às situações anteriormente previstas.

A diferença entre as marcações de (i) a (vi), as quais caracterizam uma mesma situação de *deadlock*, deve-se ao fato de que, em cada processo filósofo, este pode ocorrer antes ou depois da primeira ativação da porta lf_put, quando o processo encontra-se ainda em sua primeira iteração ou em uma ativação qualquer posterior à primeira, respectivamente, gerando várias possibilidades de marcações. Seja o processo filósofo i, $0 \leq i \leq N-1$. Caso o *deadlock* ocorra na primeira situação, os lugares em LOOP_FLAGS$_i$ e $\overline{\text{LOOP\_FLAGS}_i}$ correspondentes à porta lf_put encontram-se em seu estado inicial, quando a soma dos valores em LOOP_FLAGS$_i$ é zero, uma vez que não há ativação mais recente cuja natureza do valor transmitido necessite ser lembrada. Existe ainda uma marca no lugar pertencente a rede induzida pela ativação da porta lf_put correspondente ao lugar $first$ da rede 5.15. Após a primeira ativação a marca deste lugar desaparece e a soma das marcas nos lugares LOOP_FLAGS$_j$ torna-se 1 em qualquer marcação alcançável a partir deste estado. A Tabela 5.2 discrimina os números INA dos lugares citados, para cada processo.

Calculamos ainda o grafo de estados referente à rede de Petri original do programa em questão, sem o bloqueio forçado das transições que caracterizam a transmissão de valores marcadores de final de *stream* (desconsidera-se o lugar 618 e as transições que partem deste), visando determinar a alcançabilidade dos estados de terminação normal, descritos anteriormente, caracterizados pela presença de marca nos lugares 619 ou 124. Utilizando a mesma profundidade ($D = 180$), foram encontrados 87 estados mortos após a computação de 517.433 estados. Dentre estes, além dos estados mortos anteriormente citados, acrescentaram-se estados onde havia uma marca nos lugares 619 ou 124, como previsto. No segundo caso, alguns processos falhavam ao testar a condição de terminação da repetição, enquanto os demais bloqueavam ao tentar obter os talheres de processos que haviam terminado por falha.

```
component DiningPhilosophers<N> with

index i range [0,N-1]

interface IPhil[0] (rf_get, lf_get) → (rf_put, rf_put)
        behavior: repeat seq {lf_put!;rf_get?; lf_get?; rf_put!}

interface IPhil[1] (rf_get, lf_get) → (rf_put, rf_put)
        behavior: {repeat seq {rf_get?; lf_put!; rf_put!; lf_get?}

[/ unit phil[i] # IPhil[i mod 2] /]

[/ connect phil[i]→rf_put to phil[i-1 mod N]←lf_get, buffered
   connect phil[i]→lf_put to phil[i+1 mod N]←rf_get, buffered /]
```

Figura 5.27. Código # para a Implementação da Segunda Solução para o Problema do Jantar dos Filósofos

5.5.1.2 Aprimorando a Solução No código na Figura 5.27 é apresentada uma nova solução para o jantar dos filósofos, alternativa à primeira, porém com as seguintes características favoráveis:

- **Ausência de *deadlocks***: Os processos sincronizam de forma a evitar situações de impasse;
- **Justiça Incondicional**: A todos os filósofos é dada a chance de obter os dois talheres em um número mínimo de iterações;
- **Máximo paralelismo**: garante-se que o maior número possível de filósofos estarão de posse de dois talheres simultamente (N **div** 2);
- **Síncrona**: Não são necessários canais *bufferizados*. O processo entrega o talher diretamente ao seu vizinho quando não mais precisa deste. O uso de cada talher é alternado entre os filósofos adjacentes;

A topologia dos processos na implementação desta segunda solução encontra-se ilustrada na Figura 5.28. A interface que descreve o comportamento dos processos pares difere da interface que descreve o comportamento dos demais processos (ímpares). A solução implementa um protocolo de governa a atuação dos filósofos, evitando situações de impasse, garantindo igual prioridade no uso dos talheres e permitindo que o número máximo de filósofos estejam usando os talheres (otimização no uso dos recursos). A decisão a cerca da finalização do programa é de responsabilidade do processos pares, ao início de cada interação, sempre que todos estes, simultaneamente, enviam o valor marcador de final de lista através da porta lf_put. A partir de então, até o final desta última iteração, todas as ativações devem envolver a transmissão de marcadores de final de lista. Caso contrário ocorre uma falha de sincronização.

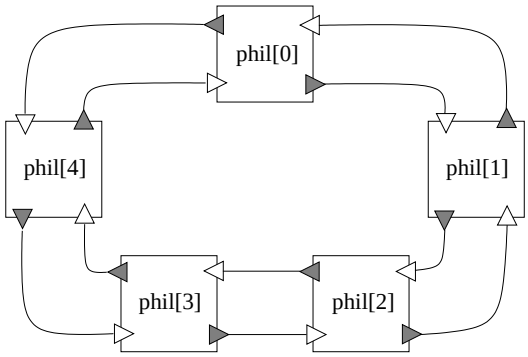


Figura 5.28. Topologia # para o Jantar dos Filósofos (Segunda Solução)

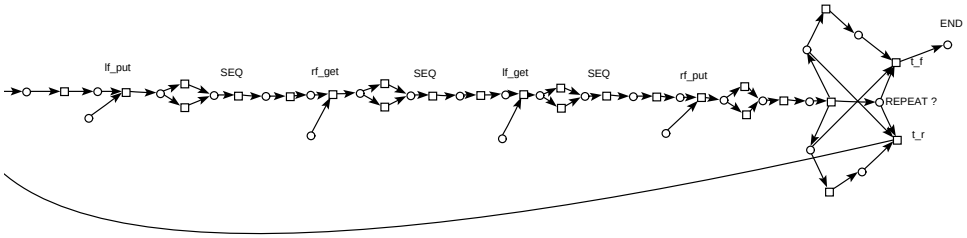


Figura 5.29. Rede de Petri que Modela o Comportamento de um Processo *phil[0]* no Programa # (Versão 2) que Implementa o Problema Jantar dos Filósofos (Ignorando Sincronização de de Finalização de *Streams*)

A rede de Petri induzida pelo comportamento do processo *phil*[0] é apresentada na Figura 5.29, sem considerar o protocolo de finalização de *streams*. Na Figura 5.30 é apresentada a mesma rede, porém incluído o protocolo de sincronização de *streams*. As redes induzidas pelos demais processos são análogas, com a pequena diferença de que para os processos ímpares a ordem de ativação das portas é alterada. Na Figura 5.26, ilustra-se o aspecto da rede de processos quando inseridos os cinco processos filósofos.

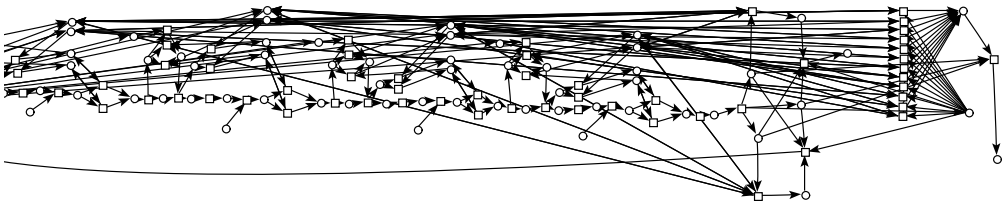


Figura 5.30. Rede de Petri que Modela o Comportamento de um Processo *phil[0]* no Programa # (Versão 2) que Implementa o Problema Jantar dos Filósofos (Incluindo Protocolo de Sincronização de *Streams*)

A propriedades elemetares inferidas por INA para a rede induzida por esta segunda versão # para o jantar dos filósofos coincidem com aquelas inferidas para a sua primeira versão.

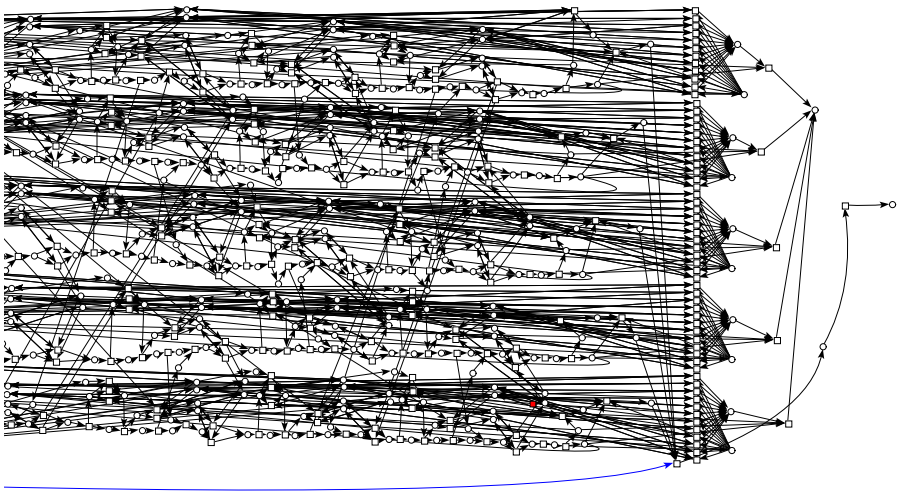


Figura 5.31. Rede de Petri para o Programa # de um Processo (Versão 2) que Implementa o Problema do Jantar dos Filósofos

Para demonstrar a inexistência de *deadlocks* causados pela ordem em que os talheres são requisitados pelos filósofos, como ocorre no caso anterior, utilizamos o mesmo esquema usado para demonstrar a existência de *deadlocks* na implementação anterior, tornando mortas desde a marcação inicial as transições que caracterizam a transmissão de valores marcadores de final de *stream*. Espera-se, no entanto, que não sejam encontrados estados mortos no grafo de marcações da rede induzida por este programa, ao contrário do caso anterior. Por ser a rede de Petri em questão menos complexa que a anterior, foi possível calcular o seu grafo de estados completo. Não foram encontrados estados mortos, como esperado. Observe-se a saída fornecida pelo INA, a qual atesta este fato:

```
Computation of the reachability graph using stubborn reduction.
States generated: 4458 Arcs generated: 5019

Capacities needed: Place   1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27
28  29  30  31  32  33  34
    :  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
    :  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
    : 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
    : 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
    : 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
    : 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238
    : 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272
    : 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306
    : 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340
    : 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361
 Cap:   1   0   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   0   1   0   1   1   1   1   0   1   1   1   1   1   1   1   0
    :   1   0   1   1   1   1   1   1   0   0   1   0   1   0   1   1   1   0   1   0   1   1   1   1   1   1   1   1   1   1   1   0   0   1
    :   1   1   1   1   1   1   1   1   1   0   1   1   1   0   1   0   1   1   1   0   0   1   0   1   1   1   1   1   1   1   1   0   1   1
    :   1   1   1   0   1   0   1   1   1   0   0   1   1   1   1   1   1   1   1   1   0   1   1   1   0   1   0   1   1   0   1   0   1
    :   1   0   1   0   1   0   1   0   1   1   1   1   1   0   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   0   1   0   1   0   1   0   1   0
The net has no dead reachable states.
```

Assim como na implementação anterior, os lugares de capacidade zero constituem um sub-conjunto do conjunto daqueles que caracterizam estados que só são alcançados como

consequência da transmissão de um valor marcador de lista na execução do programa. INA foi ainda capaz de analisar propriedades de *liveness*, como pode ser observado a seguir:

```
Liveness test: The set of live transitions is
       5,   6,   7,   8,  10,  11,  12,  13,  15,  16,  17,  18,  19,  20,  22,  23,  24,  25,  26,  27,  28,  31,  32,  33,
      34,  35,  37,  38,  40,  41,  42,  44,  45,  47,  48,  49,  50,  51,  53,  54,  55,  58,  59,  61,  62,  63,  64,  65,
      67,  68,  69,  70,  71,  73,  74,  75,  76,  77,  79,  80,  81,  82,  83,  85,  86,  87,  88,  89,  91,  92,  93,  94,
      95,  97,  98,  99, 100, 101, 103, 104, 105, 106, 107, 109, 110, 111, 112, 113, 115, 116, 117, 118, 119, 121, 122, 123,
     124, 125, 127, 128, 129, 131, 132, 133, 135, 136, 137, 139, 140, 141, 142, 144, 145, 147, 148, 149, 151, 152, 155, 156,
     157, 159, 160, 162, 164, 166, 168, 170, 172, 174, 176, 177, 180, 182, 184, 186, 188, 190, 191, 193, 194, 197, 199, 201,
     204, 206, 207, 209, 212, 214, 215, 217, 220, 222, 223, 229, 230, 231, 232, 233,

The net is not live, if dead transitions are ignored. The net has
dead transitions at the initial marking. The set of dead
transitions is
       2,   3,   4,   9,  14,  21,  29,  30,  36,  39,  43,  46,  52,  56,  57,  60,  66,  72,  78,  84,  90,  96, 102, 108,
     114, 120, 126, 130, 134, 138, 143, 146, 150, 153, 154, 158, 161, 163, 165, 167, 169, 171, 173, 175, 178, 179, 181, 183,
     185, 187, 189, 192, 195, 196, 198, 200, 202, 203, 205, 208, 210, 211, 213, 216, 218, 219, 221, 224, 225, 226, 227, 228,
     234, 235, 236, 237, 238, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282,
     283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306,
     307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323,
```

As transições mortas na marcação inicial correspondem exatamente às transições forçadamente bloqueadas, bem como aquelas que dependem do disparo das primeiras para que sejam disparadas. O conjunto de transições *live* correspondem às transições da rede disparadas durante a atuação dos processos filósofos, incluindo aquelas que caracterizam a obtenção de permissão para obter e liberar um talher. A propriedade *live* destas transiçõs permite afirmar que estas são disparadas infinitas vezes na execução do programa, demonstranso que a solução é justa (todos os filósofos adquirem os talheres em algum momento).

Calculamos ainda o grafo de estados completo da rede original, sem o bloqueio das transições cujo disparo caracteriza a transmissão de um valor marcador de final de *stream*, com a finalidade de analisar a alcançabilidade dos estados de terminação normal do programa. O grafo de estados resultante possui 373.410 estados e 451.818 transições. A saída produzida por INA é resumidamente listada a seguir:

```
Current analysis options are:
     no symmetrical reduction
     stubborn reduction
     no depth restriction
     do not use a 'bad' predicate
Computation of the reachability graph using stubborn reduction.
States generated: 373410 Arcs generated: 451818

Capacities needed: Place   1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27
28  29  30  31  32  33  34
     :  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
     :  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
     : 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
     : 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
     : 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
     : 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238
     : 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272
     : 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306
     : 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340
     : 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360
 Cap:    1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   5   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
     :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
Dead states:
```

```
.
.
Omitimos os números dos estados devido a grande quantidade destes.
.
.

Number of dead states found: 5108 The net has dead reachable
states. The net is not reversible (resetable). The following
transitions have not been fired:
      130.T1317, 134.T1329, 138.T1333, 143.T1338, 146.T1344, 150.T1466, 153.T1470, 154.T1472, 158.T1582, 161.T1585,
      163.T1587, 165.T1589, 167.T1591, 169.T1593, 171.T1595, 173.T1597, 175.T1599, 178.T1602, 179.T1603, 181.T1605,
```

Com execção do lugar 320, discutido adiante, a capacidade necessária para todos os lugares da rede é 1. Portanto todos os lugares possuem no máximo uma marca em alguma marcação alcançável da rede. O lugar 320 corresponde ao lugar p^{fail} na Figura 5.1. Por este motivo possui capacidade 5, uma vez que o número de marcas nesse lugar ao final da execução indica quantos processos falharam ao verificar a condição de terminação de uma repetição. Entretanto, modificações simples seriam necessárias de forma que a capacidade do lugar p^{fail} fosse limitada a uma única marca, a qual indicaria a ocorrência de uma falha. A rede de Petri resultante teria a propriedade de ser uma rede de Petri *segura*. Foram encontrados 5.108 estados mortos. Destes, 5.107 correspondem a estados que caracterizam falhas de sincronização da finalização de *streams*, quando ao final da última iteração de um ou mais processos filósofos é verificado que as *streams* não sincronizaram a natureza do valor transmitido, desobedecendo a restrição imposta pelos delimitadores '<' e '>' na condição de terminação do **repeat**. Este estado, como vimos, é modelado pela presença de marcas no lugar 320, uma para cada processo que falhou. Como esperado, um único estado morto, dentro os 5.108 inferidos, caracteriza a terminação normal do programa, quando, ao final de uma iteração, todos os filósofos transmitiram na última ativação de cada uma de suas portas um marcador de final de lista. Esse estado é modelado pela presença de uma marca no lugar 3. Foram ainda encontradas 20 transições que nunca são disparadas. Assumindo que todas as *streams* transmitidas no programa possuem fator de aninhamento 1 ($n = 1$), estas correspondem às transições t_0^r na Figura 5.15, responsáveis por limpar o lugar $flag_0$ em uma nova iteração, caso um marcador de final de lista em aninhamento 1 tenha sido lido na ativação mais recente. Entretanto, este programa sempre termina quando ao final de uma iteração pelo menos uma das portas *stream* de algum processo transmite um marcador de final de lista. Assim, a próxima iteração, onde deveria ocorrer o disparo de alguma das transições t_0^r, não ocorre. Isso demonstra uma importante propriedade satisfeita por esse programa: os processos não tentam ativar uma porta após a *stream* transmitida por esta ter sido finalizada, sem necessidade de introduzir-se qualquer protocolo que controle a ordem da natureza dos valores transmitidos em sequências de ativação de uma mesma porta, como aquele discutiro na Seção 5.17.

5.5.2 O Protocolo do Bit Alternado

O protocolo do *bit* alternado é uma forma simples, porém efetiva, de gerenciar a retransmissão de mensagens perdidas em implementações de baixo nível do modelo comunicação por passagem de mensagens. Considerando um processo emissor A e um processo receptor B, connectados por um canal *stream* ponto-a-ponto, o protocolo garante

```
program({unit (phil[0],{port (rf_get,stream),port (lf_get,stream),
                        port (rf_put,stream),port (lf_put,stream)},non-repetitive,
                behavior (∅, repeat (seq{lf_get!,rf_get!,lf_put!,rf_put!},
                                     < lf_get & rf_get & lf_put & rf_put >))
         ),
         unit (phil[1],{port (rf_get,stream),port (lf_get,stream),
                        port (rf_put,stream),port (lf_put,stream)},non-repetitive,
                behavior (∅, repeat (seq{lf_get!,rf_get!,lf_put!,rf_put!},
                                     < lf_get & rf_get & lf_put & rf_put >))
         ),
         unit (phil[2],{port (rf_get,stream),port (lf_get,stream),
                        port (rf_put,stream),port (lf_put,stream)},non-repetitive,
                behavior (∅, repeat (seq{lf_get!,rf_get!,lf_put!,rf_put!},
                                     < lf_get & rf_get & lf_put & rf_put >))
         ),
         unit (phil[3],{port (rf_get,stream),port (lf_get,stream),
                        port (rf_put,stream),port (lf_put,stream)},non-repetitive,
                behavior (∅, repeat (seq{lf_get!,rf_get!,lf_put!,rf_put!},
                                     < lf_get & rf_get & lf_put & rf_put >))
         ),
         unit (phil[4],{port (rf_get,stream),port (lf_get,stream),
                        port (rf_put,stream),port (lf_put,stream)},non-repetitive,
                behavior (∅, repeat (seq{lf_get!,rf_get!,lf_put!,rf_put!},
                                     < lf_get & rf_get & lf_put & rf_put >))
         )
        },
        {
         connect ((phil[0],rf_put),(phil[1],lf_get),synchronous),
         connect ((phil[0],lf_put),(phil[4],rf_get),synchronous),
         connect ((phil[1],rf_put),(phil[2],lf_get),synchronous),
         connect ((phil[1],lf_put),(phil[0],rf_get),synchronous),
         connect ((phil[2],rf_put),(phil[3],lf_get),synchronous),
         connect ((phil[2],lf_put),(phil[1],rf_get),synchronous),
         connect ((phil[3],rf_put),(phil[4],lf_get),synchronous),
         connect ((phil[3],lf_put),(phil[2],rf_get),synchronous),
         connect ((phil[4],rf_put),(phil[0],lf_get),synchronous),
         connect ((phil[4],lf_put),(phil[3],rf_get),synchronous),
        }
)
```

Figura 5.32. Código *abstract* # do Jantar dos Filósofos (Segunda Versão)

que sempre que uma mensagem transmitida de A para B é perdida, esta é retransmitida.

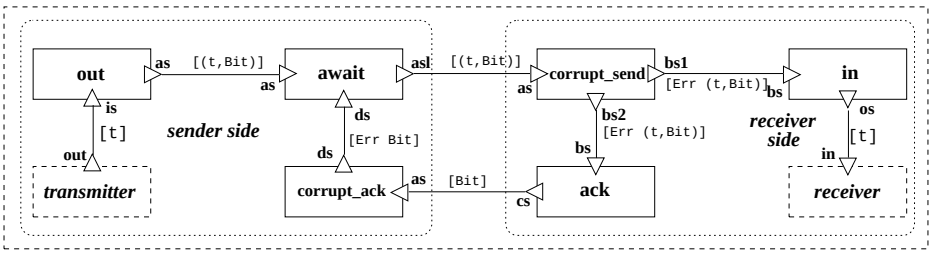


Figura 5.33. Topologia de Processos # para o Protocolo do Bit Alternado

A implementação # descrita a seguir é baseada na implementação funcional descrita em [78]. A Figura 5.33 ilustra a topologia de processos de um componente abstrato ABP especificado em Haskell$_{\#}$. As unidades virtuais *transmitter* and *receiver* modelam as unidades emissoras e receptoras da *stream* de valores. As demais unidades, não virtuais, implementam o protocolo. Em um sistema real, as unidades que pertencem ao mesmo lado da comunicação devem ser alocadas ao mesmo nó. O lado emissor é composto pelas seguintes unidades: *transmitter*, *out*, *await*, e *corrupt_ack*, enquanto o lado receptor é composto por: *receiver*, *in*, *ack*, and *corrupt_send*. O processo *await* é capaz de reenviar uma mensagem repetidamente até que esta seja corretamente recebida pelo processo *ack*.

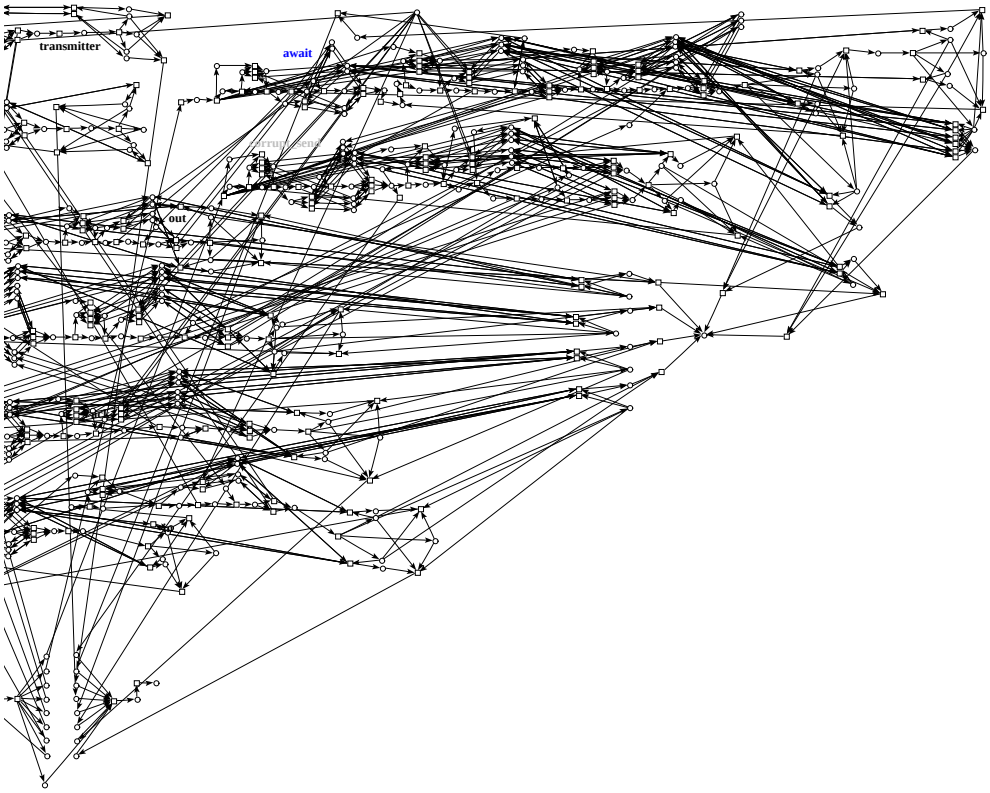


Figura 5.34. Rede de Petri que Modela o Programa # que Simula o Protocolo do Bit Alternado

Retransmissões são modeladas por meio de *streams* de fator de aninhamento 2, onde cada stream aninhada corresponde às retransmissões de um determinado valor. A verificação da chegada correta da mensagem é feita pelo valor recebido pela sua porta *ds*. Os processos *corrupt_ack* e *corrupt_send* verificam a ocorrência de erros nas mensagens que chegam do lado emissor e receptor, respectivamente, modelando a natureza não-confiável dos canais.

O código # apresentado na Figura 5.35 implementa o componente ABP. A rede de Petri que modela seu comportamento encontra-se ilustrada na Figura 5.34. Suas propriedades coincidem com aquelas apresentadas na Tabela 5.1. Foi verificada ainda a ausência de *deadlocks* no protocolo de retransmissão de mensagens. Para isso, utilizou-se a mesma técnica usada nos exemplos anteriores, tornando as transições que determinam a transmissão de valores marcadores de final de *stream* mortas a partir do estado ini-

```
component ABP with

use Out, Await, Corrupt, Ack, In

interface Transmitter # () → (out*::t)
          behavior: repeat out! until out

interface Receiver    # (in*::t) → ()
          behavior: repeat in? until in

interface Out         # (is::t) → (as::(t,Bit))
          behavior: repeat seq {is?; as!} until <is & as>

interface Await       # (as*::(t,Bit),ds**::Err Bit) → (as'**::(t,Bit))
          behavior: repeat seq { as?; repeat seq {as'!; ds?} until <as' & ds>} until <as & as' & ds>

interface Corrupt     # (as**::(t,Bit)) → (bs**::Err (t,Bit))
          behavior: repeat seq {as?; bs!} until <as & bs>

interface Ack         # (bs**::Err (t,Bit)) → (cs**::Bit)
          behavior: repeat seq {bs?; cs!} until <bs & cs>

interface In          # (bs**::Err (t,Bit)) → (os*::t)
          behavior: repeat seq { repeat bs? until bs; os!} until <bs & os>

unit transmitter # Transmitter () → out
unit receiver    # Receiver in → ()

unit out          # IOut     as Out
unit await        # IAwait   as Await
unit corrupt_ack  # ICorrupt as Corrupt
unit in           # IIn      as In
unit ack          # IAck     as Ack
unit corrupt_send # ICorrupt as Corrupt groups bs<2>:broadcast

connect * transmitter→out     to out←is
connect * in→os               to receiver←in
connect * out→as              to await←as
connect * await→as            to corrupt_send←as  buffered
connect * corrupt_ack→ds      to ack←ds
connect * corrupt_send→bs[1]  to in←bs
connect * corrupt_send→bs[2]  to ack←bs
connect * ack→cs              to corrupt_ack←cs   buffered
```

Figura 5.35. Componente *ABP*

cial. A execução da rede com o bloqueio dessas transições modela a situação onde um dado é sucessivamente retransmitido pelo protocolo, sem nunca alcançar sucesso. Se o protocolo estiver corretamente modelado, não devem ocorrer situações de impasse (*deadlock*). Para a rede de Petri em questão, assumindo-se as citadas restrições, foi possível computar-se seu grafo de estados completo, onde não foi encontrado nenhum estado morto, caracterizando-se assim a ausência de *deadlocks*. Eis a saída produzida por INA nesta análise:

```
Current analysis options are:
     no symmetrical reduction
     stubborn reduction
     no depth restriction
     do not use a 'bad' predicate
Computation of the reachability graph using stubborn reduction.
States generated: 9396 Arcs generated: 13170

Capacities needed: Place   1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27
28  29  30  31  32  33  34
    :  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
    :  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
    : 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
    : 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
    : 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
    : 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238
    : 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272
    : 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306
    : 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340
    : 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364
 Cap:   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   0   1   0   0   0   1   1   1   1   1   1   0   0   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   0   0   1   1   0   1   1   1
    :   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   0   0   1   1   0   1   1
    :   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   0   1   0   1   1   1   1   1   0   1   1   0   0   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   2   1   1   1   2   1   1   1   1   1   1   1   1   1
    :   1   1   1   0   0   1   1   1   1   1   0   1   0   1   0   0   1   0   1   1   0   1   0   1   0   1   0   1   0   1   0   1   0   1
    :   0   0   1   0   1   1   1   1   1   1   1   1   0   0   1   0   0   1   0   0   1   0   0   1   1   1   1   0   0   1   1   0   0   0
    :   0   1   1   1   0   0   0   0   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   0   0   1   1   1   1   0   0   1   1
    :   1   1   1   1   1   1   1   0   0   1   1   1   1   1   1   1   1   1   1   0   0   1   1   1   1   1   1   1   1   1   0   0   1
    :   1   0   1   1   1   0   0   0   0   1   0   1   1   1   0   1   1   1   0   1   1   1   1   1   1   0   0   0   0   0   1   0   1   1
    :   1   0   1   1   0   0   0   0   0   0   1   0   1   0   0   0   1   0   1   0   1   0   0   0

The net has no dead reachable states. The following transitions
have not been fired:
       6.T6,   8.T8,  11.T19,  14.T22,  18.T55,  19.T56,  22.T59,  24.T61,  25.T62,  26.T63,  27.T64,  29.T66,
      30.T67,  31.T68,  32.T69,  33.T115,  37.T119,  40.T122,  42.T124,  47.T129,  50.T132,  51.T133,  59.T141,
      63.T145,  65.T178,  69.T182,  74.T187,  79.T192,  82.T195,  83.T196,  91.T204,  95.T208,  97.T241,
      98.T242, 101.T245, 106.T250, 111.T255, 114.T258, 115.T259, 117.T261, 121.T265, 123.T267, 127.T271,
     143.T416, 144.T417, 149.T422, 150.T423, 153.T426, 154.T427, 156.T429, 160.T459, 162.T620, 163.T621,
     165.T624, 166.T625, 168.T627, 169.T741, 170.T742, 171.T745, 172.T746,
```

Os lugares de capacidade zero, jutamente com as transições não disparadas em nenhuma marcação constituem a sub-rede morta devido ao bloqueio das transições que caracterizam a finalização de *streams*. Observe-se que dois lugares (157 e 161) possuem capacidade 2. Respectivamente, estes correspondem aos canais *bufferizados* que ligam a porta $transmitter \rightarrow out$ à porta $out \leftarrow is$ dos processos e a porta $out \rightarrow as$ á porta $await \leftarrow as$. Isto indica que em alguma marcação é possível que o processo remetente tente enviar um novo valor antes que o processo receptor tenha consumido o valor enviado na última ativação.

