Universidade Federal do Ceará
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

Contratos Formais para Derivação e Verificação de Componentes Paralelos

Thiago Braga Marcilon

Fortaleza – Ceará
Setembro de 2012

Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

Contratos Formais para Derivação e Verificação de Componentes Paralelos

Autor
Thiago Braga Marcilon

Orientador
Prof. Dr. Francisco Heron de Carvalho Junior

Dissertação de mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal do Ceará como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Fortaleza – Ceará
Setembro de 2012

Resumo

A aplicação de nuvens computacionais para oferecer serviços de Computação de Alto Desempenho (CAD) é um assunto bastante discutido no meio acadêmico e industrial. Esta dissertação está inserida no contexto do projeto de uma nuvem computacional para o desenvolvimento e execução de aplicações de CAD baseadas em componentes paralelos, doravante denominada *nuvem de componentes*. Um dos principais desafios na sua utilização consiste no suporte à programação paralela, tarefa bastante suscetível à erros, pois tais erros podem levar, ao longo do desenvolvimento, a problemas de sincronização de processos, que podem causar abortamento da execução e a produção de dados incorretos, bem como a problemas relacionados ao uso ineficiente dos recursos computacionais. É importante que tais problemas sejam tratados no caso de aplicações de longa duração cujo respeito a um cronograma para obtenção de resultados é crítico, aplicações estas bastante comuns no contexto de CAD. Uma possível solução para tais problemas consiste na verificação do comportamento e das propriedades dos componentes na nuvem, antes que seja feita a sua execução, tornando possível que os usuários dos componentes da nuvem saibam se um componente pode ser utilizado com segurança em sua aplicação. Nesse cenário, o uso de métodos formais surge como uma alternativa atraente. A contribuição desta dissertação consiste em um processo de derivação e verificação de propriedades de componentes na nuvem. Tal processo envolve a especificação formal do comportamento dos componentes por meio de *contratos* descritos pela linguagem Circus. Então, através de um processo de refinamento e tradução tendo como ponto de partida o contrato, chega-se à implementação de um componente para execução sobre uma plataforma de computação paralela. Através desse processo, torna-se possível oferecer garantias aos desenvolvedores em relação ao comportamento dos componentes no contexto de suas aplicações. Para a prova de conceito, o processo é aplicado sobre a especificação "papel-e-caneta" de dois *benchmarks* do NAS Parallel Benchmarks, IS e CG, bastante difundidos na área de CAD.

Abstract

The use of cloud computing to offer High Performance Computing (HPC) services has been widely discussed in the academia and industry. In this respect, this dissertation is included in the context of designing a cloud computing platform for the development of component-based parallel computing applications, referred as *cloud of components*. Many important challenges about using the *cloud of components* relate to parallel programming, an error-prone task due to synchronization issues, which may lead to abortion and production of incorrect data during execution of applications, and the inefficient use of computational resources. These problems may be very relevant in the case of long running applications with tight timelines to obtain critical results, quite common in the context of HPC. One possible solution to these problems is the formal analysis of the behavior of the components of an application through the cloud services, before their execution. Thus, the users of the components may know if a component can be safely used in their application. In this scenario, formal methods becomes useful. In this dissertation, it is proposed a process for specification and derivation of parallel components implementation for the cloud of components. This process involves the formal specification of the components behavior through *contracts* described using the Circus formal specification language. Then, through a refinement and translation process, which takes the contract as a start point, one may produce an implementation of a component that may execute on a parallel computing platform. Through this process, it becomes possible to offer guarantees to developers about the components behavior in their applications. To validate the proposed idea, the process is applied to contracts that have been described based on two benchmarks belonging to the NAS Parallel Benchmarks, widely adopted in HPC for evaluate the performance of parallel programming and computing platforms.

Agradecimentos

Ao fim desta jornada, que dura 2 anos e meio, lembro-me e reflito a respeito de todas as dificuldades e obstáculos que apareceram pelo caminho e como sempre pude contar com a compreensão e suporte daqueles que me apoiaram. Tais pessoas foram essenciais na conclusão deste trabalho de mestrado, pois, sem elas, certamente eu haveria desistido ao longo do caminho.