Eliminando o bloqueio forçado às transições que caracterizam a transmissão de valores marcadores de final de lista, o grafo de estados cresceu de forma que sua computação completa tornou-se impossível no computador empregado para esta tarefa. Limitando-a à profundidade 170, construiu-se um grafo de estados parcial com 183.954 estados e 253.164 arcos. Dentre os estados, 1.136 estados mortos constituíam estados mortos. Eis a saída gerada por INA:

```
Current analysis options are:
     symmetrical reduction
     stubborn reduction
     maximal depth = 170
     do not use a 'bad' predicate

Computation of the reachability graph using symmetrical and
stubborn reduction. States generated: 183954 Arcs generated:
253164

Capacities needed so far: Place   1   2   3   4   5   6   7   8
9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25
26  27  28  29  30  31  32  33  34
    :  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
    :  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
    : 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
    : 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
    : 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
    : 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238
    : 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272
    : 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306
    : 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340
    : 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364
 Cap:   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0
    :   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   2   1   1   1   2   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
    :   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
Dead states:


.
.
.
Omitimos os números dos estados devido a grande quantidade destes.

.
.
.


Number of dead states found: 1136 The net has dead reachable
states. The net is not reversible (resetable). The computed graph
is not complete.
```

Dentre os estados mortos, um deles correspondia ao alcance da terminação normal do programa, com a finalização de todas as *streams*. Os demais correspondiam a terminação por falha da sincronização da finalização de *streams*. Neste caso, a capacidade de todos os lugares é maior que zero (0), com exceção dos lugares 34, 37 e 290, cujas marcações não foram ainda atingidas na computação parcial do grafo de estados. Os lugares 157 e 161 permanecem os únicos cuja capacidade necessária é dois (2), pelo motivo exposto anteriormente.

CAPÍTULO 6

IMPLEMENTAÇÃO E AVALIAÇÃO DE DESEMPENHO

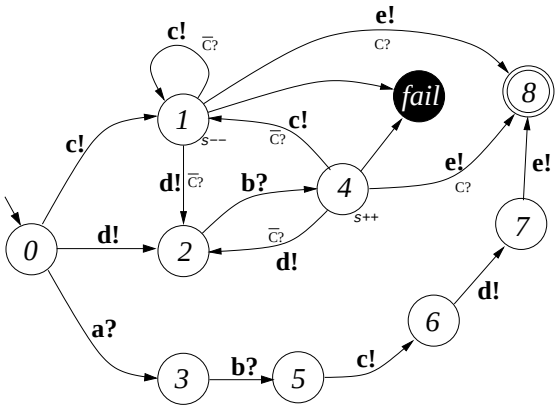
Neste capítulo, são apresentadas técnicas que demonstram como o modelo # pode ser implementado. Inicialmente, sob uma perspectiva geral, é discutida a implementação de processos # de acordo com o tipo de módulo funcional usado para descrição do meio de computação (Seção 3.3). Posteriormente, são apresentados detalhes inerentes à implementação da linguagem Haskell$_{\#}$.

6.1 A IMPLEMENTAÇÃO DE PROCESSOS

Na Seção 3.2.1, foi apresentado como processos # são caracterizados em nível de coordenação por meio do conceito primitivo de interface. A interface de um processo # constitui-se de portas de entrada e saída e uma expressão regular sincronizada por semáforos cuja linguagem gerada descreve a ordem de ativação destas portas. A ativação de uma porta completa-se com a efetivação de uma operação de comunicação no canal que conecta a porta ativada ao seu par de comunicação (Seção 3.2.3). O módulo funcional associado a uma unidade, o qual caracteriza a unidade em questão como um processo, define sua funcionalidade. As portas de entrada e saída da interface do processo são mapeadas aos argumentos e pontos de retorno do módulo funcional associado. O comportamento do módulo funcional com respeito a ordem em que os valores são consumidos e produzidos, respectivamente em seus argumentos e pontos de retorno, deve ser compatível com a ordem de ativação das portas de entrada e saída, assumindo-se o mapeamento especificado. Nos parágrafos que se seguem, propõe-se uma técnica para garantia de que o comportamento do processo definido pela sua interface seja respeitado na sua execução.

O modelo # permite ao programador validar de maneira mais precisa o programa paralelo desenvolvido, em comparação com outros modelos e linguagens voltadas ao mesmo fim. As propriedades estruturais e comportamentais da rede de comunicação do programa # podem ser efetivamente analisadas formalmente. Assumindo-se que o comportamento expresso pela interface do processo seja compatível com a implementação do módulo funcional associado, o programa se comportará como previsto na análise. Caso isso não ocorra, a especificação do comportamento da interface ou a implementação do módulo funcional deve estar incorreto. Uma vez que é possível analisar formalmente o comportamento do processo em nível de coordenação e detectar eventuais erros, usando ferramentas de redes de Petri como apresentados no Capítulos 5, pode-se isolar com facilidade a origem do problema, corrigindo-o. Tal recurso constitui um avanço frente aos mecanismos convencionais de programação por passagem de mensagens, onde programas, mesmo simples, podem ser difíceis de analisar e corrigir na presença de erros (*debugging*).

O compilador # é capaz de gerar código que verifica a ocorrência de incompatibilidade entre o comportamento especificado pela interface do processo e o comportamento real




```
interface IExample # (a,b) → (c,d,e)
        behavior: seq { alt { repeat alt {seq {c!; wait s}
                                          seq {d!; b?; signal s } } until c | < d & b >;
                              seq {a?; b?; c!; d!} };
                  e!}
```

$$
\begin{array}{rl}
C = c \mid (d \,\&\, b) & \text{(Condição de Terminação na transição } 8 \rightarrow 11\text{)} \\
\overline{C} = \neg c \,\&\, \neg d \,\&\, \neg b & \text{(Condição de Repetição nas transições } 8 \rightarrow 1 \text{ e } 8 \rightarrow 2\text{)} \\
\neg(C \vee \overline{C}) = (\neg d \wedge b) \mid (d \wedge \neg b) & \text{(Condição de Falha nas transições } 8 \rightarrow 1 \text{ e } 8 \rightarrow 2 \text{ e } 8 \rightarrow 11\text{)}
\end{array}
$$

Figura 6.1. Exemplo de Autômato de Validação

programado no módulo funcional. As mensagens de erro produzidas orientam o programador de maneira mais inteligível e confiável, facilitando a tarefa de validação do código computacional e assim permitindo a construção de programas mais robustos.

Na Seção 3.3, foram apresentados as classes de processos que podem ocorrer em programas #, segundo a linguagem usada na implementação do módulo funcional associado. Na seção seguinte, apresentamos o mecanismo empregado na verificação de incompatibilidade para processos *assíncronos*. Em seguida, discutiremos a extensão do mecanismo proposto para sua implementação sobre processos *orientados à demanda* e *orientados ao fluxo de dados*. Discutiremos ainda a implementação da linguagem Haskell$_{\#}$, a qual supõe a existência de processos orientados à demanda.

6.1.1 Implementando Processos Assíncronos

Na execução de processos assíncronos, a ordem em que as operações de envio e recebimento são realizadas é diretamente dependente do fluxo de controle do módulo funcional que o implementa. Em geral, estes são programados por meio de linguagens imperativas convencionais, como C ou Fortran. Para validação da sequência de efetivação das operações de comunicação em processos assíncronos, segundo o comportamento descrito por sua interface, é empregada a noção de *autômato de validação*, formalismo capaz de reconhecer a linguagem gerada pela expressão regular controlada por semáforos (ERCS) que descreve o comportamento do processo. A construção do autômato de validação a partir de uma ERCS é simples. Inicialmente é construído um autômato finito determinístico (AFD) mínimo equivalente à ERCS, considerando-se apenas as ativações e portas como

símbolos de entrada. Dessa forma, ignorando-se a semântica de semáforos, uma ERCS corresponde a uma expressão regular simples. As transições do AFD resultante estão portanto associadas à operações de comunicação sobre portas. A cada operação de comunicação sobre uma porta realizada pelo processo deve existir uma transição a partir do estado corrente rotulada com o identificador da porta em questão. O autômato de validação final é obtido ao adicionarem-se as informações relativas à atualização de semáforos e verificação das condições de repetição, terminação e falha das iterações. As operações de atualização de semáforos devem ser associadas a estados. Para um mesmo semáforo, em um certo estado, é possível haver mais de uma possibilidade de atualização. Durante a execução, em um dado instante, devem ser lembrados os valores (positivos) que um semáforo pode assumir. A invariante dos semáforos garante que deve existir pelo menos uma valoração positiva para cada semáforo, sendo portanto descartadas as valorações negativas de um semáforo sempre que estas ocorrerem. Caso não hajam valorações positivas possíveis, um erro em tempo de execução é informado. Entretanto, vale ressaltar que para a grande maioria dos programas paralelos, padrões regulares de comunicação são suficientes para descrever seu comportamento. Portanto, semáforos dificilmente são efetivamente utilizados. Em geral, o autômato de validação constitui um AFD convencional, permitindo uma implementação mais eficiente do algoritmo de validação dinâmica. Na Figura 6.1 é ilustrado o autômato de validação equivalente a expressão que especifica o comportamento de uma interface de processo.

Em seguida, o conceito de *autômato de validação* é formalizado e a implementação do algoritmo de validação da ordem das operações de comunicação usando este conceito é descrito.

Definição 6.1 (Autômato de Validação) *Um autômato de validação é definido por uma tupla C, descrita da seguinte forma:*

$$C = (\Pi, Q, T, \varphi_0, \varphi_1, \rho, q_0, F, \phi, S, \sigma, T^r, T^t, \gamma^r, \gamma^t, \gamma^f)$$

onde:

- Π é um conjunto de portas, as quais correspondem aos símbolos da linguagem aceita pelo autômato;
- Q é um conjunto finito de estados;
- T é um conjunto finito de transições;
- $\varphi_0 : T \rightarrow Q$ é a *função origem*, a qual mapeia uma transição ao seu *estado de origem*;
- $\varphi_1 : T \rightarrow Q$ é a *função destino*, a qual mapeia uma transição ao seu *estado de destino*;
- $\rho : T \rightarrow \Pi$ é a função de rotulação das transições, indicando qual a porta cuja ativação causa a mudança de estados;

- $q_0 \in Q$ é *estado inicial* do autômato;
- $F \subset Q$ é o conjunto de *estados finais* do autômato;
- $(\Pi, Q, T, \varphi_0, \varphi_1, \rho)$ define um autômato finito determinístico, cuja linguagem aceita descreve sequências válidas de ativação de portas, a menos da semântica associada à semáforos, caracterizada nos items adiante;
- $\phi : T \rightarrow \{\mathbf{I}, \mathbf{O}\}$ é a função que mapeia cada transição do autômato de validação à direção da porta correspondente ($\mathbf{I}$ = entrada, $\mathbf{O}$ = saída);
- S é um conjunto de símbolos que representam *semáforos*;
- $\sigma : Q \rightarrow 2^{S \times Nat}$ é a função de atualização de semáforos, aplicada sobre estados. Seja q, $q \in Q$. Para um certo semáforo s, $s \in S$, as duplas (s, n), $(s, n) \in \sigma(t)$, indicam as possíveis atualizações sobre o semáforo s como efeito da entrada no estado q;
- $T^r \in T$ contém as transições de repetição de iteração, induzidas pela ocorrências do combinador **repeat**;
- $T^t \in T$ contém as transições de terminação de iterações, induzidas pela ocorrências do combinador **repeat**;
- $\gamma^r : T^r \rightarrow \{Falso, Verdade\}$ é uma função que aplicada a uma transição verifica se a *condição de repetição* da iteração é *verdadeira* ou *falsa*;
- $\gamma^t : T^t \rightarrow \{Falso, Verdade\}$ é uma função que aplicada a uma transição verifica se a *condição de terminação* da iteração é *verdadeira* ou *falsa*;
- $\gamma^f : T^r \cup T^t \rightarrow \{Falso, Verdade\}$ é uma função que aplicada a uma transição verifica se a *condição de falha* da iteração é *verdadeira* ou *falsa*;

Na Figura 6.2 é apresentada a represenção formal do autômato ilustrado na Figura 6.1.

Definição 6.2 (Função de Mudança de Estado) *Seja M um autômato de validação como descrito na Definição 6.1. A função $\tau : Q \times \Pi \times 2^{(S \times Nat)} \rightarrow Q \times 2^{(S \times Nat)}$, a qual controla sua mudança de estados, validando um símbolo de entrada a, $a \in \Pi$, é definida como se segue*[1]*:*

[1] O símbolo $\perp$ indica *valor indefinido.*

$$C = (\Pi, Q, T, \varphi_0, \varphi_1, \rho, q_0, F, \phi, S, \sigma, T^r, T^t, \gamma^r, \gamma^t, \gamma^f)$$

onde:

$$\Pi = \{a, b, c, d, e\},\ Q = \{0, 1, 2, 3, 4, 5, 6, 7, 8, fail\}$$
$$T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}\}$$
$$F = \{8\},\ q_0 = 0,\ S = \{s\},\ T^r = \{t_4, t_5, t_7, t_9\},\ T^t = \{t_6, t_{13}, t_{14}\}$$

$$\gamma^r(t_4) = \gamma^r(t_5) = \gamma^r(t_7) = \gamma^r(t_9) = \neg c\ \&\ \neg d\ \&\ \neg b$$
$$\gamma^t(t_6) = \gamma^t(t_{13}) = \gamma^t(t_{14}) = c\ \mid\ (d\ \&\ b)$$
$$\gamma^f(t_{15}) = \gamma^f(t_{16}) = (\neg d \wedge b)\ \mid\ (d \wedge \neg b)$$
$$\sigma(1) = \{(s, 1)\}, \sigma(4) = \{(s, -1)\}$$

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}
ρ	c	d	a	c	d	e	c	b	d	b	c	d	e	e	λ	λ
φ_0	0	0	0	1	1	1	4	2	4	3	5	6	7	4	1	4
φ_1	1	2	3	1	2	8	1	4	2	5	6	7	8	8	$fail$	$fail$
ϕ	O	O	I	O	O	O	O	I	O	I	O	O	O	O	$-$	$-$

Figura 6.2. Definição Formal do Autômato de Validação Ilustrado na Figura 6.1

$$\tau(e, p, V) = \begin{cases} \bot, & \neg\exists t \mid t \in T \wedge \varphi_0(t) = e \wedge \rho(t) = p & \textit{(erro i)} \\ \bot, & \exists s \in S \mid \neg\exists (s, n) \in V' & \textit{(erro ii)} \\ \bot, & t \in T^r \wedge \neg\gamma^r(t) & \textit{(erro iii)} \\ \bot, & t \in T^t \wedge \neg\gamma^t(t) & \textit{(erro iv)} \\ \bot, & t \in T^r \cup T^t \wedge \gamma^f(t) & \textit{(erro v)} \\ (e', V'), & \textit{caso contrário} & \textit{(transição válida)} \end{cases}$$

onde:

$$\begin{cases} t : t \in T \wedge \varphi_0(t) = e \wedge \rho(t) = p \\ e' : e' \in Q \wedge \varphi_1(t) = e' \\ V' = \{(s, v + i) \mid (s, v) \in V \wedge (s, i) \in \sigma(e') \wedge v + i \geq 0\} \end{cases}$$

A função τ recebe com entrada um estado (e), um identificador de uma porta (p) e uma lista de pares com os valores correntes do semáforo (V), retornando o novo estado do autômato (e') e um novo conjunto de pares que indica os novos valores dos semáforos (V') após efetivação das atualizações associadas ao estado e' atravé da função σ. A função τ não valida a entrada p (está indefinida) caso algumas das condições de erro ocorram:

i) Não existe transição com rótulo p e origem no estado corrente (e), condição que indica que a operação de comunicação sobre a porta p não pode ser efetivada a partir deste estado;

ii) A invariante do semáforo (valor não-negativo) é invalidada pela entrada no estado e'. Em outras palavras, nenhum dos possíveis valores que o semáforo pode assumir no estado resultante (e') é não-negativo;

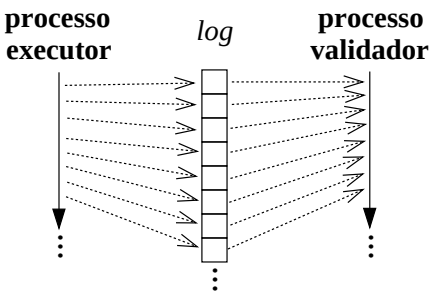


Figura 6.3. Validação Assíncrona

iii) A transição efetivada (t) corresponde à repetição de uma iteração, porém a condição de repetição da iteração é *falsa* naquele instante;

iv) A transição efetivada (t) corresponde à terminação de uma iteração, porém a condição de terminação da iteração é *falsa* naquele instante;

v) A transição efetivada (t) corresponde à repetição ou terminação em uma iteração, porém a condição de falha de sincronização de *streams* é *verdadeira* naquele instante;

Definição 6.3 (Função de Mudança de Estados Extendida) *A função* $\overline{\tau} : Q \times \Pi^* \times 2^{(S \times Nat)} \rightarrow Q \times 2^{(S \times Nat)}$ *aplica-se sobre uma sequência de símbolos* $w \in \Pi^*$ *e um conjunto de valorações para semáforos, retornando o estado resultante e uma nova valoração para os semáforos devido à aplicação de* τ *sobre a sequência* w *de ativações de portas.*

$$\overline{\tau}(q, \epsilon, S) = (q, S)$$
$$\overline{\tau}(q, aw, S) = \overline{\tau}(q', w, S'), \text{ onde } (q', S') = \tau(q, a, S)$$

Definição 6.4 (Linguagem Aceita pelo Autômato de Validação) *Define-se a linguagem aceita por um autômato de validação* M *como:*

$$ACEITA(M) = \{w \mid \overline{\tau}(q_0, w, S \times \{0\}) = (q_f, S \times \{0\}) \wedge q_f \in F\}$$

A linguagem aceita por um autômato de validação correspondente a uma ERCS que descreve o comportamento da interface de um processo indica as sequências válidas de ativações de suas portas. Observe que o valor de todos os semáforos ao final da execução deve ser zero, além do fato de que um dos estados finais deve ser alcançado em uma execução correta do processo. O conjunto das linguagens geradas por todos autômatos de validação válidos é equivalente ao conjunto $\ell_\lambda{}^1$ (linguagens terminais de rede de Petri, definição 5.11).

6.1.2 Validação de Operações de Comunicação pelo Autômato de Validação

Na implementação de um processo do tipo assíncrono, a ordem em que as operações são efetivadas pelo fluxo de controle do módulo funcional do processo em execução pode ser validada dinamicamente de duas formas: *síncrona* e *assíncrona*. Ambas assumem a existência de uma estrutura de dados que implementa um autômato de validação (Definição 6.1), possivelmente implementada como um *tipo abstrato de dados*.

Na forma *síncrona*, a cada operação de comunicação, a função τ é aplicada com o fim de atualizar o estado do autômato de validação e verificar as condições de erro. O estado atual e as valorações corrente dos semáforos devem ser passados a função τ e atualizadas de acordo com o valor retornado devido a aplicação desta função. As condições de erro devem ser testadas antes que a operação seja completada. Caso a operação seja válida, é efetivada a chamada às primitivas de comunicação de mais baixo nível que implementam a operação.

A forma assíncrona, ilustrada na Figura 6.3, emprega dois processos concorrentes: o primeiro, denominado *executor*, executa o processo em si, produzindo um *log* das operações de comunicação efetivadas ao longo de sua execução. O segundo, denominado *validador*, executa a função $\overline{\tau}$ sobre o autômato de validação do processo, usando o *log* como entrada a ser validada. Observe que *executor* e *validador* executam assincronamente. O único ponto de sincronização ocorre quando o *log* encontra-se vazio e o *executor* permanece ativo. Neste caso, o *validador* precisa esperar que uma nova entrada no *log* seja disponibilizada pelo executor, ao modo de uma sincronização do tipo *produtor/consumidor* com *buffer* ilimitado. Ao final da execução, para que o *log* seja válido, o estado atual do *validador* deve ser um estado final do autômato de validação e o valor de todos os semáforos deve ser zero.

A implementação síncrona é mais segura, impedindo que se completem ativações inválidas de portas. Porém, a implementação assíncrona é potencialmente mais eficiente. Embora a validação de uma operação de comunicação seja realizada de maneira atrasada, a execução do *validador* pode sobrepôr-se aos momentos de ociosidade do *executor*, tornando mais eficiente o uso do processador em uma implementação uniprocessada. Além disso, os processadores modernos implementam uma série de extensões que permitem a execução de múltiplas linhas de instrução simultâneas, inclusive com o uso de multiprocessamento. Um compilador poderia fazer uso desses recursos, cada vez mais comuns, minimizando a sobrecarga do *validador* sobre o *executor*, ou mesmo eliminando-a.

O compilador # deve oferecer o suporte para geração de código com ou sem validação, de forma a permitir a otimizar o desempenho do programa, evitando a sobrecarga gerada pelo algoritmo de validação, embora esta seja pouco significativa em programas paralelos # de média e grossa granularidade. A opção *default* neste caso deve ser o uso do validador. Outra opção considerada seria oferecer o suporte dinâmico, através de linha de comando, para desabilitação do validador, ao comando do programador.

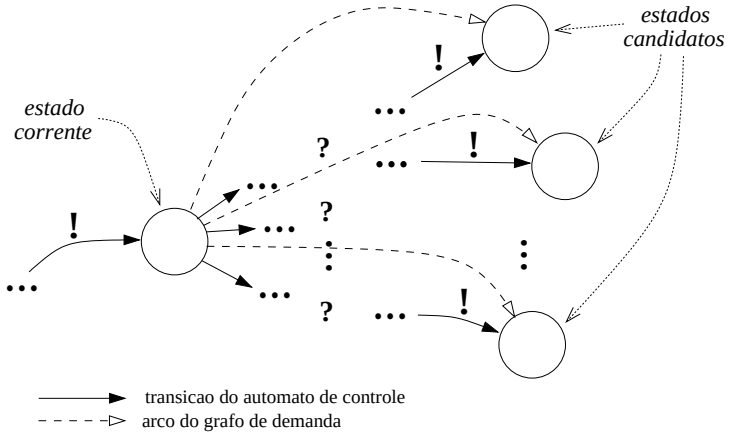


Figura 6.4. Determinando os Arcos do Grafo de Controle de Demanda a Partir do Autômato de Validação

6.1.3 Extendendo o Mecanismo de Validação a Processos Orientados à Demanda e Orientados ao Fluxo de Dados de Entrada

Processos *orientados à demanda* executam sob a demanda pelos valores produzidos nos pontos de retorno do módulo funcional associado. Por este motivo, a ordem de ativação das portas de saída, expressa na descrição do comportamento da interface, deve guiar a execução deste tipo de processo. No caso de processos *orientados ao fluxo de dados*, ao contrário, a execução ocorre de acordo com o provimento de dados aos argumentos do módulo funcional associado. Portanto, a ordem de ativação das portas de entrada deve guiar a execução de um processo orientado ao fluxo de dados na entrada.

Discutiremos em seguida a extensão necessária ao mecanismo proposto para validação dinâmica do comportamento de processos assíncronos para suporte à implementação e validação do comportamento em processos orientados à demanda. O mecanismo necessário para suporte à implementação de processos orientados ao fluxo de dados na entrada é análogo, porém invertendo-se os papéis das operações de entrada e saída. Na definição 6.5, é introduzido o conceito de *grafo de controle de demanda*, usado para controlar a demanda pelos valores de saída do processo e consequentemente a ordem de efetivação das operações de comunicação na execução do processo.

Definição 6.5 (Grafo de Controle de Demanda) *: Seja* $C = (\Pi, Q, T, \varphi_0, \varphi_1, \rho, q_0, F, S, \sigma, T^r, T^t, \gamma^r, \gamma^t, \gamma^f)$ *um autômato de validação. Definimos o* grafo de controle de demanda *de* C, $\Delta(C)$, *da seguinte forma:*

$$\Delta(C) = (\overline{Q}, \overline{T}, \overline{\varphi_0}, \overline{\varphi_1}, \overline{\rho})$$

onde:

- $\overline{Q} = \{q \mid q \in Q \wedge (\exists t \in T \mid \varphi_1(t) = q \wedge \phi(t) = \mathbf{O})\}$. O conjunto de vértices do grafo de controle de demanda corresponde a um sub-conjunto dos estados do autômato de validação correspondente, onde estão incluídos todos os estados para os quais existe uma transição de entrada rotulada com o identificador de uma porta de saída;
- $\forall t \in \overline{T}, \overline{\varphi_0}(t) = e_0 \wedge \overline{\varphi_1}(t) = e_1 \Leftrightarrow (\exists V, V' \in (S \times Nat) \mid \overline{\tau}(e_0, w, V) = (e_1, V'))$. No grafo de controle de demanda, deve existir um arco entre dois vértices sempre que existir um caminho no grafo do autômato de validação correspondente ligando os dois estados correspondentes aos vértices em questão, o qual deve satisfazer a seguinte restrição: sejam $t_1, t_2, \ldots, t_n$ as transições que compõem o caminho. As transições $t_1, \ldots, t_{n-1}$ são associadas a ativação de porta de entrada e t_n é associada a uma ativação de porta de saída. A Figura 6.4 ilustra a determinação do conjunto de arcos do grafo de controle de demanda.

Como se pode observar pela sua definição, o *grafo de controle de demanda* é obtido diretamente a partir do *autômato de validação* associado ao comportamento da interface do processo. Assim, seja A o autômato de validação de um processo, com estado inicial q_0. O grafo de controle de demanda $\Delta(A)$ é obtido pela expressão $demand\_graph(q_0, q_0, \emptyset, A)$, onde $demand\_graph$ é uma função que expressa o algoritmo de construção de um grafo de controle de demanda a partir de uma autômato de validação. Sua definição encontra-se na Equação 6.1.

$$
\begin{gathered}
demand\_graph(q_i, q, Q', A) = \begin{cases} (\overline{Q}, \overline{T}, \overline{\varphi_0}, \overline{\varphi_1}, \overline{\rho}) \cup G^{look\_ahead} \cup G^{restart} & , \ q \notin Q' \\ (\emptyset, \emptyset, \emptyset, \emptyset) & , \ q \in Q' \end{cases} \\
A = (\Pi, Q, T, \varphi_0, \varphi_1, \rho, q_0, F, \phi, S, \sigma, T^r, T^t, \gamma^r, \gamma^t, \gamma^f) \\
Q^{ahead} = \{q' \mid \exists t \in T : q' = \varphi_1(t) \wedge \varphi_0(t) = q \wedge \phi(t) = I\} \\
\begin{gathered} \overline{T} = \{t \mid t \in T \wedge \varphi_0(t) = q \wedge \phi(t) = O\} \\ \overline{Q} = \{q' \mid \exists t \in T : q' = \varphi_1(t) \wedge \varphi_0(t) = q \wedge \phi(t) = O\} \\ \overline{\varphi_0}(t) = q_i, \forall t \in \overline{T} \\ \overline{\varphi_1}(t) = \varphi_1(t), \forall t \in \overline{T} \\ \overline{\rho}(t) = \rho(t) \end{gathered} \\
\begin{gathered} G^{look\_ahead} = \bigcup_{q' \in Q^{ahead}} demand\_graph(q_i, q', Q' \cup \{q\}, A) \\ G^{restart} = \bigcup_{q' \in \overline{Q}} demand\_graph(q', q', Q' \cup \{q\}, A) \end{gathered}
\end{gathered}
\tag{6.1}
$$

O grafo de controle de demanda guia a execução do processo, determinando a ordem em que suas portas de saída são ativadas, o que causa a avaliação dos pontos de retorno associados a estas. A partir do vértice e do grafo de controle de demanda, correspondente ao estado corrente do autômato de validação, os vértices conectados a e por arcos que o têm como origem correspondem aos próximos estados (E') que podem ser alcançados pelo autômato de validação como efeito da efetivação da operação sobre portas de saída associadas às transições que têm como destino os lugares de E', em um caminho que parte de e. A estes estados nos referimos como *estados candidatos*. Seja P o conjunto de

portas de saída correspondente aos estados em E'. Atente para dois fatos importantes para a compreensão do algoritmo:

- Uma mesma porta $p \in P$ pode estar associada a mais de um estado em E';
- As transições intermediárias no caminho entre e e os estados candidatos, em E', validam operações de comunicação sobre portas de entrada, de cujos valores depende o valor a ser produzido para ser enviado pela porta de saída.

Na execução, deve ser escolhida uma dentre as portas em P, chamada a. Não deterministicamente, a porta a é escolhida dentre as portas em P cujo par de comunicação encontra-se ativado, aguardando a ativação de seu par para completar a operação de comunicação. Os estados em E' associados a a formam o conjunto de *estados alvo*. A demanda pelo valor da porta de saída associada aos estados alvo correspondente à porta escolhida força a avaliação do ponto de retorno mapeado à porta em questão, podendo causar a demanda por valores em argumentos do módulo funcional, necessários à computação do valor produzido, o que pode requisitar a ativação de portas de entrada associadas a estes argumentos. Portanto, após o envio do valor através da porta saída ativada, o autômato de validação, utilizando o mecanismo de validação descrito anteriormente, deve atingir um dos *estados alvo*, efetivando uma série de operações de entrada seguidas da operação de saída resultante. Caso contrário, um erro deve ser informado. O estado alvo escolhido corresponde portanto ao estado atual do autômato de controle após a efetivação da porta de saída.

Importantes restrições devem ser observadas com respeito a terminação e a repetição de iterações. Em processos sob demanda, deve-se assumir a restrição de que a última porta ativada antes do teste de uma iteração (combinador **repeat**) seja sempre uma porta de saída. Dessa forma, é possível verificar a condição de terminação da repetição somente após o alcance do *estado alvo* pelo autômato de validação. Caso a condição de terminação seja satisfeita, são considerados estados candidatos somente aqueles alcançáveis a partir da transição que caracteriza a terminação. Caso contrário (repetição), consideram-se aqueles alcançáveis a partir da transição de repetição.

O algoritmo descrito na Figura 6.5, em pseudo-Haskell, ilustra os passos descritos nos parágrafos anteriores. Os autômatos de validação e grafo de demanda são implementados como tipos abstratos de dados, de forma que seu estado corrente e estrutura só podem ser acessados por meio de funções pré-definidas. Evidentemente, este trata-se apenas de um esboço de implementação, sem o compromisso com eficiência necessária em uma implementação final.

6.2 IMPLEMENTANDO HASKELL$_{\#}$

Nesta seção, é apresentada a implementação de Haskell$_{\#}$ (Seção 3.4). Uma das mais importantes e vantajosas características advindas da independência entre os meios de coordenação e computação no modelo # é a facilidade de implementação de linguagens que aderem a este modelo utilizando ferramentas pré-existentes, sem necessidade de modificá-las. Haskell$_{\#}$ é implementada no topo de MPI (*Message Passing Library*)

Sejam: $A = (\Pi, Q, T, \varphi_0, \varphi_1, \rho, q_0, F, \phi, S, \sigma, T^r, T^t, \gamma^r, \gamma^t, \gamma^f)$
$\Delta(A) = (\overline{Q}, \overline{T}, \overline{\varphi_0}, \overline{\varphi_1}, \overline{\rho})$

```
main = do
        initialise_control_automata
        (r_1, r_2, ..., r_k) ← main a_1 a_2 ... a_n
        perform_actions r_1 r_2 ... r_k
    where
        a_i = recv_stream p_i, 1 ≤ i ≤ n

        {- A função recv_stream avalia perform_communication na porta p para obter
           cada elemento da stream, possivelmente aninhada, formando a lista correspondente.
           Como a função e polimorfica no aninhamento da lista retornada, nao a detalhamos. -}
        recv_stream p = ... perform_communication p Nothing ...

        perform_actions [] [] ... [] = if current_state ∈ F
                                       then return ()
                                       else error "Estado final nao alcancado"
        perform_actions r_1 r_2 ... r_k =
                        do
                            q ← current_state
                            (P^c,Q^c) ← (unzip.fetch_candidates) q
                            p ← choose_port P^c
                            Q^t ← {q | (p',q) ∈ zip P^c Q^c ∧ p = p'}
                            i ← exit_point_of p
                            v ← head r_i
                            perform_communication p (Just v)
                            q' ← current_state
                            if q' ∈ Q^t then perform_actions r_1 r_2 ... (tail r_i) ... r_k
                                       else error "Comportamento invalido detectado."


        {- Computa os estados candidatos a partir do estado q, no grafo de demanda -}
        fetch_candidates q = {(ρ̄(t), q') | ∃t : φ̄_0(t) = q ∧ φ̄_0(t) = q' ∧
                                               (t ∈ T^t ⇒ γ^t(t)) ∨ (t ∈ T^r ⇒ γ^r(t))}

        {- Escolhe uma das portas em P cujo par de comunicacao encontra-se ativado -}
        choose_port P = ...

        {- Retorna o indice do valor de retorno ao qual a porta de saida p esta mapeada -}
        exit_point_of p = ...

        {- Retorna o estado corrente do automato de validacao -}
        current_state = ...

        {- Efetiva uma operacao de comunicacao na porta port.
           Atualiza o estado corrente do automato de validacao por meio da funcao τ. -}
        perform_communication port maybe_value = ...
```

Figura 6.5. Código, em Pseudo-Haskell, para a Implementação de Processos sob Demanda

[73], para gerenciamento do paralelismo, e GHC (*Glagow Haskell Compiler*) [102], para compilação de módulos funcionais. O uso destas ferramentas tem um impacto importante para a eficiência e portabilidade de programas desenvolvidos com Haskell$_{\#}$.

Por que MPI ? MPI é atualmente considerada a mais eficiente e portável biblioteca de passagem de mensagens disponível, tendo sido desenvolvida por um comitê científico durante a década de 90. Possui atualmente implementações comeciais e gratuitas, robustas e eficientes, sobre uma extensa variedade de arquiteturas. Neste trabalho, é empregado o LAM-MPI [45].

Por que GHC ? GHC é considerado atualmente o compilador sequencial mais eficiente para a linguagem Haskell, implementando o estado-da-arte das tecnologias envolvidas na compilação de linguagens funcionais não-estritas. Por ser capaz de gerar código C, é bastante portável. O uso de um compilador Haskell sequencial eficiente para compilação de módulos funcionais é um aspecto importante, uma vez que Haskell$_{\#}$ é implementada sob a suposição do paralelismo de média ou grossa granularidade. Em aplicações que possuem essa característica, a maior parte do tempo é desperdiçado na execução de trechos sequenciais do código.

A arquitetura alvo da implementação descrita nesta tese são *clusters* de computadores pessoais equipados com o sistema operacional Linux. O compilador Haskell$_{\#}$ tem sido inteiramente implementado em Haskell. A biblioteca de suporte ao paralelismo de Haskell$_{\#}$ encontra-se implementada em Haskell e C/MPI, utilizando FFI(*Foreign Function Interface*) [55] para interface entre as duas linguagens.

A seção que se segue descreve as etapas do processo de compilação de um programa #. Posteriormente, detalharemos aspectos relevantes sobre a implementação da comunicação entre processos. Uma vez que Haskell é uma linguagem funcional não-estrita que adota o mecanismo de avaliação *lazy*, processos Haskell$_{\#}$ são do tipo *orientado a demanda*, sendo portanto implementados usando o mecanismo descrito na Seção 6.1.3. Uma seção é ainda dedicada a demonstrar como esqueletos MPI são implementados na versão atual de Haskell$_{\#}$.

6.2.1 Visão Geral do Processo de Compilação

No diagrama da Figura 6.6 ilustram-se as etapas do processo de compilação de programas Haskell$_{\#}$, as quais serão descrito nos parágrafos adiante. Como mostrado na Seção 3.2.11, um programa # é descrito por uma *módulo de aplicação*, tipo especial de configuração onde é unicamente declarada a unidade *main* do programa, a qual está associada o *componente de aplicação* que descreve sua funcionalidade. Os parágrafos a seguir descrevem detalhes relevantes sobre as etapas do processo de compilação ilustrado na Figura 6.6.

6.2.1.1 Análise Sintática (*parsing*) do Módulo de Aplicação O primeiro passo do processo de compilação consiste na geração da árvore sintática correspondente à configuração que descreve a aplicação. Para implementação do *parser* #, foi empregada

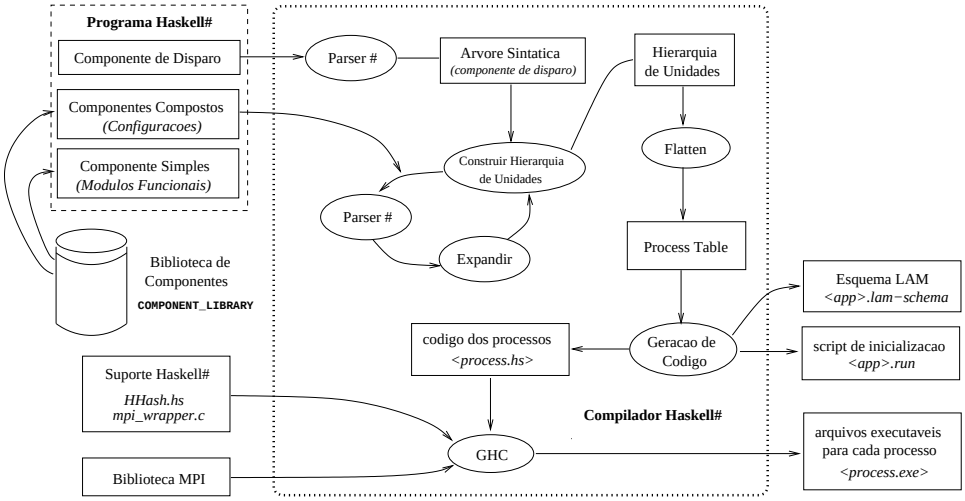


Figura 6.6. Processo de Compilação de Programas Haskell$_{\#}$

a ferramenta *Happy* [164]. O analisador léxico foi implementado por meio da ferramenta *Alex* [75]. Estas constituem os correspondentes Haskell às ferramentas *Yacc*[128] e *Lex*[145], respectivamente, disponíveis para a linguagem C. A árvore sintática produzida nesta etapa constitui entrada ao processo de construção da hierarquia de unidades do programa #, descrito a seguir.

6.2.1.2 Construção da Hierarquia de Unidades O procedimento de construção da estrutura de dados que define a hierarquia de unidades da aplicação é realizada recursivamente a partir de sua unidade *main*. O algoritmo abaixo elucida os passos de tal procedimento:

construa_aglomerado $u \Rightarrow$
1) seja c o componente composto associado a u
2) ache c na biblioteca de componentes
3) *parse* c
4) aplique parâmetros est*á*ticos atuais em c
5) avalie expressões em c
6) expanda c (resolução dos *escopos de variação* ou desindexação)
7) para cada unidade u' que compõe o componente composto c:
7.1) se u' e' um processo então **construa_processo** u'
7.2) se u' e' um aglomerado então **construa_aglomerado** u'
7.3) se u' e' um virtual então **construa_unidade_virtual** u'
8) resolução de argumentos e pontos de saida
9) aplique as operações de unificação, fatoração e replicação em u
10) aplique as nomeações em u

Portanto, o processo é realizado com a avaliação da seguinte expressão, onde *main* é a unidade principal da aplicação.

construa_aglomerado *main*

Note que a estrutura de dados induzida pelo procedimento é uma árvore de unidades do tipo descrito na Seção 3.2.5.3, cujos nós são construídos pelo procedimento **construa_aglomerado** e cujas folhas são construídas pelos procedimentos **construa_processo** e **construa_unidade_virtual**.

No passo 2, o componente associado ao *cluster* é procurado em uma biblioteca de componentes. Esta corresponde a uma estrutura hierárquica (diretórios ou pastas), onde as configuração dos componentes da aplicação em questão e componentes reusáveis, compartilhados com outras aplicações, estão armazenados. Uma variável de ambiente, denominada `COMPONENT_LIBRARY`, indica os diretórios do sistema de arquivos, separados por ponto-e-vírgula, que são pontos de entrada para a biblioteca de componentes do ambiente # de programação. É conveniente que o diretório corrente esteja referenciado no conteúdo dessa variável, uma vez que neste diretório normalmente estão armazenados o componente de aplicação e os demais componentes específicos do programa, caso existam. O compilador oferece ainda uma opção de compilação `-cl`, o qual permite acrescentar novas entradas à biblioteca de componentes.

No passo 6, é aplicado o esquema de desindexação descrito anteriormente na Seção 3.2.10.1, quando é efetivada a resolução das ocorrências de escopos de variação e índices (*unfolding*).

A resolução de argumentos e pontos de retorno do componente composto associado ao aglomerado que está sendo processado é efetivada no passo 7. Corresponde a conexão dos pares de comunicação das portas de entrada e saída da unidade u que encontram-se associadas aos argumentos e valores de retorno do componente c, às portas de entrada e saída associadas associadas a estes por meio de declarações **bind**, na configuração que define c. A Figura 6.7 ilustra este processo.

As operações de unificação, fatoração e replicação (passo 6) são aplicadas antes de qualquer nomeação (passo 7). É verificado se uma unidade aparece como operando em mais de uma operação ou se uma nomeação é realizada para uma unidade virtual que participou de alguma operação como operando. Ambas são operações inválidas que caracterizam a ocorrência de um erro em tempo de compilação.

Após finalizado o procedimento de geração da hierarquia de unidades, a correta aplicação de operações e nomeações deve garantir que todas as unidades virtuais devem ter sido substituídas por unidades não-virtuais (**aglomerados** ou **processos**). Entretanto, a informação que caracteriza as unidades virtuais substituídas é ainda mantida, uma vez que essa informação é usada pelo compilador para geração de código especializado para os esqueletos suportados nativamente, como os esqueletos MPI de comunicação coletiva, cuja implementação será explicada adiante.

6.2.1.3 Construção da Tabela de Processos

A partir da hierarquia de unidades, constrói-se a tabela de processos e canais, a partir de onde é gerado o código MPI do programa. O processo consiste em percorrer a hierarquia de unidades, coletando processos e canais e configurando seus parâmetros. A informação de alto nível sobre a organização topológica provida pelo uso de esqueletos (componentes abstratos) é também armazenada, uma vez que pode ser usada pelo compilador para geração de código mais eficiente e melhor alocação estática dos processos.

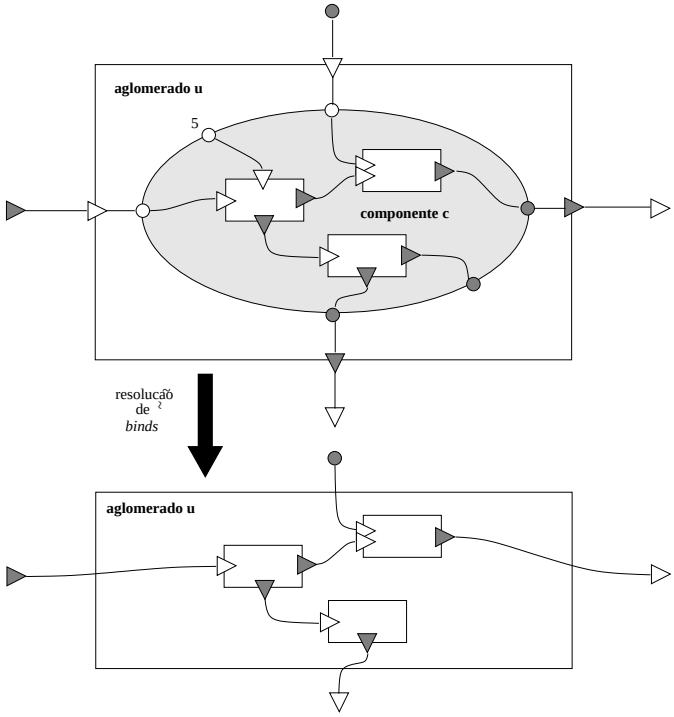


Figura 6.7. Passo Indutivo na Resolução de Argumentos e Pontos de Retorno em Aglomerados

6.2.1.4 Geração de Código A partir da tabela de processos e canais, são gerados os programas Haskell que implementam cada processo. A biblioteca de suporte de Haskell$_{\#}$ (`HHashSupport.hs`) implementa a interface do programa # com a biblioteca MPI, para isso acessando as funcionalidades providas pelo módulo `mpi_wrapper.c`, o qual fornece um conjunto de funções que implementam uma interface abstrata apropriada para acesso as funcionalidades da biblioteca MPI a partir do ambiente #. Para permitir a chamada de funções em C a partir do código Haskell, utilizamos FFI (*Foreign Function Interface*) [55], a interface padrão de Haskell com outras linguagens.

O compilador GHC é encarregado de compilar o código dos processos e módulos funcionais, ligando-os à biblioteca MPI e biblioteca de suporte de Haskell$_{\#}$ para geração do código executável para cada um dos processos. O *autômato de validação* e o *grafo de controle de demanda* são gerados segundo o comportamento especificado para o processo, como um tipo abstrato de dados definido na linguagem C, visando melhor desempenho. O grafo de demanda, como discutido na Seção 6.1.3, é usado para orientar a ativação das portas de saída. O autômato de controle valida a sequência de ativação das portas, segundo o comportamento especificado para o processo em sua interface.

Um arquivo esquema do LAM-MPI (<app>.lam-schema, onde <app> é o nome da aplicação Haskell$_{\#}$) é criado, com a finalidade de configurar a distribuição dos processos nos nós do ambiente virtual LAM. Um arquivo de *script* é usado para disparar o programa paralelo no ambiente LAM, com a chamada ao comando *mpirun*. O usuário é encarregado de criar o arquivo *lamhosts*, onde estão listados os nós do ambiente paralelo que devem ser inicializados através do comando *lamboot*.

6.2.2 Implementando Canais de Comunicação

Em processos Haskell$_{\#}$, canais são entidades identificadas unicamente por uma etiqueta (*tag*), representada por um número inteiro. No código do processo, para cada porta de comunicação, a função *register_port* é chamada com a finalidade de inicializá-la e guardar os parâmetros que definem as características da porta. Um valor inteiro, gerado estaticamente pelo compilador, é usado para identificar unicamente cada porta e deve ser passado como argumento a *register_port*. Este é usado na rotulação das transições do autômato de validação do processo. Os demais argumentos passados à *register_port* fornecem as informações sobre o canal sobre o qual aquela porta encontra-se conectada. São estes:

- Papel da porta (entrada ou saída) no canal;
- *Rank* MPI do processo possuidor da porta conectada a porta em questão (par de comunicação);
- Etiqueta que identifica o canal. O par de comunicação da porta em questão deve registrar uma porta indicando como par de comunicação este processo, com a mesma etiqueta de canal;
- Um *flag* que indica se a porta exige sondagem ou não. O recurso de sondagem permite a implementação de agrupamentos *choice* de portas, o que será explicado adiante.

A função *register_port* está implementada em C. As informações sobre registro de portas são armazenadas em um vetor C, sendo o número inteiro usado para referenciar a porta no código Haskell o índice da porta neste vetor.

6.2.3 Armazenando Valores Haskell em Buffers Contíguos

Valores Haskell são armazenados em uma *heap*, possivelmente de forma não contígua. Ao contrário, valores transmitidos via MPI devem ser armazenados em *buffers* contíguos. Portanto, a preocupação imediata quando na implementação da comunicação através de MPI consiste na tradução de valores Haskell para valores armazenáveis contiguamente em *buffers* C. Esse recurso é atualmente suportado nativamente em GHC, a partir de sua versão 5.0, através da classe de tipos *Storable*, a qual permite armazenar valores Haskell em espaços contíguos de memória, de forma a serem acessados em programas C.

A biblioteca de suporte de Haskell$_{\#}$ implementa instâncias *Storable* para tipos básicos e alguns tipos estruturados comuns em programas Haskell. O compilador é ainda capaz de definir automaticamente instâncias da classe *Storable* para tipos algébricos definidos pelo usuário e tuplas de tamanho maior que 3. Para isso, faz-se necessário a identificação dos tipos Haskell transmissíveis em operações de comunicação, a partir da inferência do conjunto de tipos de dados que constituem a estrutura de tipos estruturados que são protocolos de algum canal. Na dissertação de mestrado do autor desta tese, é apresentado um algoritmo para inferência de tipos transmissíveis [47]. Observe-se que a geração

`SingleIPort ...`	porta de entrada individual
`singleOPort ...`	porta de saída individual
`GroupIPort ...`	agrupamento de portas de entrada SEM aninhamento
`GroupOPort ...`	agrupamento de portas de saída SEM aninhamento
`NestedGroupPort ...`	agrupamento de de portas COM aninhamento

Figura 6.8. Construtores do Tipo Algébrico *PortInfo t u u'*

das instâncias da classe *Storable* para tipos transmissíveis definidos pelo usuário exige a análise sintática do código dos módulos Haskell que constituem a aplicação, onde estes tipos estão definidos. Para isso, utiliza-se o *parser* Haskell 98 fornecido na biblioteca que acompanha o compilador GHC, em sua versão 6.0, usada na implementação em questão.

6.2.4 Implementando Operações de Comunicação (primitivas ? e !)

Os operadores de ativação de portas de entrada e saída, respectivamente '?' e '!', são implementados de forma específica para cada um dos seguintes tipos de portas:

i) portas individuais;

ii) agrupamentos de portas **sem** aninhamento;

iii) agrupamentos de portas **com** aninhamento.

A função *perform_communication* é responsável pela efetivação de uma ativação sobre uma porta. Recebe como argumento um valor algébrico, do tipo *PortInfo t u u'*, cujo construtor indica a natureza da porta. Na Figura 6.8 são apresentados os construtores para cada tipo de porta. As reticências indicam os campos do construtor, os quais encapsulam as informações essenciais sobre a porta para efetivação da operação de comunicação. Dentre estas, destacam-se o *identificador* da porta e sua *função de ligação*. A função *perform_communication* é definida para cada construtor, comportando-se apropriadamente para cada tipo de porta em consideração, como explicado nos próximos parágrafos.

A efetivação da operação de envio sobre uma *porta individual* inicia com a aplicação da função de ligação sobre o valor originado no ponto de retorno, seguido do empacotamento do valor resultante, a ser enviado, no *buffer* MPI associado a porta, através da função *poke*, membro da classe *Storable*. Logo, o valor enviado deve ser de um tipo pertencente a classe *Storable*, o que é criticado pelo próprio verificador de tipos do compilador Haskell (GHC). Após o empacotamento do valor, é executada a primitiva MPI correspondente ao modo de comunicação do canal, a partir das seguintes funções:

- *perform_ssend*: modo **synchronous** (*default*);
- *perform_bsend*: modo **buffered**;
- *perform_rsend*: modo **ready**.

As primitivas MPI são chamadas pelas funções acima utilizando as funcionalidades de FFI. A efetivação da operação de recebimento sobre portas individuais ocorre com a execução da primitiva MPI de recebimento (`MPI_Recv`) seguida do desempacotamento do valor armazenado no *buffer* para o valor Haskell correspondente esperado. A função *perform_recv* é responsável por realizar estas tarefas. Ao valor desempacotado, é aplicada a função de ligação, sendo o valor resultante assumido como argumento ao componente.

A efetivação da operação de comunicação sobre grupos não aninhados é concretizada de maneira análoga a descrita anteriormente para portas individuais, modificando-se apenas a função responsável por efetivar a comunicação. Para grupos *choice*, são usadas as funções *perform_waitany_recv* e *perform_waitany_send*, respectivamente para entrada e saída, enquanto para os demais grupos é empregada a função *perform_waitall*. Estas são implementadas sobre as primitivas MPI de suporte à conclusão múltipla de operações de comunicação. Nesse contexto, as primitivas de envio (assíncronas) usadas para efetivar a comunicação em cada porta individual dentro de grupo de portas de saída, associadas a cada um dos modos de comunicação são respectivamente: `MPI_Issend`, `MPI_Ibsend`, `MPI_Isend` e `MPI_Rsend`. Para portas individuais em um grupo de portas de entrada é empregada a função `MPI_Irecv`. A diferença destas funções para aquelas usadas na comunicação de portas individuais é seu caráter assíncrono. O programa prossegue execução após a execução destas primitivas, mantendo em uma variável um *manipulador* para aquela operação (requisição). A requisição é usada na chamada às funções de conclusão múltipla de MPI. Na implementação de grupos de portas sem aninhamento, são empregadas as primitivas `MPI_Waitany` e `MPI_Waitall`, respectivamente para grupos *choice* e *non-choice*.

A execução das primitivas de comunicação assíncrona para cada porta do grupo não é realizada no momento da execução da operação, de forma a evitar o cancelamento de requisições após a chamada a `MPI_Waitany` e a implementação correta da semântica do grupo *choice*. Ao registrar-se uma porta (ver Seção 6.2.2), um *flag* é usado para indicar se uma porta é sondada (True) ou não (False). Para portas sondadas, no momento de seu registro, é efetivada a chamada assíncrona à primitiva de comunicação correspondente e guardada uma referência à requisição retornada como efeito da sua chamada. Qualquer porta individual pertencente a um grupo de comunicação deve ser sondada obrigatoriamente. Portas individuais não devem ser sondadas. A requisição resultante da execução da operação assíncrona é mantida para uso posterior. No caso de grupos *non-choice*, no momento da comunicação, as requisições associadas às portas do grupo são usadas como argumentos à primitiva `MPI_Waitall`. Espera-se a conclusão de todas as operações de comunicação para dar prosseguimento a execução, com a posterior realização de novas requisições, com a chamada às primitivas de comunicação assíncrona correspondentes a cada porta, uma vez que as requisições atendidas são invalidadas. Portas *choice*, por outro lado, efetuam chamada à primitiva `MPI_Waitany` de forma que é aguardada a conclusão de uma entre as várias requisições passadas como parâmetro, sob o total controle do MPI. A efetivação de nova requisição somente precisa ser realizada para a porta associada a requisição concluída. Observe-se que a requisição pode ser vista como uma sonda, usada para, além de efetivar a operação de comunicação, verificar o estado do canal.

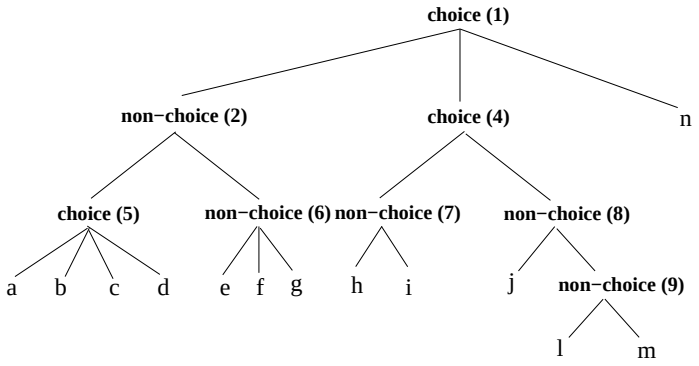


Figura 6.9. Grupos Aninhados Representados como Árvores

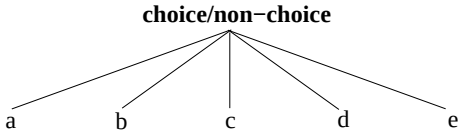


Figura 6.10. Grupo Não-Aninhado Representado como uma Árvore

Em grupos com aninhamento, grupos *choice* podem aparecer aninhados a grupos *non-choice* e vice-versa, arbitrariamente e em qualquer profundidade. Grupos aninhados podem então ser modelados como árvores, de tal modo que cada folha corresponde a uma porta individual pertencente a um grupo e cada nó caracteriza um grupo. Os nós são rotulados como *choice* ou *non-choice*, dependendo da natureza de sua função de ligação.