Gostaria de agradecer primeiramente a Deus, pois, sem o seu contínuo cuidado e misericórdia, certamente eu não teria forças sequer para levantar da cama pela manhã. Também gostaria de agradecer à minha família como um todo, onde destaco a minha irmã, Lilian, e o meu pai, Nacélio. O carinho e suporte me dados pela minha família, muitas vezes, me deram forças e perseverança para continuar esta jornada. Agradeço também o meu orientador, Heron, cujos conselhos e orientação objetiva fez possível o desenvolvimento deste trabalho. Agradeço a paciência dos meus amigos Max Douglas, Leiliane e Edvânia, pelos convites negados e compromissos "furados" durante esse um ano e meio de convivência. Gostaria de agradecer também à minha namorada, Paloma, pela sua paciência, suporte e compreensão em relação as minhas ausências e aos meus momentos de estresse. E, por fim, agradeço ao CNPQ pelo apoio financeiro dado através da bolsa de estudo.

Sumário

Capítulo 1

Introdução

Nos últimos anos, o conceito de nuvem computacional tem sido bastante difundido na indústria e despertado o interesse acadêmico, sendo ainda objeto de estudo e debate acadêmico a viabilidade da sua utilização para o provimento de acesso a recursos de Computação de Alto Desempenho (CAD), notadamente plataformas de computação paralela. Tendo em vista esse desafio, os objetivos que serão anunciados nessa dissertação estão inseridos no contexto do projeto e implementação de uma nuvem computacional para o desenvolvimento e a execução paralela de aplicações de CAD baseadas em componentes. Esse é um projeto ainda em estágio embrionário dentro do grupo de pesquisa Paralelismo, Grafos e Otimização (ParGO), surgido a partir das experiências do grupo com o HPE (*Hash Programming Environment*) [38], uma plataforma de componentes paralelos voltada para plataformas de *cluster computing*. Por não termos ainda atribuído uma denominação a essa plataforma idealizada de nuvem computacional, a denominamos nesta dissertação como *nuvem de componentes*.

A *nuvem de componentes* destina-se a prover serviços para o desenvolvimento de componentes e aplicações capazes de executar em plataformas de computação paralela distribuídas, bem como serviços para implantação e execução desses componentes e aplicações sobre as plataformas as quais se destinam. Para isso, é aplicado o referencial teórico e conceitual do modelo de componentes Hash, o qual prevê o tratamento uniforme às abstrações envolvidas por meio dos componentes-# [35]. Assim, um conjunto de componentes, aplicações formadas a partir desses componentes, e plataformas de computação paralela estariam disponíveis como recursos acessíveis através dos serviços da *nuvem de componentes*, sob a abstração de componente-#, sendo os usuários responsáveis pela sua composição com o intuito

de tomar proveito dos recursos computacionais disponibilizados.

A nuvem de componentes destina-se a usuários de aplicações que demandam o poder computacional de plataformas de computação paralela para solucionar problemas de seu interesse. Em geral, são aplicações provenientes de domínios das ciências computacionais (*scientific computing*) e engenharias (*technical computing*), embora o interesse em áreas corporativas tenha emergido nos últimos anos. Através da *nuvem de componentes*, esses usuários podem controlar todo o ciclo de vida de componentes paralelos, desde o seu projeto e desenvolvimento até a sua implantação e execução nas plataformas de computação paralela participantes da nuvem, utilizando abstrações mais próximas dos seus domínios de interesse.

Dentre os desafios na utilização da *nuvem de componentes*, sabe-se que a programação paralela é uma tarefa difícil e bastante suscetível a erros devido à complexidade das interações entre os processos, as quais podem levar a problemas não-triviais de sincronização. Com isso, podem surgir problemas na execução de aplicações constituídas por componentes paralelos da *nuvem de componentes*. Dentre as consequências desses problemas, estão o abortamento da execução e a produção de dados incorretos. Tais consequências podem ser muito graves no caso de aplicações de longa duração, bastante comuns em CAD, as quais possuem requisitos críticos de tempo de terminação. Como exemplo desse tipo de aplicação, podemos citar a previsão climática e várias aplicações estratégicas de interesse industrial das quais o tempo de ciclo projeto e implementação de novos produtos são dependentes.

Nesse contexto, a possibilidade de verificação do comportamento de componentes na nuvem é uma funcionalidade desejável, pois é importante que os clientes dos componentes da nuvem saibam se um determinado componente pode ser utilizado com segurança no contexto de sua aplicação. Deve-se ressaltar que por tratar-se de um ambiente de nuvem, o uso de tais componentes pode ser gratuito ou pago. Nesse último caso, torna-se ainda mais crítica a possibilidade de oferecer garantias sobre o comportamento esperado do componente sobre uma determinada plataforma de computação paralela.