A árvore apresentada na Figura 6.9 modela a estrutura de um grupo aninhado hipotético. A usaremos no auxílio à descrição informal do algoritmo usado para ativação de um grupo de portas COM aninhamento.

Para um grupo de portas SEM aninhamento do tipo *choice*, o qual pode ser modelado como uma árvore com um único nó (profundidade 1), a escolha da porta a ser ativada corresponde a escolha de um dos ramos da árvore, correspondente à porta. Como vimos, o caminho escolhido é aquele correspodente a uma das portas cujo par conectado por meio de um canal encontra-se pronto para comunicação, obedecendo a semântica da primitiva `MPI_Waitany`. Por outro lado, para grupos sem aninhamento de tipo *non-choice*, todos os caminhos da árvore são assumidos, uma vez que os valores de todas as portas são necessários. A Figura 6.10 ilustra esse fato.

Para grupos COM aninhamento, é necessário generalizar o algoritmo de escolha em grupos de aninhamento 0 (sem aninhamento) descrito no parágrafo anterior, permitindo a escolha de portas em um grupo *choice*, assumindo que outros grupos (*choice* ou *non-choice*) possam estar aninhados a ele, como o é o caso, na Figura 6.9, dos nós 1 e 3 (Nota-se que o nó 4 pode ser tratado como um grupo *choice* SEM aninhamento).

A idéia contida no algoritmo de resolução dinâmica de grupos *choice* é simples. A partir do grupo raiz, percorre-se a estrutura de grupos aninhados recursivamente, segundo a seguinte regra: para cada grupo *choice*, é escolhido um dos ramos que dele partem e

este é percorrido. Caso seja este ramo um grupo aninhado, percorra-o, caso contrário ative a porta individual. Por outro lado, para cada grupo não-*choice* todos os ramos são considerados. Observe-se que ao final, são ativadas somente as portas individuais atingidas.

Surge então a questão de como efetuar a escolha entre grupos de portas. Considere a seguir o grupo aninhado na Figura 6.9. O ramo escolhido corresponde a um dentre aqueles que possuem pelo menos 1 porta pronta para comunicação. Por exemplo, para o grupo (nó da árvore) 1, se, no momento da comunicação, a única porta pronta for a porta e, o ramo esquerdo é escolhido. A escolha da porta d significa que as portas que aparecem no ramo direito da árvore (h, i, j, l, m e n) não deverão ser ativadas. Portanto, após a escolha da porta d, o estado destas não precisam mais ser verificado com `MPI_Waitany`. Por outro lado, as portas f e g devem ser lidas obrigatoriamente, com a chamada a primitiva `MPI_Waitall`, pois não encontram-se no contexto de nenhuma escolha no ramo escolhido (esquerdo), ao contrário das portas a, b e c, no contexto do grupo *choice* 4, as quais precisam ser verificadas com `MPI_Waitany` na próxima iteração do algoritmo. Se por exemplo, a porta b for a escolhida na próxima iteração, o algoritmo pára, pois não há mais nenhum grupo *choice* a ser resolvido. As funções de ligação de cada grupo de portas são aplicadas segundo as restrições impostas pela natureza das portas agrupadas.

O algoritmo de ativação de grupos de portas COM aninhamento emprega duas funções, definidas automaticamente pelo compilador ao analisar a estrutura do grupo em questão: *which_to_discard* e *which_to_force*. A primeira recebe como argumento uma porta e retorna aquelas que são descartadas quando esta é escolhida. Estas correspondem às portas que aparecem no contexto dos ramos que partem nos nós *choice* que surgem no caminho da porta até a raiz da árvore, e que não a contém. Para a porta i, por exemplo, estes nós correspondem aos nós 3 e 1, enquanto para a porta a correspondem aos nós 4 e 1. Por outro lado, a segunda função retorna aquelas portas cuja ativação deve ser forçada, correspondentes aquelas que aparecem no contexto em grupos *non-choice* adjacentes à porta e não intercaladas por um grupo *choice*. Como exemplo, apresentamos, na Tabela 6.1, a definição das funções *which_to_discard* e *which_to_force* para o grupo apresentado na Figura 6.9. O algoritmo usado para ativação das portas de um grupo COM aninhamento pode ser resumido da seguinte forma:

$ANY \leftarrow$ Todas as portas do grupo
enquanto $ANY \neq \emptyset$
  $\underline{\text{choice}} \leftarrow$ `mpi_waitany` ANY
  $\underline{\text{discarded}} \leftarrow$ *which_to_discard* $\underline{\text{choice}}$
  $\underline{\text{forced}} \leftarrow$ *which_to_force* $\underline{\text{choice}}$
  $ANY \leftarrow \{a \mid a \in ANY \wedge a \notin \{\underline{\text{choice}}\} \cup \underline{\text{discarded}} \cup \underline{\text{forced}}\}$
  $ALL \leftarrow ALL \cup \underline{\text{forced}}$
fim-enquanto
`mpi_waitall` ALL

6.2.4.1 Implementando a Escolha do Estado Alvo na Execução Guiada pelo Grafo de Controle de Demanda O algoritmo de resolução de grupos *choice* pode ser facilmente adaptado para implementação da escolha dos *estados alvo*, dentre os estados candidatos, na execução guiada pelo grafo de controle de demanda (Seção 6.1.3). Neste

which_to_discard	a	b	c	d	e	f	g	h	i	j	l	m	n
a		X	X	X				X	X	X	X	X	X
b	X		X	X				X	X	X	X	X	X
c	X	X		X				X	X	X	X	X	X
d	X	X	X					X	X	X	X	X	X
e								X	X	X	X	X	X
f								X	X	X	X	X	X
g								X	X	X	X	X	X
h	X	X	X	X	X	X	X			X	X	X	X
i	X	X	X	X	X	X	X			X	X	X	X
j	X	X	X	X	X	X	X	X	X				X
l	X	X	X	X	X	X	X	X	X				X
m	X	X	X	X	X	X	X	X	X				X
n	X	X	X	X	X	X	X	X	X	X	X	X	

which_to_force	a	b	c	d	e	f	g	h	i	j	l	m	n
a					X	X	X						
b					X	X	X						
c					X	X	X						
d					X	X	X						
e						X	X						
f					X		X						
g					X	X							
h									X				
i								X					
j											X	X	
l										X		X	
m										X	X		
n													

Tabela 6.1. Funções *which_to_discard* e *which_to_force* para o grupo representado pela árvore na Figura 6.9

caso, o conjunto de portas associadas às transições que chegam a cada estado candidato no caminho em questão atuam como se formassem a um grupo *choice*. O ramo escolhido a partir da raiz da árvore indica a porta escolhida, a qual está associada a um conjunto de estados alvo, como discutido na Seção 6.1.3.

6.2.5 Implementando os Esqueletos MPI para Comunicação Coletiva

Esqueletos constituem a principal ferramenta para geração de programas eficientes provida pelo modelo #, a despeito do elevado nível de abstração inerente a este modelo. Como vimos, através de seu uso disciplinado, o programador é capaz de informar ao compilador sobre características topológicas e comportamentais da aplicação que não poderiam ser inferidas automaticamente e que podem ser usadas pelo compilador para geração de código apropriado. Esqueletos podem ainda ser usados para implementação de bibliotecas paralelas, especialmente aquelas voltadas à computação científica, sendo essa uma de suas principais motivações.

O compilador # deve oferecer o suporte nativo ao esqueleto utilizado. Vislumbra-se ainda o suporte a facilidade de ensinar-se ao compilador com o gerar o código apropriado para um determinado esqueleto, utilizando a noção de *template*. Usando tal ferramenta de alto nível, seria possível ao usuário a flexibilidade de extender as funcionalidades do ambiente # de programação, com o desenvolvimento de bibliotecas paralelas voltadas às suas aplicações específicas, porém implementadas no topo do compilador # com suporte nativo. Outra possibilidade é o desenvolvimento de componentes que abstraiam as funcionalidades de bibliotecas de computação científica pré-existentes, sequenciais ou paralelas, como PetsC, Linpack, Scalapack, Reduce, etc. O uso destes componentes, ou esqueletos, implicaria a geração de código apropriado com chamadas às primitivas destas

#	**MPI**
BCast	`MPI_BCast`
Gather	`MPI_Gather`
Gatherv	`MPI_Gatherv`
AllGather	`MPI_Allgather`
AllGatherv	`MPI_Allgatherv`
Scatter	`MPI_Scatter`
Scatterv	`MPI_Scatterv`
AllToAll	`MPI_AllToall`
AllToAllv	`MPI_AllToallv`
Reduce	`MPI_Reduce`
AllReduce	`MPI_AllReduce`
Reduce_Scatter	`MPI_Reduce_scatter`
Scan	`MPI_Scan`

Tabela 6.2. Esqueletos MPI

bibliotecas, de forma totalmente abstrata ao ambiente de programação, onde estes são tratados com um componente # qualquer.

Nesta seção, demonstraremos as potencialidades citadas no parágrafo anterior mostrando como um conjunto específico de esqueletos de grande utilidade em programas científicos que fazem uso de MPI foram implementados nativamente no compilador Haskell$_{\#}$. Estes abstraem o uso das primitivas de comunicação coletiva suportadas por esta biblioteca, permitindo que sejam efetivamente empregadas no código gerado pelo compilador. O objetivo é demonstrar a viabilidade do uso de esqueletos para geração de código específico e apropriado em um determinada aplicação. A tabela 6.2 enumera os esqueletos MPI, associando-os às suas correspondentes primitivas MPI.

As primitivas de comunicação coletiva de MPI pressupõem que os processos participantes da comunicação estão organizados em um *grupo*. Nos termos de MPI, um grupo é uma coleção ordenada de processos, aos quais é associado um identificador único, o *rank*. A um grupo, deve ser associado um *comunicador*[2], o qual é a realização daquilo que MPI define com o termo *contexto*, espaços disjuntos onde mensagens transitam. Uma mensagem que trafega em um contexto não pode ser enxergada em outro contexto. Grupos, comunicadores e contextos são abstrações criadas para o suporte à construção de bibliotecas científicas com MPI. Além disso, são usadas com o fim de permitir que grupos de processos efetuem operações coletivas entre si.

O compilador # deve ser hábil a verificar na tabela de processos as ocorrências de esqueletos MPI e descobrir os processos participantes em cada um destes. Para isso, basta verificar se o processo foi nomeado a alguma unidade virtual do esqueleto MPI em questão. Para cada esqueleto, deve-se ser então gerada uma chamada à rotina de inicialização do padrão coletivo de comunicação, denominada `mpi_create_<`*`esqueleto`*`>_comm`, onde <*esqueleto*> é o nome do esqueleto, iniciado com letra minúscula. Por exemplo, considere o kernel IS, discutido na Seção 4.2.5.2, em sua instância para 8 processadores. As seguintes linhas são geradas no código # de seus processos:

[2] *Intra-communicator.*

```
comm_bs = mpi_create_allreduce_comm 8 ((unsafePerformIO.pack_list) [0::Int,1,2,3,4,5,6,7]) bufsize_bs
                                      DATATYPE_MPI_INT REDUCEOP_MPI_SUM offset_bs

comm_kb = mpi_create_alltoallv_comm 8 ((unsafePerformIO.pack_list) [0::Int,1,2,3,4,5,6,7]) bufsize_kb
```

Os *bindings* (tipo *Int*) `comm_bs` e `comm_kb` são *manipuladores* (*handles*) para entradas onde encontram-se armazenadas as informações sobre o contexto do grupo de comunicação criado para cada ocorrência de esqueleto. No caso de IS, são duas estas ocorrências, respectivamente dos esqueletos AllReduce e AllToAllv, como se pode observar no código. Uma lista que contém os *ranks* dos processos participantes da comunicação é passada como argumento, de forma a permitir a criação do grupo e de um comunicador para o contexto de comunicação envolvido em cada ocorrência do esqueleto. Para a inicialização do esqueleto AllReduce é ainda necessário prover o tipo e operação MPI empregado na redução dos valores transmitidos entre os processos. O uso de um esqueleto MPI de comunicação coletiva em um processo induz a existência de uma porta de entrada e uma porta de saída, as quais não precisam ser diretamente referenciadas na configuração, como pode ser observado no código # de IS apresentado na Figura B.2. A mesma observação é válida aos kernels EP e CG e às aplicações LU, SP e BT, de NPB. Para isso, existe uma referência na interface das unidades de IS (*IIS*) para sub-interfaces denominadas *bs* e *kb*, as quais herdam as portas das interfaces *IAllReduce* e *IAllToAllv*, as interfaces das unidades virtuais contidas em AllReduce e AllToAllv. No comportamento da interface *IIS*, o uso do combinador **do** indica que o comportamento da sub-interface indicada deve ser copiado literalmente na posição onde ocorre. Em termos de esqueleto MPI, significa que naquela posição a operação de comunicação coletiva é efetivada. No autômato de validação do processo, a interface do esqueleto MPI na interface do processo é tratada como uma única porta, ativada pelo combinador **do**, sendo tratada como uma porta de saída. Esse mesmo recurso foi usado na modelagem de esqueletos MPI com redes de Petri (Seção 5.3). Entretanto, a função *perform_communication* possui comportamento especial sobre portas virtuais induzidas por operações coletivas. No caso do *kernel* IS, as comunicações coletivas são efetivadas pelas seguintes chamadas:

```
perform_communication (ALLREDUCE comm_bs 8 (AllOS id) (AllIS id) (StreamPort 0))

perform_communication (ALLTOALLv comm_kb 8 (AllOG distribute_keys) (AllIG combine_keys) (StreamPort 0))
```

Os identificadores *id*, *distribute_keys* e *combine_keys* são funções de ligação. O número 8 indica a quantidade de processos participantes da comunicação. Observe que os respectivos manipuladores dos esqueletos (*comm_bs* e *comm_kb*) são também passados como argumentos.

O recurso de esconder portas é usada na definição das interfaces das unidades dos esqueletos MPI com o fim de evitar que o programador, ao definir a interface de unidades em uma aplicação que fazem uso de esqueletos MPI, faça referências de ativação explícitas sobre as portas da sub-interface MPI herdada. Isso pode ser observado no código que define a topologia destes esqueletos, exposto no Apêndice A. Portanto, na interface *IIS*, é vedado ao programador referenciar as portas *bs.in*, *bs.out*, *kb.in* e *kb.out* na definição do comportamento desta, restando somente a opção de utilizar o combinador **do** sobre as sub-interfaces *bs* e *kb*. O compilador deve prover ainda a funcionalidade de geração de

código ponto-a-ponto, ao invés de código coletivo, utilizando as configurações originais dos esqueletos, caso o programador assim deseje. Em certas circunstâncias, o uso de primitivas ponto-a-ponto parece ser mais eficiente que o uso de primitivas de comunicação coletiva.

6.2.5.1 Efetivando as Operações de Comunicação Coletiva Como ilustrado nos parágrafos anteriores, as operações de comunicação coletiva são efetivadas pela chamada a função *perform_communication*, assim como ocorre com a comunicação ponto-a-ponto em portas individuais e agrupamentos de portas discutido na Seção 6.2.4. Para isso, faz-se necessário extender o tipo algébrico *PortInfo* com construtores associados a cada tipo de operação coletiva: `BCAST`, `GATHER`, `GATHERv`, `ALLGATHER`, `ALLGATHERv`, `SCATTER`, `SCATTERV`, `ALLTOALL`, `ALLTOALLv`, `REDUCE`, `ALLREDUCE`, `REDUCE_SCATTER`, `SCAN`. Estes construtores encapsulam em seus campos as informações necessárias a efetivação da operação de comunicação, resumidas a seguir:

- Manipulador da operação coletiva;
- Funções de ligação para as portas de entrada e saída envolvidas;
- Tipo da porta: *stream* ou não-*stream*.

Deve-se lembrar que, embora as informações providas no valor algébrico que define a operação coletiva assumam a existência de uma porta de entrada e uma porta de saída na interface do processo, nem sempre isso ocorre. Este é o caso dos esqueletos de BCast, Gather, Gatherv, Scatter e Scatterv, onde podem existir unidades com porta única. Entretanto, isso é resolvido pela função *perform_communication*, utilizando a função *am_i_root*, capaz de verificar se uma unidade é raiz ou não. Nesse caso, uma unidade que só utiliza uma das duas portas, simplesmente ignora a informações que caracterizam a outra. Vale ressaltar que em esqueletos que possuem porta de entrada e saída, dois *buffers* de comunicação são necessários em cada processo: um para recebimento e outro para envio.

Alguns esqueletos possuem características peculiares que merecem atenção. Por exemplo, a efetivação de uma operação associada a ocorrência de um esqueleto AllToAllv exige a chamada a duas primitivas de MPI: `MPI_AllToall`, seguida de `MPI_AllToallv`. A primeira chamada é necessária para que cada processo informe os demais sobre o número de elementos que cada processo receberá. Essa informação é necessária para configuração dos parâmetros da chamada principal, a `MPI_AllToallv`. Na prática, a própria programação MPI, em baixo nível, exige o uso deste recurso, como pode ser observado no kernel IS. Em seu código C original, a chamada à `MPI_AllToallv` vêm precedida de uma chamada à `MPI_AllToall`. Portanto, em sua implementação #, ao usar-se o esqueleto AllToAllv nestas circunstâncias, são mantidas as característcas de comunicação do programa original, embora de forma mais abstrata do que como usado no código C original.

Nos esqueletos Reduce, AllReduce, Reduce_Scatter e Scan é necessário informar o tipo de dados e operação MPI aplicado na operação de redução que ocorre como

efeito da comunicação. Na definição destes esqueletos (Apêndice A), observa-se que estes são informados como parâmetros estáticos. Note-se que, na configuração apresentada, estes parâmetros não tem nenhuma utilidade. São portanto usados somente para informar ao compilador. Tais parâmetros podem ser usados para prover parâmetros adicionais ao compilador, em nível de configuração do esqueleto.

6.3 AVALIAÇÃO DE DESEMPENHO

Nesta seção, os programas IS, EP, CG introduzidos na Seção 4.2.5 e cujo código encontra-se ilustrado no Apêndice B, serão usados para avaliação de desempenho da implementação de Haskell$_{\#}$ descrita nas seções anteriores deste capítulo. O motivo pelo qual foi escolhido este sub-conjunto de NPB deve-se ao emprego efetivo nos programas nele contidos de operações coletivas com transmissão de grandes massas de dados a cada iteração. Dessa forma, é possível avaliar de maneira mais precisa o ganho de desempenho advindo do uso de esqueletos MPI.

O parâmetro analisado é a escalabilidade do desempenho destas aplicações de acordo com o número de processadores usados na solução do problema, mantendo-se fixo o tamanho do problema. Especial atenção é reservada à análise da sobrecarga de comunicação imposta pelo gerenciamento do paralelismo em Haskell$_{\#}$. Embora a implementação apresentada não seja a mais eficiente que possa ser construída para esta linguagem, serve como protótipo para analisar o potencial de desempenho para implementações mais eficientes, estabelecendo um limitante inferior. Contudo, resultados promissores têm sido obtidos, como apresentado adiante. As seções que se seguem descrevem o ambiente de *hardware* e *software* usado para realização das medições apresentadas, seguindo-se a explanação da metodologia empregada e análise e discussão dos resutados obtidos.

6.3.1 Plataforma de Hardware

As medições de desempenho foram realizadas em um *cluster* constituído por dezesseis (16) computadores pessoais, equipados com processadores Intel Pentium IV (2 GHz), conectados através de uma rede Ethernet a velocidade de 100Mbs. Com a finalidade de adaptar-se os recursos disponíveis aos diferentes requisitos de memória inerentes aos programas e instâncias de problemas considerados na avaliação, a configuração de memória do *cluster* foi realizada da seguinte forma: O nó mestre possui 1GB de memória, usado para execução de instâncias maiores de problemas em suas versões sequenciais. Quatro nós possuem 512MB de memória e foram usados para medições com dois (2) e quatro (4) processadores. Os demais nós possuem 256MB de memória e foram usados nas medições com oito (8) processadores.

6.3.2 Ambiente de Software

O sistema operacional Linux Red Hat 8.0 é o sistema operacional empregado nos nós do *cluster*. O protocolo TCP/IP é o protocolo de comunicação entre-nós utilizado. O ambiente de *software* necessário para execução de programas Haskell$_{\#}$ inclui alguma

versão do LAM-MPI [45]. Particularmente para as medições efetivadas para esse trabalho, foi empregada a versão 6.3.9 desta ferramenta. O compilador GHC, versão 6.0, foi usado para compilação dos módulos funcionais dos programas em questão.

6.3.3 Restrições e Metodologia

Uma importante restrição a ser considerada na elaboração da metodologia empregada nesta análise diz respeito a maior granularidade dos processos nas versões Haskell$_{\#}$ em relação às versões MPI dos programas considerados, para um mesmo tamanho de problema. Programas Haskell, por ser esta uma linguagem funcional *lazy*, são significativamente menos eficientes que programas C e Fortran para execução de programas científicos com requisitos de alto desempenho, seja sob o ponto de vista de seu comportamento espacial (uso da memória) ou temporal (tempo de execução), a despeito dos significativos avanços obtidos desde a última década na tecnologia de compilação de linguagens funcionais não-estritas. Assim, com a finalidade de evitar a sugestão de falsas conclusões, omitimos nas medições adiante os tempos de execução obtidos para as versões originais Fortran e C, com MPI, das aplicações analisadas, uma vez que alguém poderia arguir que o maior custo computacional dos processos em programas Haskell$_{\#}$ poderia esconder o maior tempo gasto por estes programas no gerenciamento do paralelismo (maior granularidade), tornando os resultados pouco conclusivos com respeito a eficiência da linguagem avaliada, causando ainda a ilusão de ser Haskell$_{\#}$ mais eficiente que MPI puro no gerenciamento do paralelismo. Entretanto, não é realista a expectativa de implementar-se um mecanismo de programação paralela de alto nível de forma mais eficiente do que o mecanismo de mais baixo nível sobre o qual foi implementado, no que diz respeito a sobrecarga no gerenciamento do paralelismo. Haskell$_{\#}$ é implementada no topo de MPI, sendo obviamente a sobrecarga de gerenciamento do paralelismo desta biblioteca um limitante inferior para a eficiência de programas Haskell$_{\#}$. Outra observação importante é que a implementação apresentada neste trabalho é considerada um protótipo, não tendo portanto a intenção de afirmar-se como a forma mais eficiente de implementar-se esta linguagem.

A despeito dos fatos expostos anteriormente, foram obtidos resultados bastante satisfatórios de desempenho para as aplicações analisadas, que pode ser ainda melhorada em implementações de mais baixo nível do modelo #. É importante ainda ressaltar a intenção de que futuras implementações do modelo # incluirão a possibilidade de que módulos escritos em outras linguagens poderiam gerar código C/MPI diretamente ao invés de Haskell, sem o emprego de funções de alta ordem e *lazy* evaluation, como empregado na implementação apresentada neste trabalho.

A diferença computacional entre as duas classes de linguagens causa ainda uma segunda restrição. Para os programas EP, IS, CG, é necessário ainda escolher um tamanho apropriado para a instância de problema usada na medição. NPB oferece instâncias de problema padrão, identificadas por S (*Sample*), W (*Workstation*), A, B e C, enumeradas em ordem de complexidade computacional. Entretanto, a partir de uma certa instância de problema, a memória necessária para sua execução sequencial supera a memória física disponível no computador empregado nas medições. Isso causa o uso extensivo

KERNEL	TAMANHO DE PROBLEMA 1	TAMANHO DE PROBLEMA 2
EP	$m = 25$	$m = 28$
IS	$total\_keys\_log2 = 21$ $max\_key\_log2 = 17$	$total\_keys\_log2 = 22$ $max\_key\_log2 = 18$
CG	$na = 14000$ $nonzer = 11$ $niter = 15$	$na = 18000$ $nonzer = 12$ $niter = 45$

Legenda ↓

PARÂMETRO	SIGNIFICADO
$total\_keys\_log2$	O número de chaves a serem ordenadas é $2^{total\_keys\_log2}$
max_key_log2	Os valores das chaves é entre 1 and $2^{max\_key\_log2}$
na	Dimensão da matriz esparsa é $na \times na$
$nonzer$	Controla o número de elementos não-zero nas linhas
$niter$	Número de iterações do laço mais externo
m	

Tabela 6.3. Instâncias de Tamanho de Problema Aplicadas a cada Programa

da memória virtual, o qual deve ser evitado em qualquer avaliação de programa paralelo. Uma vez que programas Haskell efetivamente usam significativamete mais memória que suas versões C, o tamanho da máxima instância de problema tratada pela versão Haskell sem que haja necessidade de recorrer-se ao uso da memória virtual é bem menor que aquela suportada pela versão C ou Fortran.

As medidas foram realizadas segundo a abordagem CPS[3] (tamanho de problema constante). Observando as limitações descritas acima, duas instâncias de tamanho de problema foram escolhidas para cada programa (Tabela 6.3). A maior delas foi escolhida de forma a aproximar-se a exaustão dos recursos de memória física do computador paralelo. Utilizamos números de processadores em potências de 2. Assim, dada a quantidade de processadores disponíveis, realizamos as medições para 2, 4 e 8 processadores, um requisito da maioria dos programas que compõem o NPB. As figuras de desempenho são apresentadas na Figura 6.11 (primeira instância) e 6.12 (segunda instância). Os gráficos à esquerda descrevem a evolução do tempo de execução (em segundos) para cada programa, de acordo com o número de processadores. Os gráficos à direita descrevem as curvas de *speedup* associadas. Os números nos eixos verticais correspondem aos valores mensurados. A linha tracejada nos gráficos que descrevem *speedup* identifica a curva de *speedup* linear (ótimo). Nos gráficos apresentados, as linhas cheias e pontilhadas correspondem aos valores de desempenho mensurados, respectivamente considerando-se e desprezando-se o tempo despendido pelo programa funcional com o gerenciamento dinâmico de memória [130][4]. Os resultados são analisados adiante.

[3] *Constant Problem Size.*

[4] *Garbage collection.*

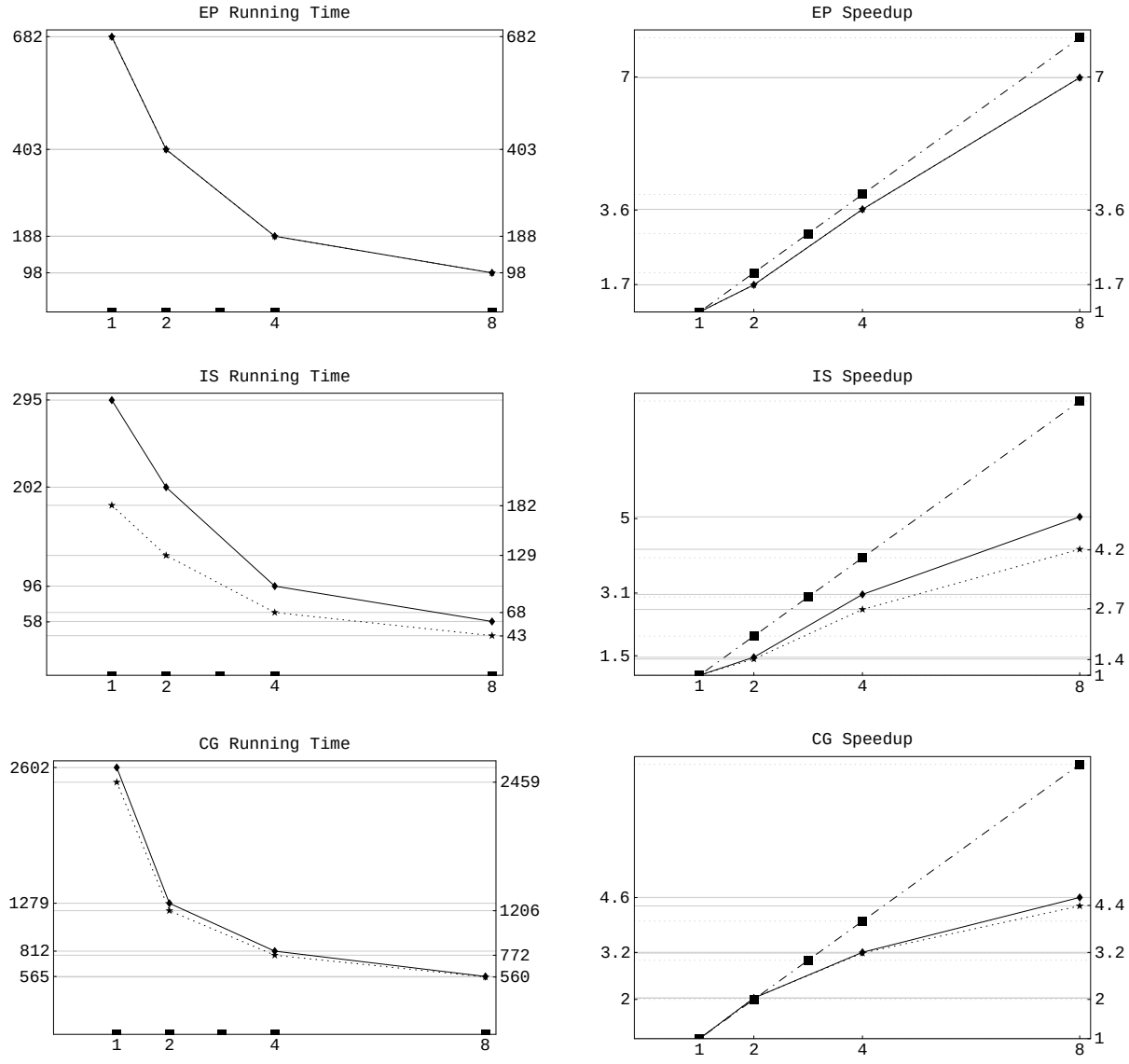


Figura 6.11. Figuras de Desempenho para Versões # dos programas EP, IS e CG (Caso 1)

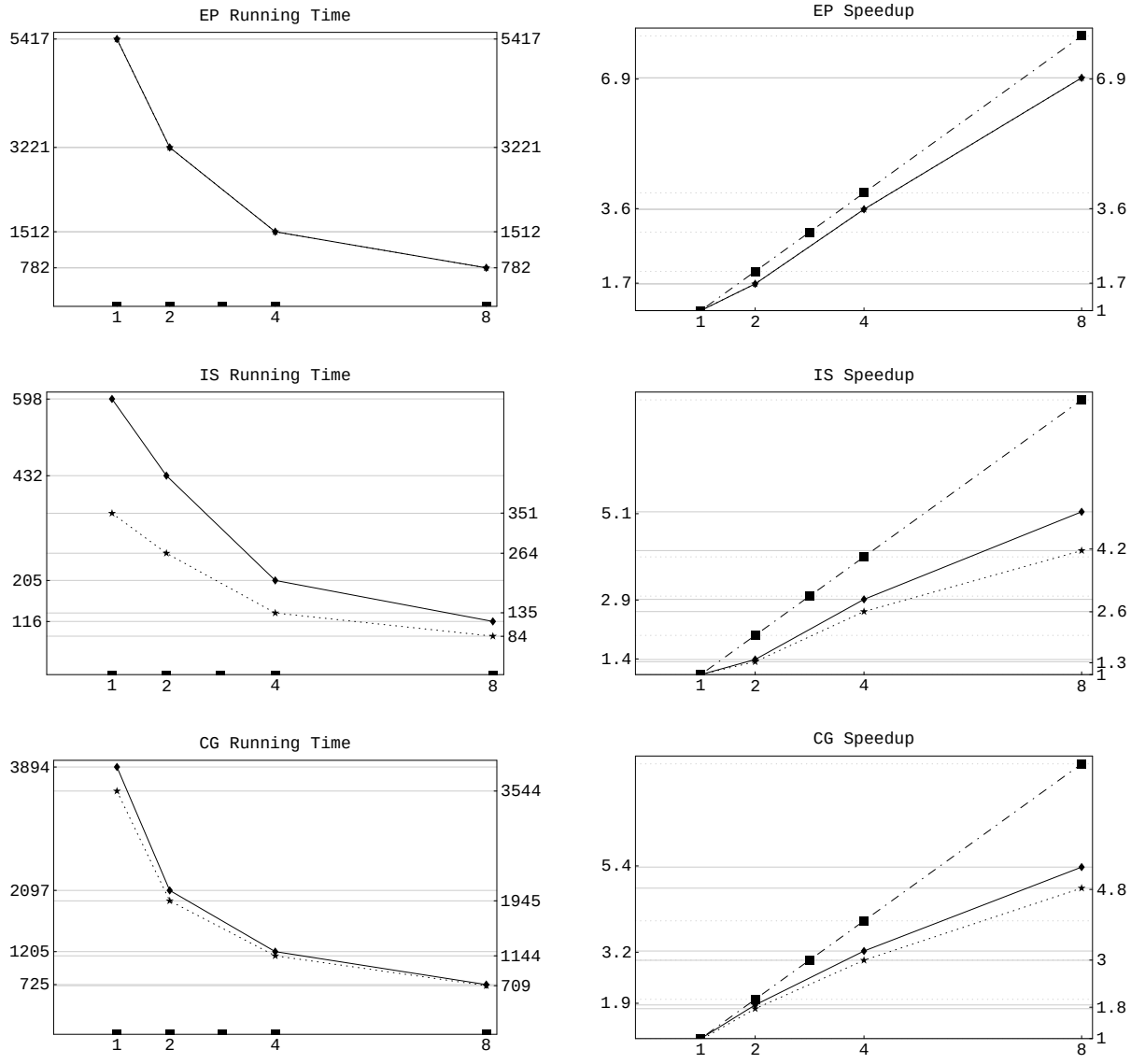


Figura 6.12. Figuras de Desempenho para Versões # dos programas EP, IS e CG (Caso 2)

6.3.4 Resultados e Discussão

A curva de *speedup* observada para o kernel EP é aproximadamente linear. Esse fato era esperado, uma vez que neste programa a comunicação ocorre em uma única vez ao final da execução, após um grande período despendido com computações numéricas realizada em paralelo. O mesmo comportamento foi observado ao mensurarmos o desempenho de sua versão Fortran/MPI original.

Nos kernels IS e CG, a curva de *speedup* obtida não aproxima-se ao caso linear. Ainda assim pode ser considerada satisfatória, tendo em vista as características da rede de comunicação do *cluster* empregado. Propositalmente, este foi configurado com mínimas otimizações para execução paralela de alto desempenho, tendo em vista que uma das premissas do desenvolvimento de Haskell$_{\#}$ é o seu uso potencial em arquiteturas paralelas com alta latência de comunicação, quando comparada à supercomputadores convencionais. Vale ainda ressaltar que os resultados obtidos usando as versões originais de IS e CG foram ainda mais distantes em relação ao caso linear. Evidentemente isso deve-se a menor granularidade dos processos, como discutido anteriormente. Por isso, resolvemos omiti-los para evitar falsas conclusões.

Procedemos assim a análise dos pontos de ineficiência do mecanismo de paralelismo empregado na implementação de Haskell$_{\#}$, buscando formas de otimizá-lo e justificar os resultados obtidos. Para isso, empregamos a ferramenta de *profiling* que acompanha o compilador GHC[197]. Sua utilização nos permitiu inferir os custos de computação e gerenciamento de paralelismo em IS e CG. Os principais custos identificados nas versões paralelas destes programas são enumerados a seguir:

i) **Custo de computação efetiva:** calculado segundo a porcentagem de tempo despendida por cada processo paralelo na avaliação da função *main* do módulo funcional associado a este;

ii) **Custo de avaliação de funções de ligação:** inerente a abordagem de paralelização, sob responsabilidade do programador e dependente de características inerentes às funcionalidades fornecidas pelo compilador GHC;

iii) **Custo de cópia de valores Haskell em *buffers* MPI:** o uso de tipos de dados que armazenam grandes coleções de elementos de dados, como *arrays*, extensivamente empregadas em IS e CG, podem resultar em um grande custo de cópia para *buffers* contíguos, o que é necessário para uso das primitivas de envio e recepção de dados por meio da biblioteca MPI;

iv) **Custo de sincronização e comunicação:** dependente das características físicas da rede de comunicação e características de sincronização inerentes à abordagem de paralelização do programa em questão;

v) **Custo de gerenciamento de memória:** sobrecarga inerente ao coletor de lixo generacional implementado em GHC[59];

Vale ressaltar que somente o custo associado ao empacotamento e desempacotamento de valores Haskell para *buffers* contíguos (item *iii*) é inerente a implementação

		comput.	f.lig.	buf.	sync	coletor
IS-1	SEQ	45,9%	-	-	-	54,1%
	2	35,4%	3,0%	7,4%	4,6%	45,3%
	4	37,6%	3,0%	7,2%	11,0%	35,7%
	8	36,0%	2,7%	7,2%	20,8%	28,7%
IS-2	SEQ	34,5%	-	-	-	65,5%
	2	-	-	-	-	-
	4	35,3%	2,8%	6,8%	11,7%	38,9%
	8	32,8%	2,7%	7,0%	21,1%	32,6%
CG-1	SEQ	90,2%	-	-	-	9,8%
	2	79,1%	1,5%	2,1%	4,7%	7,7%
	4	70,9%	1,8%	3,2%	11,6%	5,8%
	8	57,5%	3,5%	5,6%	24,0%	2,7%
CG-2	SEQ	84,5%	-	-	-	15,5%
	2	68,5%	1,2%	1,7%	10,8%	11,6%
	4	70,2%	1,5%	2,5%	12,9%	6,3%
	8	61,1%	3,2%	5,1%	19,5%	4,5%

LEGENDA	
comput.	Tempo efetivo de computação
f.lig.	Avaliação de funções de validação
buf.	Cópia de valores Haskell em *buffers* contíguos (*marshalling*)
sync	Comunicação e Sincronização
coletor	Gerenciamento Dinâmico de Memória

Tabela 6.4. Análise de Custos para IS e CG

de Haskell$_{\#}$. Os demais são dependentes de características do compilador GHC, rede de comunicação ou da biblioteca MPI empregada. Na Tabela 6.4 são apresentados os dados de *profiling* obtidos para as instâncias de IS e CG consideradas. É importante ressaltar que ao compilar-se um programa Haskell com o compilador GHC com a opção de *profiling* habilitada, algumas de suas características dinâmicas são alteradas, em especial relativas ao comportamento espacial, o que tem impacto significativo sobre o comportamento do gerenciador de memória.

Nos dados apresentados da Tabela 6.4, ressalta-se a significativa maior sobrecarga do gerenciador de memória na execução de IS em comparação à CG, uma vez que ordenação de inteiros pelo método *bucketsort* requer grande movimentação em memória. Ao contrário, o kernel CG caracteriza-se pela realização de grande quantidade de computações numéricas sobre *arrays* de grandes dimensões.

Ainda em relação a influência do gerenciador de memória sobre as medições, um fato merece atenção: em todos os casos considerados, observa-se que a eficiência do coletor de lixo de GHC cresce a medida que o número de processadores usados na solução paralela é incrementado, com a consequente diminuição da quantidade de memória (*heap*) necessária para execução de cada processo. Como consequência, o *speedup* obtido considerando-se somente o tempo de execução do *mutador* é menor em comparação ao *speedup* total, somando-se os custos do mutador e do coletor. Esse fato pode ser observado nas Figuras 6.11 e 6.12, descritas anteriormente. Na Tabela 6.3.4 são quantificadas as medições de

		eficiência coleta	tempo de coleta	speedup coletor	speedup mutador
IS-1	SEQ	38,1%	112,3	-	-
	2	36,1%	72,8	1,5	1,4
	4	29,3%	28,1	4,0	2,7
	8	25,9%	15,1	7,4	4,2
IS-2	SEQ	41,4%	247,7	-	-
	2	38,9%	168,0	1,4	1,3
	4	34,4%	70,5	3,5	2,6
	8	27,5%	31,9	7,7	4,1
CG-1	SEQ	5,5%	143,1	-	-
	2	5,7%	72,2	1,9	2,0
	4	4,9%	39,8	3,6	3,1
	8	0,8%	4,7	30,4	4,4
CG-2	SEQ	9,0%	350,5	-	-
	2	7,3%	152,0	2,3	1,8
	4	5,1%	61,1	5,7	3,1
	8	2,2%	16,2	21,5	5,0

Tabela 6.5. Análise de Eficiência do Gerenciamento Dinâmico de Memória dos Processos

eficiência do gerenciador de memória para cada caso considerado, quando o programa é compilado sem a opção de *profiling*. As características de escalabilidade do sistema de gerenciamento dinâmico de memória de GHC em relação ao tamanho de problema tratado por um programa Haskell, justifica o uso de paralelismo como forma de aumentar a eficiência de programas científicos desenvolvidos nesta linguagem, uma vez que tais programas em geral fazem uso extenso de espaço de memória, sendo bastante sensíveis ao algoritmo de gerenciamento dinâmico de memória empregado. Além disso o coletor tende a otimizar o uso da memória *cache*, minimizando o número de *cache misses*[206]. Em certas aplicações, tal característica tende a melhorar sensivelmente seu desempenho.

O compilador GHC permite a sintonização de parâmetros que afetam a eficiência do gerenciador de memória em tempo de execução, em especial na configuração do tamanho mínimo da *heap*. Vale portanto ressaltar que, de forma empírica, escolhemos para cada instância de problema e versão paralela (2, 4 e 8) o tamanho de *heap* aproximadamente ótimo.

Observa-se, ainda na Tabela 6.4, que a principal sobrecarga no gerenciamento do paralelismo sob responsabilidade dos mecanismos usados na implementação de Haskell$_{\#}$ corresponde ao tempo gasto com a tradução de *arrays* Haskell (imutáveis e *unboxed*) para *buffers* C, de forma que estas possam ser transmitidas por meio de MPI (*marshalling*). Esta observação é válida para qualquer programa Haskell$_{\#}$ que necessite transmitir grandes *matrizes* ou *vetores* de dados entre processos. Na implementação atual do compilador, os elementos de uma *array* imutável a ser transmitida/recebida devem ser copiados/lidos elemento a elemento em um *buffer* MPI, exigindo tempo linear para realização desta tarefa, mesmo que esta trate-se de uma *array unboxed*[191]. Contudo, tal limitação de desempenho não deve ser encarada como um atributo de Haskell$_{\#}$, mas como uma característica inerente a implementação apresentada, a qual pode ser otimizada uti-

		(i)	(i) (ii)	(i) (ii) (iii)	(i) (ii) (iii) (iv)	(i) (ii) (iii) (iv) (v)
IS-1	2	2,1	1,9	1,6	1,5	1,2
	4	4,1	3,8	3,2	2,6	2,5
	8	7,5	7,0	5,9	4,0	4,4
IS-2	2	1,9	1,8	1,5	-	-
	4	4,1	3,8	3,2	2,5	2,5
	8	8,0	7,3	6,1	4,1	4,5
CG-1	2	2,0	2,0	1,9	1,8	1,9
	4	3,9	3,8	3,6	3,2	3,2
	8	7,9	7,4	6,7	4,9	5,3
CG-2	2	2,1	2,0	2,0	1,7	1,8
	4	4,0	3,9	3,7	3,2	3,4
	8	8,0	7,6	7,0	5,5	6,0

Tabela 6.6. Análise de *Speedup*

lizando técnicas de mais baixo nível para tradução de *arrays unboxed* do "*mundo Haskell*" para o "*mundo MPI*". Analisando sua implementação, concluiu-se que GHC já poderia oferecer suporte nativo para *marshalling* de arrays *unboxed* em tempo constante, sem cópia, o que foi sugerido aos seus desenvolvedores. Independente disso, é facilmente possível alterar-se o código de GHC para suprir esta necessidade de Haskell$_{\#}$, o que não foi porém utilizado neste trabalho, uma vez que a não alteração do compilador sequencial Haskell faz parte das premissas que orientaram o desenvolvimento de Haskell$_{\#}$. Uma vez que o processamento de grandes *arrays* é empregado em virtualmente qualquer programa científico de alto desempenho, essa abordagem parece ser bastante útil para melhorar o desempenho de programas Haskell$_{\#}$, tornando praticamente nulo o gerenciamento de paralelismo sob responsabilidade do sistema em execução de Haskell$_{\#}$, o que é previsto em suas premissas.

Usando as informações de *profiling*, na Tabela 6.3.4 é apresentada a evolução do *speedup* de acordo com os custos envolvidos na execução paralela. Obtêm-se assim uma melhor percepção com respeito ao impacto de cada item de custo sobre o *speedup* final do programa, apresentado na coluna mais à direita. Note que *speedup* linear é obtido em todos os casos quando é considerados somente o custo de avaliação da função *main* do módulo funcional, a qual descreve a computação efetivamente útil realizada pelo programa (i). Ao considerar-se o tempo de avaliação de funções de ligação, o *speedup* degrada-se ligeiramente (ii). Observe que o custo de empacotamento/desempacotamento de *arrays* Haskell em *buffers* contíguos causa uma degradação sobre o *speedup* que não pode ser desprezada (iii), sendo somente superada pela degradação causada pela sobrecarga resultante da sincronização e comunicação entre os processos (iv). Isso se deve a latência de comunicação da configuração de *cluster* adotada no experimento em questão. Como discutido anteriormente, atesta-se sensível melhora no *speedup* quando o tempo de geren-

		Ponto-a-ponto				Coletivo			
		tempo	% coletor	mutador	**comun.**	tempo	% coletor	mutador	**comun.**
IS-1	2	212,6	36,6%	134,8	**17,8**	201,8	35,8%	129,5	**6,4**
	4	100,7	28,0%	71,9	**17,8**	96,2	33,3%	64,1	**11,2**
	8	60,8	24,5%	45,8	**18,0**	58,4	25,6%	43,4	**13,9**
IS-2	2	454,4	38,1%	280,9	**42,2**	432,1	39,0%	263,3	**17,1**
	4	215,8	33,4%	143,6	**35,0**	205,0	35,4%	132,4	**20,3**
	8	118,7	27,6%	85,9	**32,3**	116,2	27,6%	84,0	**26,0**
CG-1	2	1.286,0	5,6%	1.214,7	**42,4**	1278.7	5,7%	1.206,5	**41,8**
	4	811,9	4,9%	772,3	**142,9**	811,9	4,9%	772,3	**141,0**
	8	578.3	0,8%	573.6	**233,5**	565,1	0,8%	560,4	**221,4**
CG-2	2	2.030,4	7,5%	1.878,2	**163,4**	2.096,8	7,3%	1.944,8	**171,2**
	4	1.189,9	5,1%	1.128,9	**241,8**	1.205,1	5,1%	1.144,0	**196,7**
	8	737,3	2,3%	720.4	**210,6**	725.2	2,2%	708.9	**198,7**

Tabela 6.7. Custo de Comunicação nas Versões Ponto-a-Ponto e Coletiva (em segundos)

ciamento de memória é levado em consideração, inclusive sobrepondo o item de custo (iii). Pode-se então admitir que a sobrecarga de gerenciamento de paralelismo inerentes a Haskell$_\#$ é nula, como previsto em suas premissas.

6.3.4.1 Avaliando o Desempenho de Esqueletos MPI Os dados utilizados na avaliação anterior foram obtidos com versões dos programas que fazem uso de esqueletos MPI para descrever padrões coletivos de comunicação. Adicionalmente, com a finalidade de avaliar o impacto do uso de esqueletos MPI nos programas Haskell$_\#$ considerados nesta avaliação, implementamos versões destes mesmos programas sem o uso desta funcionalidade. Estas versões empregam primitivas ponto-a-ponto de comunicação para efetivação das operações coletivas. Uma vez que os *kernels* IS e CG são aqueles que fazem uso efetivo de comunicação coletiva, apresentamos os dados obtidos nas medições para as duas versões destes programas na Tabela 6.7. Vale ressaltar que a principal motivação do uso de primitivas de comunicação coletiva em programas MPI é possibilitar melhor desempenho quando muitos processos realizam padrões coletivos de comunicação.

Usando a ferramenta de *profiling* de GHC, é possível avaliar apuradamente a sobrecarga do paralelismo nas versões de IS e CG (com e sem o uso de esqueletos). Os resultados encontram-se apresentados na Tabela 6.7, para ambas as instâncias de tamanho de problema dos dois programas. Observe a redução do custo de comunicação e sincronização nas versões coletivas, especialmente no kernel IS. Isso se deve ao fato de que as comunicações coletivas empregadas em IS correspondem a transmissão e redução de grandes *arrays*, utilizando a primitiva `MPI_Allreduce`, ao passo que em CG, as comunicações coletivas referem-se a troca de valores únicos do tipo `Double`, embora realizada com uma maior frequência. As demais comunicações da versão coletiva de CG empregam comunicação ponto-a-ponto, assim como em sua versão original, uma vez que MPI não possui primitivas para a transposição de matrizes dispostas em processos organizados em *grids*, requisito do esqueleto Transpose. Entretanto, caso existisse, poderia ser empregada diretamente como forma a melhorar o desempenho de CG. Essa independência de imple-

mentação constitui o cerne da motivação do uso de esqueletos, sendo bastante evidente dentro do modelo #. Vale ainda ressaltar que, na versão coletiva de IS, não é necessária a avaliação da função de ligação *sum_arrays*, uma vez que `MPI_Allreduce` realiza esta operação automaticamente, passando-se para esta o argumento `MPI_SUM`.

CAPÍTULO 7

CONCLUSÕES E PROPOSTAS DE FUTUROS TRABALHOS

A presente tese de doutorado apresentou o modelo # de programação paralela, concebido com vistas à engenharia de programas paralelos de larga escala, problema que ganhou maior amplitude devido ao contexto atual da computação de alto desempenho advindo do surgimento e disseminação das tecnologias emergentes de *cluster computing*[20, 35] e *grid computing*[89]. Neste contexto, atenção especial dispensa-se à adequação às modernas técnicas e metodologias de engenharia de programas de larga escala e à análise de propriedades de programas com uso de formalismos baseados em redes de Petri.

Foi discutida a implementação de um compilador para uma nova versão da linguagem Haskell$_{\#}$, aderindo ao modelo # porém ainda restringindo-se o uso de Haskell para descrição de módulos funcionais (componentes simples em nível de computação). Haskell permite a ortogonalização transparente entre os meios de coordenação e computação da linguagem, implementados pelos conceitos abstratos de *mundo dos processos* e *mundo das computações*, respectivamente. Para isso, é fundamental características inerentes a essa classe de linguagens, como avaliação procastrinada (*lazy*) e funções de alta ordem. Haskell complementa ainda o arcabouço de prova de propriedades de programas do ambiente de desenvolvimento, extendendo-o ao meio de computação, devido a existência de ferramental matemático adequado a análise formal de programas escritos em linguagem funcionais puras e não-estritas. Implementações Haskell$_{\#}$ de um sub-conjunto representativo do *NAS Parallel Benchmarks* [16] foram empregados para avaliação de desempenho desta implementação. Embora sendo ainda um compilador que gera código de alto nível, a implementação mostrou bom potencial de desempenho para as aplicações em questão.

Foi introduzido o protótipo de um ambiente visual voltado a programação de aplicações paralelas baseado no modelo #, denominado VHT (*Visual # Tool*). Dentre as funcionalidades providas por esta ferramenta, são contemplados aspectos relativos a programação # discutidos nesta tese, incluindo-se a integração a ferramentas de análise de propriedades e a avaliação de desempenho baseadas em redes de Petri, como INA [194], PEP [36] e TimeNET [234].

O ambiente VHT tem sido desenvolvido de forma incremental. Atualmente, dispõe-se de um protótipo onde estão incluídas a programação visual das abstrações do modelo # para composição de topologias, interface com XML, geração de código #, geração de código PNML a partir do código # visando interface com a ferramenta PEP e INA para análise de propriedades de programas.

Outros aspectos devem ser considerados em breve. Dentre estes destaca-se a geração de código SPNL para avaliação de desempenho de programas usando redes de Petri estocásticas, por meio da ferramenta TimeNET. Porém, isso depende ainda do projeto da

ferramenta de avaliação e predição de desempenho de programas # integrada à VHT, aspecto que não foi desenvolvido nesta tese. O objetivo é prover a capacidade de avaliação de desempenho utilizando simuladores de redes, como NS (*Network Simulator*), e ferramentas para análise de redes de Petri estocásticas, como o TimeNET. Adicionalmente, o suporte a especificação visual do comportamento de processos (combinadores baseados em OCCAM), o qual atualmente é realizado textualmente, e abstrações visuais para o suporte a notação indexada e parametrizada suportada pela linguagem # devem ainda ser concretizadas em VHT. Uma versão final de VHT, voltada a eventuais usuários deste ambiente, deve ser disponibilizada em meados do ano de 2004, com funcionalidades básicas incluídas. Versões sub-seqüentes deverão incluir novas funcionalidades e evoluções que eventualmente deverão surgir.

As contribuições da presente tese moldam as fundações a partir de onde dar-se-á continuidade a pesquisa relacionada ao modelo #, visando um objetivo maior: o desenvolvimento de um ambiente integrado para o desenvolvimento de programas paralelos de larga escala, baseado no modelo #, e seu emprego efetivo no desenvolvimento de aplicações reais. O protótipo de VHT, apresentado neste trabalho, é o ponto de partida para alcançar este objetivo. A seções que se seguem enumeram trabalhos de pesquisa futuros que conduzirão ao concretização desse ambiente.

7.1 ALOCAÇÃO DE PROCESSOS A PROCESSADORES

Nesta tese, optou-se por omitirem-se considerações a respeito da definição de estratégias de alocação de processos # aos processadores da arquitetura distribuída alvo, propondo-se tal como um trabalho futuro a ser desenvolvido. Contribui para este fato a complexidade associada a este problema, a qual exige o estudo, concepção e avaliação de várias alternativas. Espera-se a definição de uma abordagem suficientemente flexível para contemplar a alocação de processos # sobre *clusters* ou *grids*, arquiteturas de características completamente distintas, com intervenção direta e explícita do programador, obedecendo as premissas originais que deram origem ao modelo. Ferramentas que auxiliem ao programador nesta tarefa são bem vindas e também deverão ser propostas.

Uma possível abordagem que vem sendo levada em consideração, porém ainda não avaliada diretamente, diz respeito ao uso de esqueletos topológicos para descrição das propriedades arquiteturais da máquina paralela alvo. Assim, a alocação dos processos em um programa # poderia ser tratada considerando-se tal como um aspecto (Seção 4.3), sendo a tarefa de alocação resultado do mapeamento da topologia de aplicação do programa ao aspecto correspondente às características da arquitetura.

7.2 UMA VERSÃO MULTILINGUAL PARA O MODELO

A versão atualmente implementada do modelo # é a linguagem Haskell$_{\#}$, implementada no topo de MPI, a qual utiliza Haskell para descrição de computações sequenciais. Porém, deseja-se oferecer o suporte a possibilidade de que outras linguagens sejam usadas para descrição das computação realizadas por um programa #. Esta constitui uma possibilidade real sob o modelo #, tendo em vista que bibliotecas de passagem de mensagens

estão disponíveis para várias linguagens, servindo como "cola" para unir módulos computacionais compilados separadamente por meio de geradores de código apropriados às linguagens com os quais encontram-se implementados. Haskell é vista como a linguagem ideal à prototipação de módulos funcionais, em uma fase anterior ao desenvolvimento de seu código propriamente dito, utilizando uma linguagem apropriada à implementação de sua funcionalidade. Por exemplo, módulos funcionais que implementam computações intensivamente numéricas devem ser implementados em FORTRAN, enquanto módulos funcionais de manipulação simbólica, os quais exigem eficiente tratamento do fluxo de controle devem ser implementados em C. Módulos estruturalmente complexos podem ser implementados em linguagens de mais alto nível, possivelmente orientadas a objetos, como JAVA. Esta característica multi-lingual é um ponto crucial ao suporte de implementação de programas de larga escala sobre a infra-estrutura de *grids*.

Entretanto, surge a questão de como manter a ortogonalidade transparente entre os meios de coordenação e computação do modelo # na ausência de avaliação *lazy* e funções de alta ordem, características inerentes às linguagens funcionais. Deve ser investigado para este propósitos o uso de aspectos #, embora outras alternativas devam ser exploradas.

7.3 IMPLEMENTAÇÃO DE INTERFACES COM BIBLIOTECAS CIENTÍFICAS

Bibliotecas de rotinas voltadas a computação científica, paralelizadas ou não, têm sido vastamente usadas durante décadas por cientistas de todas as áreas para implementação de soluções para problemas de seu interesse. Não é possível ignorar décadas de desenvolvimento, validação e aperfeiçoamento dos códigos que implementam essas bibliotecas. Em geral, bibliotecas científicas implementam o estado da arte das técnicas de solução de problemas comuns em computação científica, sendo portanto voltadas a fins específicos.

Funcionalidades de bibliotecas científicas podem ser oferecidas pelo ambiente # como componentes encapsulados. No caso de bibliotecas paralelizadas, a noção de paralelismo hierárquico suportada pelo modelo # é de grande valia, permitindo inclusive que funções que executam de maneira mais eficiente sobre multi-processadores sejam alocadas aos nós multi-processados em arquiteturas de constelações. Esqueletos parciais podem ser usados para configurar a funcionalidade de uma rotina, caso necessário, e otimizar seu desempenho, ao modo dos esqueletos MPI apresentados nesta tese. O uso de bibliotecas científicas é uma questão que diz respeito a aspectos puramente de implementação, tendo em vista que o arcabouço para seu suporte encontra-se já inerentemente suportado pelo modelo #.