Assim, para resolver essa questão, propomos o uso de métodos formais para a descrição, análise e verificação do comportamento dos componentes na *nuvem de componentes*, através de um processo de desenvolvimento de componentes que envolve a especificação formal do comportamento de componentes por meio de *contratos*. Então, através de um processo de refinamento e tradução que tem como ponto de partida um contrato, chega-se à implementação concreta de um

componente para execução otimizada sobre uma certa plataforma de computação paralela. Entendemos que, através desse processo, seria possível oferecer garantias aos desenvolvedores de que o comportamento do componente obedece as restrições do contrato que ele se propôs a implementar.

Depois de compilado, o componente propriamente dito carregará consigo a sua especificação concreta, de modo que o seu comportamento e as suas propriedades possam ser verificadas por usuários que desejem fazer uso do componente ou pela própria nuvem, garantindo, assim, que o componente fará o que se propõe a fazer.

Na *nuvem de componentes*, esse ciclo de derivação e verificação é importante, pois aplicações CAD são tipicamente de longa duração. Nesse caso, a identificação e eliminação de erros na fase de projeto do componente pode oferecer considerável economia de tempo, dado que, quando não identificados, erros em tempo de execução obrigarão os desenvolvedores a ir em busca de soluções para erro nem sempre triviais, sendo necessário parar a execução do programa.

Este trabalho parte de ideias originalmente utilizadas na linguagem Haskell$_{\#}$ [33], que consistem na especificação do comportamento de componentes compostos usando expressões de comportamento. Tais expressões de comportamento podem ser traduzidas para Redes de Petri, tornando possível fazer análises diversas em relação ao comportamento dos componentes utilizando ferramentas de verificação para Redes de Petri. Uma vez que a nuvem de componentes não esteve funcionalmente disponível até a conclusão da dissertação, trabalhamos com a especificação e derivação formal de componentes-# sobre a plataforma HPE (*Hash Programming Environment*), porém já tendo em vista os requisitos do ambiente da nuvem computacional. Utilizamos uma extensão da linguagem de especificação formal Circus [128] para a especificação dos contratos de componentes, pois essa linguagem, além de permitir a separação da descrição comportamental e funcional de componentes do modelo Hash [37], ainda oferece a possibilidade do *refinamento* como técnica principal para derivação de especificações concretas, e eventualmente programas, a partir de especificações.

Enfim, esta dissertação une os benefícios da programação baseada em componentes, da especificação e derivação formal de programas e das plataformas de nuvens computacionais a fim de alavancar o desenvolvimento de aplicações típicas de Computação de Alto Desempenho.

1.1 Objetivos

1.1.1 Objetivo Geral

Desenvolvimento de um processo de certificação formal de componentes paralelos que atenda aos requisitos de uma plataforma de nuvem de componentes para especificação, construção e execução de aplicações de Computação de Alto Desempenho (CAD) sobre plataformas de computação paralela.

1.1.2 Objetivos Específicos

- analisar e caracterizar o contexto atual no uso de métodos formais, componentes e nuvens computacionais para aplicações de CAD;
- Investigar como pode ser gerado código fonte otimizado de acordo com as características de uma plataforma de computação paralela alvo, utilizando técnicas de programação comuns em CAD;
- Investigar a analogia do sistema de contratos de componentes com o sistema de tipos HTS (*Hash Type system*), suportado pela plataforma HPE, com a finalidade de propor um sistema de resolução de contratos para escolha de componentes de forma dinâmica e automática;
- Demonstrar a viabilidade da especificação e derivação por refinamentos sucessivos de códigos de programas paralelos no contexto de CAD;
- Definir uma extensão da linguagem Circus para especificação de componentes paralelos da plataforma HPE, baseada em HCL (Hash Configuration Language), a linguagem de descrição de arquitetura do HPE.

1.2 Estrutura do Documento

Os capítulos que compõem esta dissertação são descritos a seguir:

- O Capítulo 2 discute as áreas em que estão inseridas as contribuições esta dissertação: Computação de Alto Desempenho, desenvolvimento baseado em componentes, métodos formais e nuvens computacionais. Um especial interesse está na descrição das relações entre essas áreas.
- O Capítulo 3 apresenta os trabalhos precedentes a esta dissertação, detalhando os trabalhos com o Haskel$_{\#}$, o modelo de componentes Hash e a plataforma

de componentes paralelos HPE, oferecendo ainda uma descrição em alto nível da plataforma de nuvem de componentes idealizada com base na experiência com esses projetos de pesquisa anteriores.

- No Capítulo 4, descrevemos uma abordagem para o desenvolvimento de componentes dentro da nuvem de componentes idealizada por nosso grupo de pesquisa. Aqui nós propomos um processo a ser seguido, que utiliza especificação formal de componentes para dar garantias sobre o comportamentos dos componentes-#. É descrita a linguagem formal Circus, incluindo sua sintaxe, algumas noções sobre refinamento, e sua extensão para especificação de componentes-# proposta em trabalhos precedentes à este. Nesse capítulo, é tratado o uso da linguagem Circus como linguagem de especificação formal dos contratos dentro do contexto da metodologia de desenvolvimento, apresentando algumas ferramentas já implementadas para o Circus que podem ajudar na derivação e verificação de componentes na nuvem.

- No Capítulo 5, apresentamos o NPB, os seus conceitos e *benchmarks* e ainda os dois estudos de caso, que consistem em dois *benchmarks* integrantes do NPB com o qual pretendemos validar parcialmente a nossa proposta: IS (*Integer Sort*) e CG (*Conjugate Gradient*).

- O Capítulo 6 apresenta as conclusões desta dissertação, enfatizando a descrição de suas contribuições, as dificuldades encontradas na sua execução, as limitações do artefato proposto e propostas de trabalhos futuros decorrentes deste trabalho de pesquisa.

- Finalmente, o Apêndice A apresenta a demonstração de todas as obrigações de provas necessárias para as etapas de refinamento presentes nesta dissertação, utilizadas no Capítulo 5.

Capítulo 2

Contexto e Áreas de Interesse

A solução proposta nesta dissertação para derivação de componentes paralelos baseada em contratos formais baseia-se em contribuições provenientes de algumas áreas de interesse da ciência e da engenharia da computação, dentre as quais se destacam a *computação de alto desempenho*, a engenharia de software baseada em *componentes*, as *nuvens computacionais* e os *métodos formais* no desenvolvimento de software. Este capítulo apresenta uma visão geral dessas áreas, com o principal objetivo de motivar seu interesse no contexto deste trabalho e identificar suas relações entre si, como sintetizado na Figura 2.1.

2.1 Computação de Alto Desempenho

A Computação de Alto Desempenho (CAD) envolve a aplicação de tecnologias de computação para viabilizar a implementação de aplicações que requerem grande esforço computacional, em termos de quantidade de memória necessária e tempo de uso dos processadores disponíveis, para solução de problemas. Por esse motivo, tais aplicações não são viáveis de serem implantadas sobre plataformas de computação que não sejam especializadas em CAD ou usando técnicas e artefatos comuns de desenvolvimento de programas.

A computação paralela é principal técnica para alcançar os objetivos de CAD, envolvendo contribuições no projeto de arquiteturas de computadores, algoritmos e artefatos de programação tendo o processamento paralelo como objetivo principal a ser alcançado de forma eficiente. Dentre os artefatos de programação, incluem-se as linguagens de programação paralela, os compiladores capazes de paralelizar programas implicitamente e as bibliotecas de subrotinas, ou *middleware*, para suporte ao paralelismo [64].