7.4 FERRAMENTA PARA ANÁLISE, SIMULAÇÃO E AVALIAÇÃO DE DESEMPENHO DE PROGRAMAS

O capítulo 5.5 demonstrou como ferramentas pré-existentes podem ser usadas na análise de propriedades formais e avaliação de desempenho de programas #, assumindo-se a existência de uma tradução destes para redes de Petri, como aquela apresentada no Capítulo 5. O mecanismo empregado atualmente consiste na manipulação direta, pelo

programador, da rede de Petri gerada para o programa #, utilizando uma ferramenta de verificação de propriedades de redes de Petri apropriada, como PEP e INA. Entretanto, o baixo nível de abstração oferecido por redes de Petri lugar/transição, usadas para descrever as estruturas de alto nível de programas #, resulta na geração de redes de Petri com um número de componentes (lugares e transições) virtualmente intratável por um especialista humano, mesmo considerando programas # de complexidade mediana. Seriam assim exigidos programadores bastante habilidosos em lidar manualmente com estruturas complexas, suposição nem sempre realista. Certas propriedades que podem ser verificadas por ferramentas automáticas são comuns a quaisquer programas #, ou classes facilmente caracterizáveis destes. Outras são irrelevantes para o processo de análise. Estudando-se a relevância das propriedades e tipos de análise que podem ser conduzidas com ferramentas de suporte a análise de redes de Petri, pode-se projetar uma ferramenta orientada a análise de propriedades de programas #, porém em um nível de abstração mais próximo de sua realidade.

Assim, propõe-se o projeto de uma ferramenta de alto nível para facilitar a análise de propriedades e avaliação de desempenho de programas #, implementada no topo das ferramentas de verificação usadas no Capítulo 5.5, porém considerando somente aqueles aspectos considerados essenciais no tratamento formal de programas #. Ferramentas como PEP, INA e TimeNET oferecem o suporte necessário a integração de outras ferramentas de mais alto nível, o que viabiliza o alcance deste objetivo.

Uma outra possibilidade a ser considerada é o uso de redes de Petri de alto nível, como redes de Petri coloridas [127].

7.5 IMPLEMENTAÇÕES EFICIENTES

Por ser um protótipo, com o objetivo de demonstrar como Haskell$_{\#}$ (extensivamente o modelo #) pode ser implementado de forma relativamente eficiente com o uso de ferramentas pré-existentes, o compilador # gera código de nível elevado de abstração, com chamadas à funções de alta ordem, implementadas em Haskell, as quais implementam operações de comunicação primitivas de forma mais abstrata. Embora mesmo com tais características os resultados obtidos na avaliação de desempenho da implementação proposta tenham sido bastante satisfatórios (Capítulo 6), vislumbra-se a implementação de um compilador capaz de gerar código de mais baixo nível, com chamadas diretas às rotinas da biblioteca de passagem de mensagens sobre a qual encontra-se implementada, como MPI. Vários tipos de otimizações no código gerado poderão ser implementadas, principalmente pela análise dos esqueletos empregados na constituição da topologia de processos do programa.

7.6 CONSIDERAÇÕES FINAIS

As seções anteriores evidenciam que o modelo #, produto essencial resultante do trabalho de pesquisa que originou esta tese, constitui-se o ponto de partida para outros tantos trabalhos, que visam o aprimoramento, implementação eficiente e utilização do modelo proposto nesta tese. A perspectiva futura é consolidar o modelo # como uma ferramenta

alternativa aos mecanismos de passagem de mensagens tradicionais, disseminando seu uso no meio da comunidade interessada em computação científica de alto desempenho. Para isso, o modelo deve ser efetivamente materializado, o que deve ser conseguido em breve com a consolidação de uma força tarefa voltada a implementação deste ambiente a partir dos protótipos construídos para esta tese. Os resultados obtidos neste trabalho estabelecem assim as direções que guiarão os futuros desenvolvimentos do ambiente # de programação para que esse objetivo possa ser alcançado nos próximos anos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] T. Agerwala, J. Martin, J. Mirza, D. Sadler, D. Dias, and M. Snir. SP2 System Architecture. *IBM System Journal*, 31(2), 1995.

[2] G. Akerholt, K. Hammond, S. L. Peyton Jones, and P. Trinder. Processing Transactions on Grip, a Parallel Graph Reducer. *PARLE'93*, June 1993.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[4] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team. A Case for Networks of Workstations. *IEEE Micro*, 15(1):54–64, feb 1995.

[5] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. *New Generation Computing*, 9(3–4):445–473, May 1991.

[6] G. Andrews. *Concurrent Programming: Principles and Practice.* Addison Wesley, 1991.

[7] T. Araki, T. Kagimasa, and N. Tokura. Relations of Flow Languages to Petri Net Languages. *Theoretical Computer Science*, 15:51–75, 1981.

[8] F. Arbab. Coordination of Massively Concurrent Activities. Technical Report CS-R9565, Centrum voor Wiskunde en Informatica (CWI), 30 1995.

[9] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In P. Ciancarini and C. Hankin, editor, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061, pages 34–56. Springer-Verlag, Berlin, 1996.

[10] F. Arbab, P. Ciancarini, and C. Hankin. Coordination Languages for Parallel Programming. *Parallel Computing*, 24(7):989–1004, 1998.

[11] W. Aspray. John Von Neumman Constributions to Computing and Computer Science. *Annals of History of Computing*, 11(3):189–195, 1989.

[12] C. Assmann. A Coordination Language for Systems of Cooperating Processes. In *1995 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 162–183, June 1995.

[13] C. Assmann. Coordinating Functional Processes Using Petri Nets. *Implementation of Functional Languages, LNCS 1268*, pages 162–183, September 1997.

[14] L. Augustsson. HBC Haskell Compiler, Web Page, http://www.cs.chalmers.se/~augustss/hbc/hbc.html, Chalmers University of Technology, Department of Computing Science.

[15] J. Backus. Can Programming be Liberated from the von Newman Style ? A Functional Style and its Algebra of Programming. *Communication of ACM*, 21(8):613–641, 1978.

[16] D. H. Bailey and et al. The NAS Parallel Benchmarks. *International Journal of Supercomputing Applications*, 5(4):63–73, 1994.

[17] P. Bailey. Algorithms Skeletons in paraML. *TRACS Research Report, Edinburgh Parallel Computing Centre*, 1994.

[18] M. Baker, R. Buyya, , and D. Hyde. Cluster Computing: A High Performance Contender. *IEEE Computer*, pages 79–83, July 1999.

[19] M. Baker, R. Buyya, and D. Hyde. Cluster Computing: A High Performance Contender. *Communications of the ACM*, 42(7):79–83, July 1999.

[20] M. (editor) Baker. Cluster Computing White Paper. Technical report, Task Force on Task Force on Cluster Computing, April 2000. www.clustercomputing.org.

[21] J.-P. Banâtre and D. Le Mètayer. Gamma and the Chemical Reaction Model: Ten Years After. In J.-M. Andreoli and H. Gallaire and D. Le Mètayer, editor, *Coordination programming: mechanisms, models and semantics*, pages 1–39, 1996.

[22] U. Banerjee. *Loop Parallelization*. Kluwer Print on Demand, May 1994.

[23] M. Banville. Sonia: An Adaptation of Linda for Coordination of Activities in Organizations. In *The First International Conference on Coordination Models, Languages and Applications (COORDINATION'96)*, pages 57–74, April 1996.

[24] M. R. Barbacci, D. L. Doubleday, C. B. Weinstock, and R. W. Lichotta. Durra: An Integrated Approach to Software Specification, Modelling, and Rapid Prototyping. Technical Report 91-TR-21, Software Engineering Institute, Carnegie Mellow University, September 1991.

[25] M. R. Barbacci, D. L. Doubleday, C. B. Weinstock, and R. W. Lichotta. Durra: A Structure Description Language for Developing Distributed Applications. *IEE Software Engineering Journal*, 8(2):83–94, March 1993.

[26] E. Barscz, R. Fatoohi, V. Venkatakrishnan, and S. Weeratunga. Solution of Regular, Sparse Triangular Systems on Vector and Distributed-Memory Multiprocessors. Technical Report NAS RNR-93-007, NASA Ames Research Center, April 1993.

[27] D. H. Bayley, T. Harris, W. Shapir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995. http://www.nas.nasa.org/NAS/NPB.

[28] S. Baylor and C. Wu. *I/O in Parallel and Distributed Computer Systems*, chapter 7. Kluwer Academic, 1996.

[29] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorban, U. A. Ranawak, and C. V. Packer. Bewoulf: A Parallel Workstation for Scientific Computation. In *1995 International Conference on Parallel Processing*, 1995.

[30] M. Becker and et al. The PowerPC 601 Microprocessor. *IEEE Micro*, October 1993.

[31] G. Bell and J. Gray. What's the Next in High Performance Computing. *Communications of the ACM*, 45(2):91–95, 2002.

[32] L. Bellissard, S. B. Atallah, F. Boyer, and M Riveill. Distributed Application Configuration. In *International Conference on Distributed Computing Systems*, pages 579–585, 1996. Also as a techincal report at INRIA, RR-3119.

[33] J. A. Bergstra and P. Klint. The ToolBus - a Component Interconnection Architecture. Technical Report P9408, , Programming Research Group, University of Amsterdam, 1995.

[34] K. J. Berkling. Reduction Languages for Reduction Machines. *2nd. Annual ACM Symposium on Computer Architecture*, pages 133–140, 1978.

[35] M. Bertozzi, G. Chiola, G. Ciaccio, G. Conte, P. Marenzoni, A. Poggi, and P. Rossi. DISCO Report on the State-of-the-Art of PC Cluster Computing. Technical Report DISI-TR-98-09, DISI, Universitá de Genova, December 1998.

[36] E. Best, J. Esparza, B. Grahlmann, S. Melzer, S. Römer, and F. Wallner. The PEP Verification System. In *Workshop on Formal Design of Safety Critical Embedded Systems (FEmSys'97)*, 1997.

[37] D. Bistry. *The Complete Guide to MMX Technology*. Mc-Graw Hill/TAB Eletronics, April 1997.

[38] M. B. Blackmon, B. Boville, F. Bryan, R. Dickinson, P. Gent, J. Kiehl, R. Moritz, D. Randall, J. Shukla, S. Solomon, G. Bonan, S. Doney, I. Fung, J. Hack, E. Hunke, and J. et al Hurrell. The Community Climate System Model. *BAMS*, 82(11):2357–2376, 2001.

[39] M. Blume and A. W. Appel. Hierarchical Modularity. *ACM Transactions on Programming Languages and Systems*, 21:813–847, 1999.

[40] P. Bouvry and F. Arbab. A Coordination Language for Systems of Cooperating Processes. In *The First International Conference on Coordination Models, Languages and Applications (COORDINATION'96)*, April 1996. Short contribution for poster session.

[41] S. Breitinger, R. Lógen, Y. Ortega Mallén, and R. Peña. Eden - The Paradise of Functional Concurrent Programming. *Lecture Notes in Computer Science*, 1123:710–713, 1996. Second International Euro-Par Conference.

[42] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña. High-level Parallel and Concurrent Programming in Eden. In *Proceedings of APPIA-GULP-PRODE Joint Conference on Declarative Programming*, pages 213–224, June 1997.

[43] A. Brüll and H. Kuchen. TPascal - A Language For Task Parallel Programming. In Springer-Verlag, editor, *Proceedings of Europar'96, LNCS 1123, pages 646-654*, 1996.

[44] W. H. Burge. Recursive Programming Techniques. *Addison-Wesley Publishers Ltd.*, 1975.

[45] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[46] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 66–76. Springer-Verlag, jul 1992.

[47] F. H. Carvalho Jr. $Haskell_{\#}$: Uma Extensão Paralela para Haskell. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, January 2000.

[48] F. H. Carvalho Jr. Topological Skeletons in $Haskell_{\#}$. In *International Parallel and Distributed Symposium (IPDPS 2003)*. IEEE Press, April 2003. *(april 2003)*.

[49] F. H. Carvalho Jr., R. D. Lins, and R. M. F. Lima. Parallelizing MCP-$Haskell_{\#}$ for Evaluating $Haskell_{\#}$ Parallel Programming Environment. In UnB, editor, *Proceedings of the 13th Brazilian Symposium on Computer Architecture and High-Performance Computing*, September 2001.

[50] F.H. Carvalho Jr., R.M.F. Lima, and R.D. Lins. Coordinating Functional Processes with $Haskell_{\#}$. In ACM Press, editor, *ACM Symposium on Applied Computing, Special Tracking on Coordination Languages, Models and Applications*, pages 393–400, March 2002.

[51] F.H. Carvalho Jr. and R.D. Lins. $Haskell_{\#}$: Parallel Programming Made Simple and Efficient. *Journal of Universal Computer Science*, 9(8):776–794, August 2003. http://www.jucs.org/jucs_9_8/haskell_parallel_programming_made.

[52] F.H. Carvalho Jr., R.D. Lins, and R.M.F. Lima. Translating $Haskell_{\#}$ Programs into Petri Nets. In *Lecture Notes in Computer Science (VECPAR'2002)*, volume 2565, pages 635–649. Springer Verlag, 2003.

[53] S. Castellani. Enhancing Coordination and Modularity Mechanisms for a Language with Objects-as-Multisets. In *The First International Conference on Coordination Models, Languages and Applications (COORDINATION'96)*, pages 89–106, April 1996.

[54] Center for Petroleum and Geosystems Engineering, University of Texas at Austin. Parallel Simulation of Petroleum Reservoirs on High-Performance Clusters. *Dell Power Solutions*, 1:103–111, 2001.

[55] M. (editor) et all Chakravarty. *The Haskell 98 Foreign Function Interface (FFI) 1.0 (An Addendum to Haskell 98 Report)*. Glasgow University, Departament of Computing, Functional Programming Research Group, 2002.

[56] T. R. Chandrupatla and A. D. Belegundu. *Introduction to Finite Elements in Engineering*. Prentice Hall, January 2002. 3rd Edition.

[57] K. M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. *Research Directions in Concurrent Object-Oriented Programming*, 1993.

[58] B. Chapman, M. Haines, P. Mehrotra, H. Zima, and John Van Rosendale. Opus: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6(2), 1997.

[59] A. Cheadle, T. Field, S. Marlow, S. Peyton Jones, and L. While. Non-Stop Haskell. In *International Conference on Functional Programming (ICFP)*, 2000.

[60] A. Church. A Set of Postulates for the Foundation of Logic. *Annals of Math*, 33(2):346–366, 1932.

[61] P. Ciancarini. Coordination Models, Languages, Architectures and Applications: A Personal Perspective. Technical report, University of Leuven, February 1997.

[62] P. Ciancarini and D. Rossi. Jada: Coordination and Communication for Java Agents. In *Second Internation Workshop on Mobile Object Systems: Towards the Programmable Internet (MOS'96)*, Lecture Notes in Computer Science, pages 213–228. Springer-Verlag, July 1996.

[63] M. Cole. Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. *PhD Thesis, Department of Computer Science, University of Edinburg*, October 1988.

[64] M. Cole. *Algorithm Skeletons: Structured Management of Paralell Computation*. Pitman, 1989.

[65] M. Cole. High Order Functions for Parallel Evaluation. *Glasgow Workshop on Functional Programming Language*, August 1989.

[66] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, may 1993.

[67] M. Danelutto, R.. Di Meglio, S. Orlando, S. Pellagati, and M. Vanneschi. A Methodology for the Development and Support of Massivelly Parallel Programs. Technical report, Dipartimento de Informatica, Universidad di Pisa, 1991.

[68] J. Darlington, A. J. Field, P. G. Harrison, P. H. Kelly, D. W. Sharp, Q. Wu, and R. L. While. Parallel Programming Using Skeleton Functions. *PARLE'93 - Springer-Verlag LNCS 694*, pages 146–160, 1993.

[69] J. Darlington, Y. Guo, H.W. To, and J. Yang. Functional Skeletons for Parallel Coordination. *Lecture Notes in Computer Science*, 966:55–68, 1995.

[70] J. Darlington and M. J. Reeve. ALICE: A Multiple-Processor Reduction Machine for Parallel Evaluation Aplicative Languages. *FPCA'81: Conference on Functional Programming Languages and Computing Architecture*, pages 65–76, 1981.

[71] F. DeRemer and H. H. Kron. Programming-in-the-Large versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, pages 80–86, June 1976.

[72] E. C. Djkstra. The Structure of THE Multiprogramming System. *Communications of the ACM*, 11:341–346, November 1968.

[73] J. Dongarra, S. W. Otto, M. Snir, and D. Walker. An Introduction to the MPI Standard. Technical Report CS-95-274, University of Tennessee, January 1995. http://www.netlib.org/tennessee/ut-cs-95-274.ps.

[74] J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A Message Passing Standard for MPP and Workstation. *Communication of ACM*, 39(7):84–90, 1996.

[75] C. Dornan, I. Jones, and S. Marlow. *Alex User Guide*, 2003. http://www.haskell.org/alex.

[76] J. Du and J. Y.-T. Leung. Complexity of Scheduling Parallel Task Systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.

[77] H. Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, 21(9):5–16, February 1990.

[78] P. Dybjer and H. P. Sander. A Functional Programming Approach to the Specification and Verification of Concurrent Systems. *Formal Aspects of Computing*, 1:303–319, 1989.

[79] K. Ekanadham. A Perspective on Id. *Parallel Functional Languages and Compilers, B. K. Szymanski, Ed. ACM Press, New York*, pages 197–254, 1991.

[80] K. Fall and K. Varadhan. The NS Manual (formely NS Notes and Documentation). Technical report, The VINT Project, A Collaboration between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC, april 2002.

[81] S Finne. *A Haskell Foreign Function Interface*. Glasgow University, Departament of Computing, Functional Programming Research Group, May 2000. Version 0.99.

[82] G. Florijn, T. Bessamusca, and D. Greefhorst. Ariadne and HOPLa: Flexible Coordination of Colaborative Processes. In *The First International Conference on Coordination Models, Languages and Applications (COORDINATION'96)*, pages 197–214, April 1996.

[83] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of IEEE*, 54(12):1901–1909, December 1966.

[84] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computing*, 21(9):948–960, September 1972.

[85] S. Fortune and Wyllie J. Parallelism in Random Access Machines. In *Proceedings of the 10th Ann. ACM Symposium on Theory of Computing*, pages 114–118. ACM Press, 1978.

[86] High Performance Fortran Forum. *High Performance Fortran, Language Specification, Version 2.0*, Jan 1997.

[87] I. Foster. Compositional Parallel Programming Languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1985.

[88] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1994.

[89] I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1 edition, November 1998.

[90] I. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. Technical Report MCS-P327-0992, Argonne National Laboratory, jun 1992.

[91] I. Foster, R. Olson, and S. Tuecke. Productive Parallel Programming: The PCN Approach. *Journal of Scientific Programming*, 1(1):51–66, 1992.

[92] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1989.

[93] S. Frolund and G. Agha. Abstracting Interactions Based on Message Sets. In *Seventh European Conference on Object-Oriented Programming (ECCOP'93)*, volume 707 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag, April 1993.

[94] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant. Distributed Coordination with MESSENGERS. *Science of Computer Programming*, 31(2–3):291–311, July 1998.

[95] L. A. Galán and R. Peña. Functional Skeletons Generate Process Topologies in Eden. In *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP'96*, volume 1140 of *Lecture Notes in Computer Science*, pages 289–303. Springer, September 1996.

[96] M. Garey, D. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Graph Problems. *Theoretical Computer Science*, ??(1):237–267, 1976.

[97] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. *MIT Press, Cambridge*, 1994.

[98] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[99] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[100] D. Gelernter and D. Kaminsky. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. In *Sixth ACM Internation Conference on Supercomputing*, volume 7, pages 417–427. ACM Press, July 1992.

[101] R. German. SPNL: Processes as Language-Oriented Building Blocks of Stochastic Petri Nets. In *Computer Performance Evaluation, Modelling Techniques and Tools, Proc. 9th Conf. St. Malo*, pages 123–134. Springer Verlag, 1997.

[102] GHC Team. The Glasgow Haskell Compiler User's Guide, Version 4.01, 1998.

[103] J. Gischer. Shuffle Languages, Petri Nets, and Context-Sensitive Grammars. *Communications of the ACM*, 24(9):597–605, September 1981.

[104] K. Gödel. On Formally Undecidable Propositions of Principia Mathematica and Related Systems I. In *A Source Book in Mathamatical Logic*, pages 596–616. Harvard University Press, 1967.

[105] J. Gustafson. Reevaluating Amdahls's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[106] J. Gustafson. The Scaled-Sized Model: A Revision of Amdahl's Law. In *Third International Conference on Supercomputing*, May 1988.

[107] M. M. Hamdan. *A Combinational Framework for Paralell Programming Using Skeleton Functions*. PhD thesis, Department of Computing and Electrical Engineering, Hariot-Watt University, Jan 2000.

[108] J. Hammes, O. Lubeck, and W. Böhm. Comparing Id and Haskell in a Monte Carlo Photon Transport Code. *Journal of Functional Programming*, pages 283–316, July 1995.

[109] K. Hammond and G. Michaelson. Research Directions in Parallel Functional Programming. *Springer-Verlag*, 1999.

[110] P. G. Harrison and M. J. Reeve. The Parallel Graph Reduction Machine, Alice. *Workshop on Graph Reduction, Springer-Verlag LNCS 279*, pages 181–202, 1986.

[111] P. H. Hartel and et al. Benchmarking Implementation of Functional Languages with 'Pseudoknot', a Floating-Intensive Benchmark. *Journal of Functional Programming*, 6(4):621–655, July 1996.

[112] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[113] J. Hicks, D. Chiou, B. S. Ang, and V. K. Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributing Computing*, 18:273–300, 1993.

[114] B. Hirsbrunner, M. Aguilar, and O. Krone. CoLa: A Coordination Language for Massive Parallelism. In *Symposium on Principles of Distributed Computing*, page 384, 1994.

[115] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[116] A. A. Holzbacher. A Software environment for Concurrent Coordinated Programming. In *The First International Conference on Coordination Models, Languages and Applications (COORDINATION'96)*, pages 249–266, April 1996.

[117] J. E. Hopcroft and J. D. Ulmman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, April 1979.

[118] P. Hudak. Concept, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[119] P. Hudak. Para-Functional Programming in Haskell. *Parallel Functional Languages and Compilers, B. K. Szymanski, Ed. ACM Press, New York*, pages 159–196, 1991.

[120] P. Hudak. *Para-Functional Programming in Haskell*, chapter 5, pages 159–196. Frontier Series, ACM Press, 1991.

[121] P. Hudak, S. P. L. Jones, and P. L. Wadler. Report on Programming Language Haskell: a Non-Strict, Purely Functional Languages. *Special Issue of SIGPLAN Notices*, 16(5), May 1992.

[122] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[123] K. Hwang. *Advanced Computer Architecture (Parallelism, Scalability, Programability)*. McGrawHill Series in Computer Science, 1993.

[124] Inmos. Occam Programming Manual. *Prentice-Hall, C.A.R. Hoare Series Editor*, 1984.

[125] F. Ino, Fujimoto N., and K. Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 133–142. ACM Press, 2001.

[126] T. Ito and Y. Nishitani. On Universality of Concurrent Expressions with Synchronization Primitives. *Theoretical Computer Science*, 19:105–115, 1982.

[127] K. Jensen. Coloured Petri Nets and Invariant Method. *Theoretical Computer Science*, 14:317–336, 1981.

[128] S. C. Johnson. YACC - Yet Another Compiler Compiler. *Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey*, 1975.

[129] T. Johnsson. Compiling Lazy Functional Languages. *PhD Thesis, Chalmers Tekniska Högskola, Göteborg, Sweden*, 1987.

[130] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Willey & Sons, 1996.

[131] G. R. R. Justo. Ambiente de Programação Distribuída com Configuração Dinâmica de Processos. Master's thesis, Departamento de Informática, Universidade Federal de Pernambuco, 1988.

[132] G. R. R. Justo and P. R. F. Cunha. Programming Distributed Systems with Configuration Languages. In *International Workshop on Configurable Distributed Systems*, pages 118–127. IEEE CS Press, 1992.

[133] G. R. R. Justo and P. R. F. Cunha. Deadlock-free Configuration Programming. In *2nd. Workshop on Configurable Distributed Systems, IEEE CS Press*, pages 147–158, March 1994.

[134] G. R. R. Justo and P. R. F. Cunha. Framework for Developing Extensible and Reusable Parallel and Distributed Applications. In *IEEE International Conference on Algorithms and Architectures for Parallel Processing*, pages 29–36. IEEE CS Press, 1996.

[135] O. Kaser, C.R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Equals - A Fast Parallel Implementation of a Lazy Language. *Journal of Functional Programming*, 7(2):183–217, March 1997.

[136] P. Kelly. Functional Programming for Loosely-coupled Multiprocessors. *Research Monographs in Parallel and Distributed Computing, MIT Press*, 1989.

[137] J. M. Kewley and K. Glynn. Evaluation Annotations for Hope+. In *Glasgow Workshop on Functional Programming*, pages 329–337. Springer-Verlag WICS, 1981.

[138] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44:59–65, 2001.

[139] G. Kiczales, J. Lamping, Menhdhekar A., Maeda C., C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Lecture Notes in Computer Science (Object-Oriented Programming 11th European Conference – ECOOP '97)*, pages 220–242. Springer-Verlag, November 1997.

[140] T. Kielmann. Designing a Coordination Model for Open Systems. In *The First International Conference on Coordination Models, Languages and Applications (COORDINATION'96)*, pages 267–284, April 1996.

[141] S. C. Klenne. Representation of Events in Nerve Nets and Finite Automata. *Annals of Mathematicas Studies*, 34:3–42, 1956.

[142] J. Kramer, J. Magee, and A. Finkelstein. A Constructive Approach to the Design of Distributed Systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pages 580–587, Washington, DC, 1990. IEEE Computer Society.

[143] J. Krammer. Distributed Software Engineering. In IEEE Computer Society Press, editor, *Proc. 16th IEEE International Conference on Software*, 1994.

[144] J. Krammer. Distributed Software Engineering. In *Proceedings of 16th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 1994.

[145] M. E. Lesk. LEX - A Lexical Analyzer Generator. *Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey*, 1975.

[146] R. M. F. Lima. *Haskell$_{\#}$ - Uma Linguagem Funcional Paralela - Ambiente de Programação, Implementação e Otimização.* PhD thesis, Centro de Informáica, UFPE, July 2000.

[147] R. M. F. Lima, F. H. Carvalho Jr., and R. D. Lins. *Haskell$_{\#}$*: A Message Passing Extension to Haskell. *CLAPF'99 - 3rd Latin American Conference on Functional Programming*, pages 93–108, March 1999.

[148] R. M. F. Lima and R. D. Lins. *Haskell$_{\#}$*: A Functional Language with Explicit Parallelism. *LNCS VECPAR'98 - International Meeting on Vector and Parallel Processing*, pages 1–11, June 1998.

[149] R. M. F. Lima and R. D. Lins. Translating HCL Programs into Petri Nets. In *Proceedings of the 14th Brazilian Symposium on Software Engineering*, 2000.

[150] R. D. Lins. Categorical Multi-Combinators. *Functional Programming Languages and Computer Architecture, Springer-Verlag, LNCS 274*, pages 60–79, September 1987.

[151] R. D. Lins. Partial Categorical Multi-Combinators and Church-Rosser Theorems. *University of Kent - Canterbury - Laboratory Report 7/92*, April 1992.

[152] R. D. Lins. Functional programming and parallel processing. *2nd International Conference on Vector and Parallel Processing - VECPAR'96 - LNCS 1215 Springer-Verlag*, pages 429–457, September 1996.

[153] R. D. Lins. The Fall and Rise of Functional Programming. In *3rd Summer School on Functional Programming*, pages 1–40, 1998.

[154] R. D. Lins and B. O. Lira. ΓCMC: A Novel Way of Compiling Functional Languages. *Journal of Programming Languages*, 1:19–40, July 1993.

[155] R. D. Lins and S.J. Thompson. Implementing SASL using Categorical Multi-Combinators. *Software — Practice and Experience*, 20(8):163–166, 1990.

[156] H. W. Loidl, U. Klusik, K. Hammond, R. Loogen, and P. W. Trinder. GpH and Eden: Comparing two Parallel functional Languages on a Beowulf Cluster. In *2nd Scottish Functional Programming Workshop*, 2000.

[157] S. Lucco and O. Sharp. Delirium: An Embedding Coordination Language. In *Proceedings of the 1990 Conference on Supercomputing*, pages 515–524. IEEE Computer Society Press, 1990.