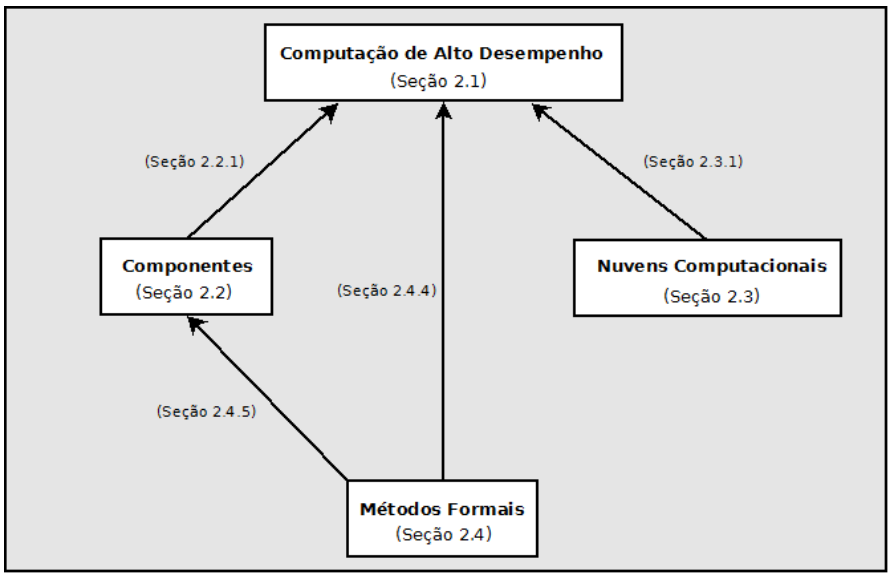


Figura 2.1: Contexto do Trabalho

As aplicações que apresentam requisitos de CAD são geralmente oriundas das ciências computacionais e engenharias, embora cada vez mais tenham surgido aplicações nos domínios corporativos (finanças e negócios). A seguir, uma lista não-exaustiva de exemplos dessas aplicações, que evidencia a sua importância para o desenvolvimento econômico e social da humanidade:

- Mineração de dados para descoberta conhecimento, em grandes bases de dados científicas ou de negócios, por meios computacionais;
- Previsão climática;
- Modelagem e simulação de bacias petrolíferas para aumentar a produtividade e segurança na extração de petróleo;
- Experimentos *in-silico* de interesse das ciências em geral;
- Simulação molecular para descoberta computacional de novos fármacos;
- Simulação da fisiologia de seres vivos, a nível celular e molecular, o que tornaria possível o estudo de novos tratamentos para doenças, inclusive congênitas, e melhoria da qualidade de vida das pessoas;
- Simulação de escoamento de fluidos para otimização aerodinâmica na indústria automobilística e aeroespacial;

- Previsão de catástrofes naturais (terremotos, tsunamis, etc) e simulação de seus efeitos sobre a infraestrutura urbana.

2.1.1 Plataformas de Computação Paralela

Como dito anteriormente, o processamento paralelo desempenha um papel primordial em CAD, sendo o requisito principal tratado na arquitetura das plataformas de computação voltadas às suas necessidades. Atualmente, as plataformas de computação paralela são classificadas em duas categorias, que as distinguem entre aquelas apropriadas para *capability computing* e aquelas apropriadas para *capacity computing* [1] [50].

Em *capability computing*, tradicionalmente estão incluídos computadores paralelos desenvolvidos para aplicações de um nicho específico ou mesmo uma aplicação específica, geralmente de tecnologia proprietária. Em geral, são os computadores paralelos que possuem a maior quantidade de unidades de processamento, totalmente dedicados a executar uma única carga de trabalho de interesse crítico para a instituição onde encontram-se instalados. Devido a essas características, são máquinas de custo de aquisição e manutenção bastante elevados, a despeito do alto grau de desempenho capazes de alcançar. Essa categoria inclui as plataformas atualmente conhecidas como MPP's (*Massive Parallel Processors*), o que poderíamos considerar como os supercomputadores da atualidade.

Em *capacity computing*, estão computadores paralelos de tecnologia aberta, em geral utilizando hardware de prateleira (processadores e interconexão), e orientados a propósitos gerais. Devido a isso, possuem excelente relação entre custo e benefício para a maior parte das cargas de trabalho típicas de aplicações de CAD de interesse tanto da industria quanto das instituições acadêmicas, resultando em um mercado bem mais abrangente do que o mercado para plataformas de *capability computing*. Essa categoria inclui os *clusters*, os quais são constituídos de um conjunto de nós computacionais homogêneos que se comunicam através de interconexões de prateleira. Pode-se imaginar um *cluster* como um conjunto de computadores *desktop* ou servidores convencionais trabalhando juntos como se fossem uma única máquina de grande porte. A Figura 2.2 denota um exemplo de *cluster* com 5 unidades de processamento e 2 unidades de gerenciamento.

Os *clusters* são atualmente distinguidos entre os de *Classe I* e os de *Classe II*. Os primeiros seguem mais fielmente a definição tradicional de *cluster*, originada do

[1] Usaremos o termo original em inglês tendo em vista a não existência tradução direta para o português ou termo disseminado pela comunidade.

projeto Beowulf em meados da década de 1990 [23], enquanto os últimos introduzem elementos normalmente encontrados em plataformas de *capability computing*, como alguns elementos constituintes proprietários que os permitem rivalizar com as próprias MPP's em seus nichos de aplicação.