[158] S. Lucco and O. Sharp. Parallel Programming With Coordination Structures. In ACM Press, editor, *Conference Record of the Eighteenth ACM Symposium on the Principles of Programming Languages*, January 1991.

[159] P. R. M. Maciel, R. D. Lins, and P. R. F. Cunha. *Introdução às redes de Petri e Aplicações*. X Escola de Computação, Campinas, SP, June 1996.

[160] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[161] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. In *International Workshop on Configurable Distributed Systems*, London, UK, 1992.

[162] J. Magee, N. Dulay, and J. Kramer. A Constructive Development Environment for Parallel and Distributed Programs. *IEE/IOP/BCS Distributed Systems Engineering*, pages 304–312, September 1994.

[163] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6), 1989.

[164] S. Marlow and A. Gill. *Happy User Guide*, 2001. http://www.haskell.org/happy.

[165] C. C. Mattax and R. L. Kyte. *Reservoir Simulation*, volume 3 of *Monograph Series*. SPE, 1990.

[166] McCarthy, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 3(4):184–195, 1960.

[167] H. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. Top 500 Supercomputer sites, Web Page, http://www.top500.org, University of Mannheim/University of Tennesee/NERSC/LBNL.

[168] E. E. Miller and R. H. Katz. Input/Output Behavior of Supercomputing Applications. In *Proceedings of Conference Supercomputing'91*, pages 567–576, November 1991.

[169] R. Milner. A Calculus of Communitating Systems. *Lecture Notes in Computer Science*, 92, 1980.

[170] R. Milner. Elements of Interaction. *Communications of the ACM*, 36(1):78–89, January 1993.

[171] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. Technical Report ECS-LFCS-89-85 and -86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.

[172] R. Milner, M. Tofte, and R. Haper. The Definition of Standard ML. *MIT Press, Cambridge, Massachussets*, 1989.

[173] N. H. Minsky and J. Leichter. Law-Governed Linda as a Coordination Model. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 125–145. Springer-Verlag, jul 1992.

[174] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[175] C. A. Moritz and M. I. Frank. LogGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):404–415, 2001.

[176] M. A. Musicante and R. D. Lins. GMC: A Graph Multi-Combinator Machine. *Microprocessing and Microprogramming*, 31:31–35, 1991.

[177] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, 1996.

[178] R. S. Nikhil. Id (Version 90.1) Reference Manual. *Technical Report CSG Memo 284-2, Lab. for Computer Science, MIT*, July 1991.

[179] E. Nöker, J. Smetsers, M. van Eekelen, and M. Plasmeijer. Concurrent Clean. *PARLE'91 - Springer-Verlag LNCS 505*, pages 202–220, 1991.

[180] OpenMP Architecture Review Board. OpenMP: Simple, Portable, Scalable SMP Programming, Web Page, www.openmp.org.

[181] Palacharla, S. and Jouppi, N. P. and Smith J. E. . Complexity-Effective Superscalar Processors. In *International Symposium on Computer Architecture (ISCA)*, pages 206–218, 1997.

[182] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *761*, page 55. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 31 1998.

[183] D. L. Parnas. On the Criteria to Be Used in Decompsing Systems into Modules. *Communications of the ACM*, 15(2):237–267, 1972.

[184] D. L. Parnas. On a 'Buzzword': Hierarchical Structure. In *IFIP Congress 74*, pages 336–339, 1974.

[185] D. L. Parnas. Designing Software for Extension and Contraction. In *3rd. Conference on Software Enginnering*, pages 264–277, 1978.

[186] B. K. Pasquale and G. Plyzos. A Statis Analysis of I/O Characterization of Scientific Applications in a Production Workload. In *Proceedings of Conference Supercomputing'93*, pages 388–397, November 1993.

[187] V. C. Paula, G. R. R. Justo, and P. R. F. Cunha. ZCL: A Formal Framework for Specifying Dynamic Software Architectures. In *Nineth European Workshop on Dependable Computing (EWDC'98)*, may 1998.

[188] C. A. Petri. Kommunikation mit Automaten. *Technical Report RADC-TR-65-377, Griffiths Air Force Base, New York*, 1(1), 1966.

[189] S. L. Peyton Jones, C. Clack, and J. Salkild. GRIP - A High-Performance Architecture for Parallel Graph Reduction. *FPCA'87: Conference on Functional Programming Languages and Computer Architecture - Springer-Verlag LNCS 274*, pages 98–112, 1987.

[190] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. *POPL'96 - Symposium on Principles of Programming Languages, ACM Press*, pages 295–308, January 1996.

[191] S. L. Peyton Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-strict Functional Language. *FPCA'91 - Functional Programming and Computer Architecture, LNCS 523*, pages 636–666, September 1991.

[192] R. S. Pressman. *Software Engineering, A Practioner's Approach.* McGraw-Hill, 1997.

[193] J. M. Purtilo. The POLYLITH software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[194] S. Roch and P. Starke. Manual: Integrated Net Analyzer Version 2.2. *Humboldt-Universität zu Berlin, Institut für Informatik, Lehrstuhl für Automaten- und Systemtheorie*, 1999.

[195] N. Röjemo. NHC Haskell Compiler, Web Page, http://www.cs.york.ac.uk/fp/nhc98, The University of York, Department of Computer Science, Functional Programmi Research Group.

[196] A. Rowstron and A Wood. BONITA, a set of tuple space primitives for distributed coordination. In *30^{th} Hawaii International Conference on System Sciences (HICSS-30)*, volume 1, pages 379–388. IEEE Press, jan 1997.

[197] P. M. Sansom and S. L. Peyton Jones. Time and Space Profiling for Non-Strict, Higher-Order Functional Languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–366. ACM Press, 1995.

[198] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Department of Computer Science, University of Glasgow, July 1995.

[199] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21:413–510, 1989.

[200] A. C. Shaw. Software Descriptions with Flow Expressions. *IEEE Transactions on Software Enginnering*, SE-4(3):299–325, May 1978.

[201] M. Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *International Workshop on Studies of Software Design*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

[202] M. Shaw, R. DeLine, Klein D. V., T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Transactions on Software Engineering*, 21(4):314–335, 1995.

[203] G. S. Shiralkar, R.E. Stephenson, W. Joubert, and Bart von Bloemen Waanders. Falcon: A Production Quality Distributed Memory Reservoir Simulator. In *Proceedings of SPE Reservoir Simulation Symposium*. Society for Petroleum Engineers, june 1997.

[204] D. B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30:123–169, June 1998.

[205] I. Somnerville and G. Dean. PCL: A Language for Modelling Evolving System Architectures. *Software Engineering Journal*, 8(2):111–121, March 1996.

[206] F. V. Souza and R. D. Lins. Analyzing the Space Behavior of Functional Programs. *CLAPF'99 - 3rd Latin American Conference on Functional Programming* , pages 147–157, March 1999.

[207] Strand Software Technologies Ltd., SSTL. *Strand User Manual*, 1989.

[208] F. Taylor. Parallel Functional Programming by Partitioning. *PhD Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London*, January 1997.

[209] The Power PC G5 Team. Power PC G5, The World First 64-bit Desktop Processor (White Paper). Technical report, Apple Computers, Inc., July 2003. http://www.apple.com/powermac/.

[210] The VIA Consortium. Virtual Interface Architecture (VIA), web page, http://www.viarch.org.

[211] S. Thompson. Haskell, The Craft of Functional Programming. *Addison-Wesley Publishers Ltd.*, 1996.

[212] S. Thompson and R. D. Lins. The Categorical Multi-Combinator Machine: CMCM. *The Computer Journal*, 35(2):170–176, 1992.

[213] R. Tolksdorf. Coordinating Services in Open Distributed Systems with Laura. In *The First International Conference on Coordination Models, Languages and Applications (COORDINATION'96)*, pages 386–402, April 1996.

[214] P. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. P. L. Jones. GUM: A Portable Parallel Implementation of Haskell. *PLDI'96 - Programming Languages Design and Implementation*, pages 79–88, 1996.

[215] P. W. Trinder and E. et al Barry Jr. GpH: An Architecture-Independent Functional Language. *Submitted to IEEE Transaction on Software Engineering*, July 1998.

[216] P. W. Trinder, K. Hammond, H-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[217] P. W. Trinder, H-W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4/5):469–510, July 2002.

[218] A. M. Turing. On Computable Numbers with an Application to the Entscheidungsproblem. *Proceedings of London Math Society*, 42:230–265, 1936.

[219] A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2:153–163, 1937.

[220] D. A. Turner. Recursion Equations as a Programming Language. *Functional Programming and its Applications*, pages 1–28, 1982.

[221] D. A. Turner. An Overview of Miranda. *Research Topics in Functional Programming, Addison Wesley*, 1990.

[222] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[223] R. Valk and G. Vidal Naquet. Petri Nets and Regular Languages. *Journal of Computer and System Sciences*, 23:299–325, 1981.

[224] H. K. Versteeg. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method* . Addison-Wesley, February 1996.

[225] P. Wadler. Lazy vs. Strict. *ACM Computing Surveys*, June 1996. http://cm.bell-labs.com/cm.cs/who/wadler/papers/lazyvsstrict/lazyvsstrict.ps.gz.

[226] I. Watson. Simulation of a Physical EDS Machine Architecture. *Technical Report, Department of Computer Science, University of Manchester, UK*, September 1989.

[227] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: A Parallel Architecture for Declarative Programming. *15th Annual ACM Symposium on Computer Architecture*, pages 124–136, 1988.

[228] P. Watson and I. Watson. Evaluating Functional Programs on Flagship Machine. *FPCA'87: Conference on Functional Programming Languages and Computer Architecture - Springer-Verlag LNCS 274*, pages 80–97, September 1987.

[229] Weber M. and Kindler E. The Petri Net Markup Language. *Lecture Notes in Computer Science*, 2002. Submitted for publication in april 2002.

[230] P. Wegner. Coordination as Constrained Interaction. In *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, pages 28–33. Springer-Verlag WICS, July 1996. Lecture Notes in Computer Science 1061, Springer-Verlag.

[231] S. Wells. Innovative Computing Laboratory Annual Report. Technical report, University of Tenessee, 2002. http://icl.cs.utk.edu/files/2002Report.pdf.

[232] D. J. Whalen, D. E. Hollowell, and J. S. Hendriks. MCNP: Photon Benchmark Problems. Technical Report LA-12196, Los Alamos National Laboratory, 1991.

[233] H-C. Yen. On the Regularity of Petri Net Languages. *Information and Computation*, 124:168–181, 1996.

[234] A. Zimmermann, J. Freiheit, R. German, and G. Hommel. Petri Net Modelling and Performability Evaluation with TimeNET 3.0. In *11th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'2000)*, pages 188–202. Lecture Notes in Computer Sciente, 2000.

APÊNDICE A

A BIBLIOTECA DE ESQUELETOS MPI (CÓDIGO #)

A.1 INTERFACES BÁSICAS

```
library Basic_Interfaces where

interface ISource # in → () behaving as: in?
interface ITarget # () → out behaving as: out!
interface ISourceTarget # in → out behaving as: seq {out!;in?}
```

A.2 ESQUELETOS ELEMENTARES

A.2.1 Prim_OneToAll

```
component Prim_OneToAll<N, HowToDivide> with

import Basic_Interfaces

unit q # ISource
unit p # ITarget

connect * q→out to p←in, buffered

replicate N p connections out: HowToDivide

unify p[1] # in → (), q # () → out to r # ISourceTarget
```

A.2.2 Prim_AllToOne

```
component Prim_AllToOne<N, HowToCombine> with

import Basic_Interfaces

unit p # ISource
unit q # ITarget

connect * q→out to p←in, buffered

replicate N p connections in: NowToCombine

unify p[1] () → out, q # in → () to r # ISourceTarget
```

A.2.3 Prim_AllToAll

```
component Prim_AllToAll<N, HowToDivide, HowToCombine> with

import Basic_Interfaces

index i range [1,N]

unit q # ISourceTarget

connect * q→out to q←in, buffered

factorize N q in [/ p[i] /] connections in<>: HowToCombine,
                                       out<>: HowToDivide
```

A.3 BCAST

```
component Bcast<N> with

import Basic_Interfaces

use Skeletons.MPI.Primitive.Prim_OneToAll

index i range [1,N-1]

interface class IBCast

interface IBCastRoot # ISourceTarget specializes IBCast
interface IBCastPeer # ITarget specializes IBCast

unit bcast as Prim_OneToAll<N, broadcast>

unit peer[0] # IBCastRoot
[/ unit peer[i] # IBCastPeer /]

alias root to peer[0]

assign root to bcast.r
[/ assign peer[i] to bcast.p[i+1] /]
```

A.4 GATHER

```
component Gather<N, HowToCombine> with

index i range [1,N-1]

import Basic_Interfaces

use Skeletons.MPI.Primitive.Prim_AllToOne

unit gather as Prim_AllToOne<N, HowToCombine>

interface class IGather

interface IGatherRoot # ISourceTarget specializes IGather
interface IGatherPeer # ISource specializes IGather

unit root # IGatherRoot
[/ unit peer[i] # IGatherPeer /]

assign root to gather.root
[/ assign peer[i] to gather.p[i+1] /]
```

A.5 GATHERV

```
component Gatherv<N, HowToCombine> with

import Basic_Interfaces

use Skeletons.MPI.Primitive.Prim_AllToOne

index i range [1,N-1]

unit gatherv as Prim_AllToOne<N, HowToCombine>

interface class IGatherv

interface IGathervRoot # ISourceTarget specializes IGatherv
interface IGathervPeer # ISource specializes IGatherv

unit root # IGathervRoot
[/ unit peer[i] # IGathervPeer /]

assign root to gatherv.root
[/ assign peer[i] to gatherv.p[i+1] /]
```

A.6 ALLGATHER

```
component AllGather<N, HowToCombine> with
import Basic_Interfaces
use Skeletons.MPI.Primitive.Prim_AllToAll
index i range [1,N]
unit allgather as Prim_AllToAll<N, broadcast, HowToCombine>
interface IAllGather # ISourceTarget is
[/ unit peer[i] # IAllGather /]
[/ assign peer[i] to allgather.p[i] /]
```

A.7 ALLGATHERV

```
component AllGatherv<N, HowToCombine> with
import Basic_Interfaces
use Skeletons.MPI.Primitive.Prim_AllToAll
index i range [1,N]
unit allgather as Prim_AllToAll<N, broadcast, HowToCombine>
interface IAllGatherv # ISourceTarget
[/ unit peer[i] # IAllGatherv /]
[/ assign peer[i] to allgather.p[i] /]
```

A.8 SCATTER

```
component Scatter<N, HowToDivide> with
import Basic_Interfaces
use Skeletons.MPI.Primitive.Prim_OneToAll
index i range [1,N-1]
unit scatter as Prim_OneToAll<N, HowToDivide>
interface class IScatter

interface IScatterRoot # ISourceTarget specializes IScatter
interface IScatterPeer # ITarget specializes IScatter

unit root # IScatterRoot
[/ unit peer[i] # IScatterPeer /]

assign root to scatter.root
[/ assign peer[i] to scatter.p[i+1] /]
```

A.9 SCATTERV

```
component Scatterv<N, HowToDivide> with
import Basic_Interfaces
use Skeletons.MPI.Primitive.Prim_OneToAll
index i range [1,N-1]
interface class IScatterv

interface IScattervRoot # ISourceTarget specializes IScatterv
interface IScattervPeer # ITarget specializes IScatterv

unit scatterv as Prim_OneToAll<N, HowToDivide>

unit root # IScattervRoot
[/ unit peer[i] # IScattervPeer /]

assign root to scatterv.root
[/ assign peer[i] to scatterv.p[i+1] /]
```

A.10 ALLTOALL

```
component AllToAll<N, HowToDivide, HowToCombine> with
import Basic_Interfaces
use Skeletons.MPI.Primitive.Prim_AllToAll
index i range [1,N]
unit alltoall as Prim_AllToAll<N, HowToDivide, HowToCombine>
interface IAllToAll # ISourceTarget
[/ unit peer[i] # IAllToAll /]
[/ assign peer[i] to alltoall.p[i] /]
```

A.11 ALLTOALLV

```
component AllToAllv<N, HowToDivide, HowToCombine> with
import Basic_Interfaces
use Skeletons.MPI.Primitive.AllToAll
index i range [1,N]
unit alltoallv as Prim_AllToAll<N, HowToDivide, HowToCombine>
interface IAllToAllv # ISourceTarget
[/ unit peer[i] # IAllToAllv /]
[/ assign peer[i] to alltoallv.p[i] /]
```

A.12 REDUCE

```
component Reduce<N, OPERATION, DATA_TYPE> with
import Basic_Interfaces
use Skeletons.MPI.Primitive.Prim_AllToOne
index i range [1,N-1]
unit reduce as Prim_AllToOne<N, OPERATION>
interface class IReduce
interface IReduceRoot # ISourceTarget specializes IReduce
interface IReducePeer # ISource specializes IReduce
unit root # IReduceRoot in → out
[/ unit peer[i] # IReducePeer /]
assign root to reduce.root
[/ assign peer[i] to reduce.p[i+1] /]
```

A.13 ALLREDUCE

```
component AllReduce<N,OPERATION,DATA_TYPE> with
import Basic_Interfaces
use Skeletons.MPI.Primitive.Prim_AllToAll
index i range [1,N]
interface IAllReduce # ISourceTarget
unit allreduce as Prim_AllToAll<N,broadcast,OPERATION>
[/ unit peer[i] # IAllReduce /]
[/ assign peer[i] to allreduce.p[i] /]
```

A.14 REDUCE_SCATTER

```
component Reduce_Scatter<N,OPERATION,DATA_TYPE> with

import Basic_Interfaces

use Skeletons.MPI.Primitive.Prim_OneToAll

index i range [1,N-1]

unit reduce_scatter as Prim_OneToAll<N, OPERATION>

interface class IReduceScatter

interface IReduceScatterRoot # ISourceTarget specializes IReduceScatter
interface IReduceScatterPeer # ITarget specializes IReduceScatter

unit root # IReduceScatterRoot
[/ unit peer[i] # IReduceScatterPeer /]

assign root to reduce_scatter.root
[/ assign peer[i] to reduce_scatter.p[i+1] /]
```

A.15 SCAN

```
component MPI_Scan<N,OPERATION,DATA_TYPE> with

import Basic_Interfaces

use Skeletons.MPI.Primitive.Prim_OneToAll

index i range [1,N]
index j range [1,i]   –sub-index

[/unit a[i] as Prim_OneToAll<i, OPERATION> /]

[/unify [/a[j+N-i].p[j]/] # in → _/ to p[i] # in → _ /]
```

APÊNDICE B

A IMPLEMENTAÇÃO # DE PROGRAMAS DE NPB (NAS PARALLEL BENCHMARKS)

B.1 O KERNEL EP

B.1.1 Configuração

```
component EP<NO_NODES,MK, MM, NN, NK, NQ, EPSILON, A, S> with

index i range [1..NO_NODES]

use Skeletons.Collective.AllReduce
use EP_FM -- EP Functional Module

interface IEP # sx like IAllReduce Double
              # sy like IAllReduce Double
              # q  like IAllReduce UDVector
              behaviour: seq {do sx; do sy; do q}

unit sx_comm as AllReduce<NO_NODES,MPI_SUM, MPI_DOUBLE>
unit sy_comm as AllReduce<NO_NODES, MPI_SUM, MPI_DOUBLE>
unit q_comm as AllReduce<NO_NODES, MPI_SUM, MPI_DOUBLE>

[/ unify sx_comm.p[i] # sx, sy_comm.p[i] # sy, q_comm.p[i] # q
        to ep_unit[i] # IEP as EP_FM (EP_Params i NO_NODES MK MM NN NK NQ EPSILON A S, sx.in, sy.in, q.in) → (sx.out, sy.out, q.out) /]
```

B.2 O KERNEL IS

B.2.1 Configuração

```
component IS<PROBLEM_CLASS, NUM_PROCS, MAX_KEY_LOG2, NUM_BUCKETS_LOG2, TOTAL_KEYS_LOG2, MAX_ITERATIONS, MAX_PROCS, TEST_ARRAY_SIZE> with

index i range [1, NUM_PROCS]

use IS_FM -- IS Functional Module
use Skeletons.Collective.AllReduce
use Skeletons.Collective.AllToAllv
use Skeletons.Misc.RShift

interface IIS # bs* like IAllReduce (UArray Int Int)
              # kb* like IAllToAllv (Int, Ptr Int)
              # k    like RShift      Int
              behaviour: seq {repeat seq {do bs; do kb} until <bs & kb>; do k}

unit bs_comm as AllReduce<NUM_PROCS, MPI_SUM, MPI_INTEGER>
unit kb_comm as AllToAllv<NUM_PROCS>
unit k_shift as RShift<NUM_PROCS> 0 → _

[/ unify bs_comm.p[i] # bs,
         kb_comm.p[i] # kb,
         k_comm.p[i]  # k
         to is_unit[i] # IIS as IS_FM (IS_Params PROBLEM_CLASS NUM_PROCS MAX_KEY_LOG2 textscnum_buckets_log2 TOTAL_KEYS_LOG2
                                              MAX_ITERATIONS MAX_PROCS TEST_ARRAY_SIZE, bs.in, kb.in, k.in) → (bs.out, kb.out, k.out) /]
```

B.3 O KERNEL CG

B.3.1 Configuração

B.3.1.1 Esqueleto Transpose

```
component Transpose<DIM, COL_FACTOR>

index i, j range [1..DIM]
index k   range [1..COL_FACTOR]
```

```
interface ITranspose x::UDVector → w::UDVector behaviour: seq { w!; x? }

[/ unit trans[i][j] # ITranspose x<DIM> → w<DIM>:broadcast /]

[/ connect trans[i][j] → w[k] to trans[k][i] ← x[j] /]

[/ factorize trans[i][j] # w → x to [/ u[(.i-1)*COL_FACTOR+k][.j] # w → x /] connections w<>: sum, x<>: split_vector /]
```

B.3.1.2 Componente CG

```
component CG<DIM, COL_FATOR, NA, NONZER, SHIFT, NITER, RCOND ZVV> # () → (zeta, x) with

use Skeletons.MPI.Collective.AllReduce
use Transpose
use CG_FM −− CG Functional Module

index i range [1..DIM]
index j range [1..COL_FACTOR]

interface ICG # () → (x::Array Int Double, zeta::Double)
                # q**          like ITranspose
                # r*           like ITranspose
                # rho**        like IAllReduce Double
                # aux**        like IAllReduce Double
                # rnorm*       like IAllReduce Double
                # norm_temp_1* like IAllReduce Double
                # norm_temp_2* like IAllReduce Double
                behaviour: repeat seq {do rho; repeat seq {do q; do aux; do rho} until <q & aux & rho>;
                                       do r; do rnorm; do norm_temp_1; do norm_temp_2;}
                           until <r & rnorm & q & aux & rho & norm_temp_1 & norm_temp_2>

unit q_comm as Transpose<DIM, DIM * DIM * COL_FACTOR>
unit r_comm as Transpose<DIM, DIM * DIM * COL_FACTOR>
[/ unit rho_comm[i] as AllReduce<DIM * COL_FACTOR, MPI_SUM, MPI_DOUBLE> /]
[/ unit aux_comm[i] as AllReduce<DIM * COL_FACTOR, MPI_SUM, MPI_DOUBLE> /]
[/ unit rnorm_comm[i] as AllReduce<DIM * COL_FACTOR, MPI_SUM, MPI_DOUBLE> /]
[/ unit norm_temp_1__comm[i] as AllReduce<DIM * COL_FACTOR, MPI_SUM, MPI_DOUBLE> /]
[/ unit norm_temp_2__comm[i] as AllReduce<DIM * COL_FACTOR, MPI_SUM, MPI_DOUBLE> /]

[/ unify q_comm.u[i][j] # q,
         r_comm.u[i][j] # r,
         rho_comm[i].p[j] # rho,
         aux_comm[i].p[j] # aux,
         rnorm_comm[i].p[j] # rnorm
         norm_temp_1_comm[i].p[j] # norm_temp_1
         norm_temp_2_comm[i].p[j] # norm_temp_2
         to cg[i][j] # ICG as CG_FM (CG_Params DIM (DIM*COL_FACTOR) NA NONZER SHIFT NITER RCOND ZVV,
                                     q.x, r.x, rho.in, aux.in, rnorm.in, norm_temp_1.in, norm_temp_1.in)
                                         → (q.w, r.w, rho.out, aux.out, rnorm.out, norm_temp_1.out, norm_temp_2.out) /]
```

B.4 AS APLICAÇÕES SIMULADAS SP E BT

B.4.1 Esqueleto X_Solve

```
component X_solve<DIM> with

index i, j range [0..DIM-1]

interface Ix_solve # (rcp_fe*, rcp_bs* :: UArray Int Double) → (env_fe*, env_bs* :: UArray Int Double)
          behaviour: seq { repeat seq {env_fe!; rcp_fe?} until <rcp_fe>;
                           repeat seq {env_bs!; rcp_bs?} until <rcp_bs>;
                   }

[/ unit xsolve_unit[i][j] # Ix_solve /]

[/ connect xsolve_unit[i][j] → env_fe to xsolve_unit[i][(j+1) mod DIM] ← rcp_fe
   connect xsolve_unit[i][j] → env_bs to xsolve_unit[i][(j+DIM-1) mod DIM] ← rcp_bs /]
```

B.4.2 Esqueleto Y_Solve

```
component Y_Solve<DIM> with

index i, j range [0..DIM-1]

interface Iy_solve # (rcp_fe*, rcp_bs* :: UArray Int Double) → (env_fe*, env_bs* :: UArray Int Double)
          behaviour: seq {repeat seq {env_fe!;rcp_fe?} until <rcp_fe>;
                          repeat seq {env_bs!;rcp_bs?} until <rcp_bs>;}

[/ unit ysolve_unit[i][j] # Iy_solve /]

[/ connect ysolve_unit[i][j] → env_fe to ysolve_unit[(i+1) mod DIM][j] ← rcp_fe
```

```
  connect ysolve_unit[i][j] → env_bs to ysolve_unit[(i+DIM-1) mod DIM][j] ← rcp_bs /]
```

B.4.3 Esqueleto Z_Solve

```
component Z_Solve<DIM> with

index i, j range [0..DIM-1]

interface Iz_solve # (rcp_fe*, rcp_bs* :: UArray Int Double) → (env_fe*, env_bs* :: UArray Int Double)
          behaviour: seq {repeat seq {env_fe!;rcp_fe?} until <rcp_fe>;
                          repeat seq {env_bs!;rcp_bs?} until <rcp_bs>; }

[/ unit zsolve_unit[i][j] # Iz_solve /]

[/ connect zsolve_unit[i][j] → env_fe to zsolve_unit[(i+1) mod DIM][(j+DIM-1) mod DIM] ← rcp_fe
   connect zsolve_unit[i][j] → env_bs to zsolve_unit[(i+DIM-1) mod DIM][(j+1) mod DIM] ← rcp_bs /]
```

B.4.4 Esqueleto Copy_Faces

```
component Copy_Faces<DIM> with

index i, j range [0..DIM-1]

interface ICopy_faces # (rec1, rec2, rec3, rec4, rec5, rec6 :: Double) → (env1, env2, env3, env4, env5, env6 :: Double)
          behaviour: par {env1!; env2!; env3!; env4!; env5!; env6!; rec1?; rec2?; rec3?; rec4?; rec5?; rec6?}

[/ unit cf_unit[i][j] # ICopy_faces /]

[/ connect cf_unit[i][j] → env1 to cf_unit[i][(j+1) mod DIM] ← rec1
   connect cf_unit[i][j] → env2 to cf_unit[i][(j+DIM-1) mod DIM] ← rec2
   connect cf_unit[i][j] → env3 to cf_unit[(i+1) mod DIM][j] ← rec3
   connect cf_unit[i][j] → env4 to cf_unit[(i+DIM-1) mod DIM][j] ← rec4
   connect cf_unit[i][j] → env5 to cf_unit[(i+1) mod DIM][(j+DIM-1) mod DIM] ← rec5
   connect cf_unit[i][j] → env6 to cf_unit[(i+DIM-1) mod DIM][(j+1) mod DIM] ← rec6 /]
```

B.4.5 Esqueleto SolveSystem (de Aplicação)

```
component SolveSystem <DIM> with

use Skeletons.Collective.AllReduce
use Skeletons.Collective.BCast
use Skeletons.Collective.Reduce
use X_Solve
use Y_Solve
use Z_Solve
use Copy_Faces

index i,j range [0..DIM-1]

interface ISolveSystem # niter          like IBCast Int
                       # dt             like IBCast Double
                       # grid_points    like IBCast UArray Int Int
                       # out_buffer*    like ICopy_faces
                       # out_bufferx*   like Ix_solve
                       # out_buffery*   like Iy_solve
                       # out_bufferz*   like Iz_solve
                       # rms_work       like IAllReduce UArray Int Double
                       # rms            like IAllReduce UArray Int Double
                       # t              like IReduce Double
                       # tmax           like IReduce Double
          behaviour: seq{
                         do niter;
                         do dt;
                         do grid_points;
                         repeat alt {do out_buffer;
                                     do out_bufferx;
                                     do out_buffery;
                                     do out_bufferz; } until <out_buffer & out_bufferx &
                                                              out_buffery & out_bufferz>;
                         do rms_work;
                         do rms;
                         do out_buffer;
                         do rms_work;
                         do rms;
                         do t;
                         do tmax; }

unit niter_comm         as BCast       <DIM*DIM>
unit dt_comm            as BCast       <DIM*DIM>
unit grid_points_comm   as BCast       <DIM*DIM>
unit out_buffer_comm    as Copy_Faces  <DIM>
unit out_bufferx_comm   as X_Solve     <DIM>
unit out_buffery_comm   as Y_Solve     <DIM>
```

```
unit out_bufferz_comm  as Z_Solve      <DIM>
unit rms_work_comm     as AllReduce    <DIM*DIM, MPI_SUM, MPI_DOUBLE>
unit rms_comm          as AllReduce    <DIM*DIM, MPI_SUM, MPI_DOUBLE>
unit t_comm            as Reduce       <DIM*DIM, MPI_MAX, MPI_DOUBLE>
unit tmax_comm         as Reduce       <DIM*DIM, MPI_MAX, MPI_DOUBLE>

[/ unify niter_comm.peer[i+j*DIM]          # niter,
         dt_comm.peer[i+j*DIM]             # dt,
         grid_points_comm.peer[i+j*DIM]    # grid_points,
         out_buffer_comm[i][j]             # out_buffer,
         out_bufferx_comm[i][j]            # out_bufferx,
         out_buffery_comm[i][j]            # out_buffery,
         out_bufferz_comm[i][j]            # out_bufferz,
         rms_work_comm[i][j]               # rms_work,
         rms_comm[i][j]                    # rms
         t_comm.peer[i+j*DIM]              # t,
         tmax_comm.peer[i+j*DIM]           # tmax to ss[i][j] # ISolveSystem /]
```

B.4.6 Componente SP

```
component SP<NPROCS,PROBLEM_SIZE,DT,ITMAX> with

use SolveSystem
use SP_FM -- SP Functional Module

index i range [1..DIM]

interface ISP # ISolveSystem

unit sp_cluster as SolveSystem<DIM>

[/ unit sp[i] # ISP as SP_FM () → ()
   assign sp[i] to sp_cluster.ss[i] /]
```

B.4.7 Componente BT

```
component BT<NPROCS,PROBLEM_SIZE,DT,ITMAX> with

use SolveSystem
use BT_FM -- BT Functional Module

index i range [1..DIM]

interface IBT # ISolveSystem

unit sp_cluster as SolveSystem<DIM>

[/ unit bt[i] # IBT as BT_FM () → ()
   assign bt[i] to sp_cluster.ss[i] /]
```

B.5 A APLICAÇÃO SIMULADA LU

B.5.1 Esqueleto Exchange_1b

```
component Exchange_1b < XDIV , YDIV ,itmax> with

index m range [0..( YDIV -1)]
index n range [0..( XDIV -1)]

interface Exchange_1b # (from_north**,from_west**, from_south**, from_east** :: UArray (Int,Int) Double)
                      → (to_south**, to_east**, to_north**,tp_west** :: UArray (Int,Int) Double)
        behaviour: repeat { seq {repeat seq{from_north?;
                                            from_west?;
                                            to_south!;
                                            to_east!} until <from_north & from_west &
                                                             to_south & to_east>;
                                 repeat seq{from_south?;
                                            from_east?;
                                            to_north!;
                                            to_west!} until <from_south & from_east &
                                                             to_north & to_west>
                                } until itmax

[/unit bigLoop[n][m] # Exchange_1b /]

[/ connect bigLoop[n][m] → to_south to bigLoop[(n+1) mod XDIV ][m] ← from_north
   connect bigLoop[n][m] → to_east to bigLoop[n][(m+1) mod YDIV ] ← from_west
   connect bigLoop[n][m] → to_north to bigLoop[(n+ XDIV -1) mod XDIV ][m] ← from_south
   connect bigLoop[n][m] → to_west to bigLoop[n][(m+ YDIV -1) mod YDIV ] ← from_east /]
```

B.5.2 Esqueleto Exchange_3b

```
component Exchange_3b < XDIV , YDIV > with

index m range [0..(YDIV-1)]
index n range [0..(XDIV-1)]

interface IExchange_3b # (from_north*,from_south*, from_east*, from_west*::UArray Int Double)
                         →(to_north*, to_south*, to_east*, to_west*:: UArray Int Double)
          behaviour: repeat seq {to_south!;from_noth?;
                                 to_north!;from_south?;
                                 to_east!; from_west?;
                                 to_west!; from_east?} until to_south

[/ unit g1[n][m] # IExchange_3b /]

[/ connect g1[n][m] → g1_ts to g1 [(n+1) mod XDIV ][m]← g1_fn
   connect g1[n][m] → g1_tn to g1 [(n+ XDIV -1) mod XDIV ][m]← g1_fs
   connect g1[n][m] → g1_te to g1 [n][(m+1) mod YDIV ]← g1_fw
   connect g1[n][m] → g1_tw to g1 [n][(m+ YDIV -1) mod YDIV ]← g1_fe /]
```

B.5.3 Esqueleto Exchange_4

```
component Exchange_4 < XDIV , YDIV > with

index n range [0..(YDIV-2)]
index s range [1..(YDIV-1)]
index l range [0..(XDIV-2)]
index r range [1..(XDIV-1)]

index i range [1..(YDIV-2)]
index j range [1..(XDIV-2)]

interface IExchange_4

interface IExchange_4_Null specializes IExchange_4

interface IExchange_4_Border # (in::UArray Int Double) → (out::UArray Int Double)
          behaviour: seq {out!;in?}
          specializes IExchange_4

interface IExchange_4_Corner_NW # (in1, in2::UArray Int Double) → ()
          behaviour: seq {in1?;in2?}
          specializes IExchange_4

interface IExchange_4_Corner_SE # ()→(out1, out2::UArray Int Double)
          behaviour: seq {out1!;out2!}
          specializes IExchange_4

[/ unit h0[i][j] # IExchange_4_Null /]

unit h0[0][0] # IExchange_4_Corner_NW
unit h0[ XDIV -1][ YDIV -1] # IExchange_4_Corner_SE

[/ unit h0[n][0] # IExchange_4_Border /]
[/ unit h0[s][ XDIV -1] # IExchange_4_Border /]

[/ unit h0[l][0] # IExchange_4_Border /]
[/ unit h0[r][ YDIV -1] # IExchange_4_Border /]

[/ connect h0[0][s] → out to h0[0][s-1] ← in /]
[/ connect h0[ XDIV -1][s] → out to h0[ XDIV -1][s-1] ← in /]

[/ connect h0[r][0] → out to h0[r-1][0] ← in /]
[/ connect h0[r][ YDIV -1] → out to h0[r-1][ YDIV -1] ← in /]
```

B.5.4 Esqueleto Exchange_5

```
component Exchange_5 < XDIV , YDIV > with

index m range [0..(YDIV-1)]
index n range [0..(XDIV-1)]

index i range [1..(YDIV-2)]
index j range [1..(XDIV-2)]

interface class IExchange_5

interface IExchange_5_Null specializes IExchange_5

interface IExchange_5_Top # (in::UArray Int Double)→() behaviour: in?
          specializes IExchange_5

interface IExchange_5_Bottom # ()→(out::UArray Int Double) behaviour: out!
```

```
        specializes IExchange_5

interface IExchange_5_Side # (in::UArray Int Double)→(out::UArray Int Double)
        behaviour: seq {out!;in?}
        specializes IExchange_5

[/ unit h1[i][m] # IExchange_5_Null /]

unit h1[0][0] # IExchange_5_Top
unit h1[0][ YDIV -1] # IExchange_5_Top

unit h1[ XDIV -1][0] # IExchange_5_Bottom
unit h1[ XDIV -1][ YDIV -1] # IExchange_5_Bottom

[/ unit h1[j][0] # IExchange_5_Side
   unit h1[j][ YDIV -1] # IExchange_5_Side /]

[/ connect h1[l][0]→out to h1[l-1][0]←in /] [/ connect h1[l][ YDIV -1]→out to h1[l-1][ YDIV -1]←in /]
```

B.5.5 Esqueleto Exchange_6

```
component Exchange_6 < XDIV , YDIV > with

index m range [0..(YDIV-1)]
index n range [0..(XDIV-1)]

index i range [1..(YDIV-2)]
index j range [1..(XDIV-2)]

interface class IExchange_6

interface IExchange_6_Null specializes IExchange_6

interface IExchange_6_Left # (in::UArray Int Double) → () behaviour: in?
        specializes IExchange_6

interface IExchange_6_Right # ()→(out::UArray Int Double) behaviour: out!
        specializes IExchange_6

interface IExchange_6_Side # (in::UArray Int Double) → (out::UArray Int Double)
        behaviour: seq {out!; in?}
        specializes IExchange_6

[/ unit h1[i][m] # IExchange_6_Null /]

unit h1[0][0] # IExchange_6_Left
unit h1[0][ YDIV -1] # IExchange_6_Left

unit h1[ XDIV -1][0] # IExchange_6_Right
unit h1[ XDIV -1][ YDIV -1] # IExchange_6_Right

[/ unit h1[j][0] # IExchange_6_Side
   unit h1[j][ YDIV -1] # IExchange_6_Side /]

[/ connect h1[0][l] → out to h1[0][l-1] ← in /] [/ connect h1[ XDIV -1][l] → out to h1[ XDIV -1][l-1 ← in /]
```

B.5.6 Componente LU (Esqueleto de Aplicação)

```
component LU <NPROCS,PROBLEM_SIZE,DT_DEFAULT,ITMAX> with

#define D ilog2(nprocs)/2
#define XDIV (ipow2(if(D*2 == ilog2(nprocs), D, D + 1)))
#define YDIV (ipow2(d))

use Skeletons.MPI.AllReduce
use Skeletons.MPI.Bcast
use Exchange_1b
use Exchange_3b
use Exchange_4
use Exchange_5
use Exchange_6
use LU_FM −− LU Functional Module

index m range [1.. YDIV]
index n range [1.. XDIV]

interface ILU # ipr, inorm, itmax, nx0, ny0, nz0 like IBCast Int
              # dt, omega                       like IBCast Double
              # tolrsd                          like IBCast MyArray1d
              # rsdnm*, errnm                   like IAllReduce MyArray1d
              # frc1, frc2, frc3                like IAllReduce Double
              # rsd1                            like IExchange_1b
              # rsd0, u1                        like IExchange_3b
              # phis                            like IExchange_4
              # phiver                          like IExchange_5
```

```
        # phihor                          like IExchange_6
      behaviour: seq { do ipr;
                       do inorm;
                       do itmax;
                       do nx0;
                       do ny0;
                       do nz0;
                       do dt;
                       do omega;
                       do tolrsd;
                       do rsd0;
                       do u1;
                       do rsdnm;
                       do rsd1;
                       do u1;
                       do rsdnm;
                       do errnm;
                       do phis;
                       do frc1;
                       do phiver;
                       do frc2;
                       do phihor;
                       do frc3 }

unit ipr_comm      as BCast          < xdiv * ydiv >
unit inorm_comm    as BCast          < xdiv * ydiv >
unit itmax_comm    as BCast          < xdiv * ydiv >
unit nx0_comm      as BCast          < xdiv * ydiv >
unit ny0_comm      as BCast          < xdiv * ydiv >
unit nz0_comm      as BCast          < xdiv * ydiv >
unit dt_comm       as BCast          < xdiv * ydiv >
unit omega_comm    as BCast          < xdiv * ydiv >
unit tolrsd_comm   as BCast          < xdiv * ydiv >
unit rsd0_comm     as Exchange_3b    < xdiv , ydiv >
unit u1_comm       as Exchange_3b    < xdiv , ydiv >
unit rsdnm_comm    as AllReduce      < xdiv * ydiv , MPI_DOUBLE, MPI_SUM>
unit ssor_comm     as Exchange_1b    < xdiv , ydiv , itmax, nz>
unit errnm_comm    as AllReduce      < xdiv * ydiv , MPI_DOUBLE, MPI_SUM>
unit phis_comm     as Exchange_4     < xdiv , ydiv >
unit frc1_comm     as AllReduce      < xdiv * ydiv , MPI_DOUBLE, MPI_SUM>
unit phiver_comm   as Exchange_5     < xdiv , ydiv >
unit frc2_comm     as AllReduce      < xdiv * ydiv , MPI_DOUBLE, MPI_SUM>
unit phihor_comm   as Exchange_6     < xdiv , ydiv >
unit frc3_comm     as AllReduce      < xdiv * ydiv , MPI_DOUBLE, MPI_SUM>

[/ unify ipr_comm.p[n][m]            # ipr ,
         inorm_comm.p[n][m]          # inorm ,
         itmax_comm.p[n][m]          # itmax ,
         dt_comm.p[n][m]             # dt ,
         omega_comm.p[n][m]          # omega ,
         tolrsd_comm.p[n][m]         # tolrsd,
         nx0_comm.p[n][m]            # nx0 ,
         ny0_comm.p[n][m]            # ny0 ,
         nz0_comm.p[n][m]            # nz0 ,
         rsd0_comm.g1[n][m]          # rsd0 ,
         u1_comm.g1[n][m]            # u1 ,
         rsdnm_comm.p[n][m]          # rsdnm ,
         ssor_comm.bigLoop[n][m]     # rsd1 ,
         errnm_comm.p[n][m]          # errnm ,
         phis_comm.h0[n][m]          # phis ,
         frc1_comm.p[n][m]           # frc1 ,
         phiver_comm.h1[n][m]        # phiver,
         frc2_comm.p[n][m]           # frc2 ,
         phihor_comm.h2[n][m]        # phihor,
         frc3_comm.p[n][m]           # frc3 to lu[n][m] # ILU as LU_FM(LU_Params nprocs problem_size dt_default itmax, ...) → (...)
```

APÊNDICE C

O CÓDIGO # DE MCP-HASKELL#

C.1 CONFIGURAÇÃO

```
component MCP< n, m > with

interface IProbDef # () → (user_info, particles, tally_entries, recip, avg_e, all_tallies)
          behaving as: seq { recip!; avg_e!; all_tallies!; tally_entries!; user_info!;
                             repeat alt { particles! → skip }
                           }

interface ITracking (user_info,particles) → (events, totals)
          behaving as IPIPE particles → events
                     as: seq { user_info?;
                               repeat alt { particles? → events! };
                               totals!
                             }

interface ITallying (events, tally_entries) → tallies
          behaving as IPIPE events → tallies
                     as: seq { tally_entries?;
                               repeat alt { events? → tallies! };
                             }

interface IStatistics (avg_e, recip, totals, tallies) → ()
          behaving as IPIPE events → tallies
                     as: seq { avg_e?; recip?; all_tallies?;
                               repeat alt { tallies? → skip };
                               totals?
                             }

unit tracking_tallying # particles → tallies as PIPE_LINE<2>

unit prob_def  # IProbDef  () → (user_info, particles, tally_entries, recip, avg_e, all_tallies) as ProbDef
unit tracking  # ITacking  (user_info,particles) → (events, totals)                              as Tracking
unit tallying  # ITallying  (events, tally_entries) → tallies                                    as Tallying
unit statistics # IStatistics (avg_e, recip, all_tallies, totals, tallies) → ()                  as Statistics

assign tracking # particles → events to tracking_tallying.pipe[1]
assign tracking # events → tallies   to tracking_tallying.pipe[1]

connect prob_def→user_info       to tracking←user_info,          synchronous
connect prob_def→particles       to tracking_tallying←particles, synchronous
connect prob_def→tally_entries   to tallies←tally_entries,       synchronous
connect prob_def→recip           to statistics←recip,            buffered
connect prob_def→avg_e           to statistics←avg_e,            buffered
connect prob_def→all_tallies     to statistics←all_tallies,      buffered
connect tracking→totals          to statistics←totals,           buffered
connect tracking_tallying→tallies to statistics←tallies,         synchronous

replicate n tracking_tallying # particles → events
            connections particles<>: choice
                        events<>: choice

replicate m statistics # (avg_e, recip, totals, tallies) → ()
            connections avg_e<>: broadcast
                        recip<>: broadcast
                        totals<>: {# (map.sum.transpose) #}
                        tallies<>: concat
```

C.2 MÓDULOS FUNCIONAIS

C.2.1 ProbDef

```
module ProbDef(main) where

import Mcp_types
import Tracks
import Spec_file
import Misc
import Directions
import Tallies
import Show_funcs
```

```
import Array
Import List

main :: IO(Double, Double, User_spec_info, [Tally_entry], [(Particle,Seed)])
main = do
            putStrLn "Parsing input file (inp2)..."
            s ← readFile "inp2"
            putStrLn "inp2 parsed ! Calculating results ..."
            putStrLn
            s ← let
                    (user_spec, srce_info, all_tallies, seed, n) = user_specs s
                    (e_thresh, e_cut, w_cut, sfcs, celldefs, mats, zlist, sfc_cref, xsec_file) = user_spec
                in readFile xsec_file
            return (avg_e, recip, user_info, all_tallies, particle_list)
        where
            ((phot_data, incv, cohv),_):_ = xsecs_load s

            user_info = (e_thresh, e_cut, w_cut, sfcs, celldefs, mats, zlist, sfc_cref, phot_data, incv, cohv)

            particle_list :: [(Particle, Seed)]
            particle_list = map (sample_source srce_info) (take n (iterate trand_9973 seed))

            recip = 1.0/(fromIntegral n)

            avg_e = (foldl (+) 0.0 (map (\((_,_,_,e,_),_) → e) particle_list))*recip

– create a source photon by choosing energy and direction
sample_source :: Srce_spec_info → Seed → (Particle, Seed)
sample_source (pt, dir, kind, cell, distrib) s0 = (particle, sf)
            where
                – choose a direction
                s2 = trand_2 s0 (dircos, sf) = case kind of
                                                Iso → isotropic s2
                                                Uni → (dir, s2)

                – interpolate in the energy distribution array
                r0 = s_f s0
                r1 = s_f (trand_1 s0)
                e_bin = 1 + length (takeWhile (r0¿) (map snd (elems distrib)))
                (e1, _) = distrib!(e_bin-1)
                (e2, _) = distrib!e_bin
                e = r1*(e2 - e1) + e1
                particle = (pt, dircos, 1.0, e, cell)
```

C.2.2 Tracking

```
module Tracking(main) where

import Track import Tallies import Mcp_types

main :: User_spec_info → [(Particle,Seed)] → IO ([[Event]],[Int])
main user_info particle_list =
    let
        event_lists = map f particle_list
    in
        return (event_lists, tally_bal event_lists)
    where
        f (particle@(_,_,_, e, _), sd) = (Create_source e):(track user_info particle [] sd)
```

C.2.3 Tallying

```
module FM_Tallies(main) where

import Tallies import Mcp_types
import Array

main :: [Tally_entry] → [[Event]] → IO [[Array (Int,Int) Double]]
main all_tallies event_lists = return (map g event_lists)
    where
        g evs = map (tally_a_source evs) all_tallies
```

C.2.4 Statistics

```
module Statistics(main) where

import Mcp_types
import Tallies
import Show_funcs
import Array
import List

main :: Double → Double → [Tally_entry] → [Int] → [[Array (Int,Int) Double]] → IO ()
main avg_e recip my_tallies totals tallies =
```

```
    let
        tally_list :: [[Array (Int,Int) Double]]
        tally_list = transpose tallies

        accums :: [Array (Int,Int) Double]
        accums = map accum_tally tally_list

        squares :: [Array (Int,Int) Double]
        squares = map accum_tally_squares tally_list

        final_tallies = zip5 my_tallies accums squares (repeat recip) [1..]

    in do
          return (shows_results avg_e totals final_tallies )
          return ()

– accumulate a list of tally arrays into a single array
accum_tally :: [Array (Int,Int) Double] → (Array (Int,Int)Double)
accum_tally tally@(tt:_) =
            let
                bnds = bounds tt
            in
               accumArray (+) 0.0 bnds [((i,j),tly!(i,j)) — (i,j) ← range bnds, tly ← tally]

– accumulate a list of tally arrays into a single array of sum of squares
accum_tally_squares :: [Array (Int,Int) Double] →(Array (Int,Int) Double)
accum_tally_squares tally@(tt:_) =
                    let
                        bnds = bounds tt
                    in
                       accumArray (+) 0.0 bnds [((i,j),tly!(i,j)*tly!(i,j)) — (i,j) ← range bnds, tly ← tally]
```

APÊNDICE D

SINTAXE FORMAL ABSTRATA DA LINGUAGEM DE CONFIGURAÇÃO

<CONFIGURATION> ::= '**component**' <FORMAL_PARAMETERS> <INTERFACE_PORTS> <DECLARATION>$^+$

<FORMAL_PARAMETERS> ::= '<' <*formal_id*> (, <*formal_id*>)* '>'

<DECLARATION> ::= <USE_DECL> | <IMPORT_DECL> | <INDEX_DECL> |
<INTERFACE_DECL> | <UNIT_DECL> | <UNIFICATION_DECl> |
<FACTORIZATION_DECL> | <REPLICATION_DECL> | <ASSIGNMENT_DECL> |
<CHANNEL_DECL> | <BIND_DECL>

<USE_DECL> ::= '**use**' <*component_Id*>(.<*component_Id*>)*

<IMPORT_DECL> ::= '**import**' <*library_Id*>(.<*library_Id*>)*

<INDEX_DECL> ::= '**index**' <*index_id*>(,<*index_id*>)* '**range**' '[' <NUMERIC_EXP> '**..**' <NUMERIC_EXP> ']'

<INTERFACE_DECL> ::= '**interface**' <*interface_Id*> <INTERFACE_ESTRUCTURE> <SPECIALIZATION>

<INTERFACE_SPEC> ::= <INTERFACE_COMPOSITION> <INTERFACE_BEHAVIOUR>

<INTERFACE_COMPOSITION> ::= (<INTERFACE_PORTS> '**like**' <*interface_Id*>)*

<INTERFACE_PORTS> ::= # <*id*> | # <PORT_LIST> '→' <PORT_LIST>

<PORT_LIST> ::= '**(**' <*port_id*> ('**;**' <*port_id*>)$^+$ '**)**' | <*port_id*>

<INTERFACE_BEHAVIOR> ::= '**behaviour**' <SEMAPHORE_LIST>$^?$ <ACTION>

<SEMAPHORE_LIST> ::= '**sem**' '**(**'<*sem_id*> (, <*sem_id*>)* '**)**'

<ACTION> ::= '**seq**' {<ACTION> ('**;**' <ACTION>)$^+$}
| '**par**' {<ACTION> ('**;**' <ACTION>)$^+$}
| '**alt**' {<ACTION> ('**;**' <ACTION>)$^+$}
| '**repeat**' <ACTION> <REPEAT_TERMINATION>$^?$
| '**signal**' <*sem_id*>
| '**wait**' <*sem_id*>
| <*port_id*> '**?**'
| <*port_id*> '**!**'

<REPEAT_TERMINATION> ::= '**until**' <REPEAT_CONDITION> | '**counter**' <NUMERIC_EXP>

<REPEAT_CONDITION> ::= <CONJUNCTION> (| <CONJUNCTION>)*

<CONJUNCTION> ::= <MAYBE_SYNC_CONJUNCTION> (& <MAYBE_SYNC_CONJUNCTION>)*

<MAYBE_SYNC_CONJUNCTION> ::= <*port_id*> | '<' <*port_id*> (& <*port_id*>)$^+$ '>'

<SPECIALIZATION> ::= '**specializes**' <SPEC_INTERFACE_LIST>

<SPEC_INTERFACE_LIST> ::= '**(**' <SPEC_INTERFACE> ('**,**' <SPEC_INTERFACE>)* '**)**'

<SPEC_INTERFACE> ::= <*interface_Id*> <INTERFACE_PORTS>

<UNIT_DECL> ::= '**unit**' <UNIT_SPEC>

<UNIT_SPEC> ::= <*unit_id*> <INTERFACE_SPEC> '**groups**' <GROUP_LIST> <UNIT_COMPONENT>

<UNIT_COMPONENT> ::= '**as**' <*component_Id*> '**(**' <ACTUAL_PARAMETERS> '**)**'

<FORMAL_PARAMETERS> ::= '<' <EXPRESSION> (, <EXPRESSION>)* >

<UNIFICATION_DECL> ::= '**unify**' <OPERAND_UNIT>('**,**' <OPERAND_UNIT>)*
'**to**' <UNIT_SPEC>
'**connections**' <GROUP_LIST>

<FACTORIZATION_DECL> ::= '**factorize**' <OPERAND_UNIT>
'**to**' <UNIT_SPEC> ('**,**' <UNIT_SPEC>)*
'**connections**' <GROUP_LIST>

<OPERAND_UNIT> ::= <*unit_qual_id*> <INTERFACE_PORTS>

<REPLICATION_DECL> ::= '**replicate**' <NUMERIC_EXPR> <*unit_qual_id*> ('**,**' <*unit_qual_id*>)$^+$
'**connections**' <GROUP_LIST>

<ASSIGN_DECL> ::= '**assign**' <*unit_qual_id*> '**to**' <*unit_qual_id*> <INTERFACE_PORTS>

<GROUP_LIST> ::= (<*port_id*> <GROUP_SIZE>'**:**'<WIRE_FUNCTION>)$^+$

<GROUP_SIZE> ::= '<' <NUMERIC_EXP> '>'

<WIRE_FUNCTION> ::= '**choice**' <*wfunc_id*> | <*wfunc_id*>

<CHANNEL_DECL> ::= '**connect**' <OUTPUT_PORT_REF> '**to**' <INPUT_PORT_REF>

<OUTPUT_PORT_REF> ::= <UNIT_ID> '→' <*port_id*>

<INPUT_PORT_REF> ::= <UNIT_ID> '←' <*port_id*>

<UNIT_QUAL_ID> ::= <*unit_id*> ('**.**' <*unit_id*>)*

<BIND_DECL> ::= '**bind**' <PORT_REF> '**to**' <*id*>

<PORT_REF> ::= <OUTPUT_PORT_REF> | <INPUT_PORT_REF>

<NUMERIC_EXPR> ::= <NUMERIC_EXPR> <BYNARYOP> <NUMERIC_EXPR>
| <UNARYOP> <NUMERIC_EXPR>
| '**(**' <NUMERIC_EXPR> '**)**'
| <*integer_constant*>
| <*float_constant*>
| <*param_id*>
| <*index_id*>

<BYNARYOP> ::= '**+**' | '**-**' | '*****' | '**/**' | '**^**' | '**div**' | '**mod**' | '**#**' | '**log**'

<UNARYOP> ::= '**int**' | '**real**' | '**chr**' | '**ord**' | '**round**' | '**trunc**' | '**log2**' | '**ln**'

<*formal_id*> ::= <*lid*>
<*index_id*> ::= <*lid*>
<*port_id*> ::= <*lid*>
<*sem_id*> ::= <*lid*>
<*unit_id*> ::= <*lid*>
<*wfunc_id*> ::= <*lid*>
<*param_id*> ::= <*lid*>
<*component_Id*> ::= <*uid*>
<*library_Id*> ::= <*uid*>
<*interface_Id*> ::= <*uid*>

<*lid*> ::= <*lower_letter*> <*id_symbol*>$^+$
<*uid*> ::= <*upper_letter*> <*id_symbol*>$^+$

<*id_symbol*> ::= <*lower_letter*> | <*uper_letter*> | <*digit*> | '**_**' | ''

Este texto foi tipografado em LaTeX na classe UFPEThesis (www.cin.ufpe.br/~paguso/ufpethesis).
A fonte do corpo do texto é a Computer Modern Roman 12pt.

Tese de Doutorado apresentada por **Francisco Heron de Carvalho Junior** a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Programação Paralela de Alto Nível e Eficiente sobre Arquiteturas Distribuídas"** orientada pelo **Prof. Rafael Dueire Lins** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Nelson Souto Rosa
Departamento de Sistemas de Computação – CIn / UFPE

Prof. Paulo Romero Martins Maciel
Departamento de Sistemas de Computação – CIn / UFPE

Prof. Edil Severiano Tavares Fernandes
COPPE – Programa de Engenharia de Sistemas / UFRJ

Prof. Ricardo Massa Ferreira Lima
Escola Politécnica de Pernambuco - UPE

Prof. Siang Wun Song
Instituto de Matemática e Estatística / USP

Visto e permitida a impressão.
Recife, 9 de dezembro de 2003.

Prof. JAELSON FREIRE BRELAZ DE CASTRO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.