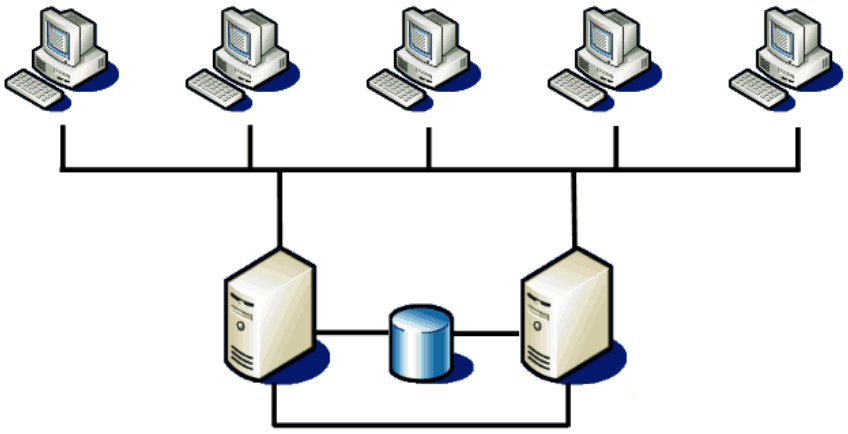


Figura 2.2: Representação de um *cluster*

O projeto Top500, mantido por instituições acadêmicas e parceiros industriais, mantém uma classificação dos computadores paralelos de maior desempenho, publicamente acessível através de um *website* [2]. Essa classificação oferece uma perspectiva muito fiel do estado-da-arte em termos arquiteturas de computação paralela. Na última lista, publicada em Junho de 2012, os *clusters*, a maioria dos quais de classe II, ocupam 82,2% das posições, enquanto as MPP's ocupam 17,4%. Em termos desempenho agregado, a participação dos *clusters* reduz-se para 67,1% do desempenho total, enquanto a participação das MPP's aumenta para 32,7%, o que é esperado devido ao seu maior desempenho e tecnologias especializadas. Aproximadamente 83% das máquinas possuem entre 4.000 e 8.000 processadores, enquanto há 10 máquinas com mais de 128.000 núcleos de processamento, ocupando as primeiras posições.

Nos últimos anos, além da consolidação e uso disseminado de processadores de múltiplos núcleos de processamento [80], tanto clusters quanto MPP's tem incorporado em seu projeto o uso de aceleradores computacionais, dentre os quais FPGA (*Field-Programmable Gateway Array*) [66], GPGPU (*General-Purpose Graphical Processing Units*) [55] e MIC (*Many Integrated Core*) [54], tornando a exploração do paralelismo nessas arquiteturas ainda mais desafiadora para

projetistas de linguagens e artefatos de programação, introduzindo novos níveis nas hierarquias de processamento e de memória. Em relação ao Top500, tais tecnologias tem causado um grande incremento no número de núcleos de processamento das plataformas que encontram-se nas primeiras posições.

Grades Computacionais Grades computacionais são infraestruturas que agregam recursos computacionais pertencentes à múltiplos domínios administrativos através da internet a fim de atingir um determinado objetivo. Dentre esses objetivos, grade computacionais podem ser empregadas com o propósito de oferecer um sistema de computação paralela distribuída, cujos nós de processamento são heterogêneos e geograficamente dispersos, o que resulta em severas limitações com respeito a intensidade e carga de comunicação possível entre os processos. Por esse motivo, grades computacionais são mais apropriadas para padrões de paralelismo como *bag-of-tasks* [44]. Embora uma grade possa ser projetada para uma aplicação em especial, o mais comum é o projeto de grades para os mais variados tipos de aplicações e interesses. Os recursos em uma grade são transparentes ao usuário, ou seja, o usuário não sabe onde geograficamente estão localizado os recursos que ele está utilizando, o que dá uma impressão ao usuário de que a grade é um só computador que dispõe de recursos virtualmente infinitos. Para que haja transparência em relação aos recursos, as grades computacionais fazem uso extensivo de *middlewares*. Costuma-se tratar *grid computing* como uma terceira classe de plataformas, alternativa a *capacity* e a *capability computing* [50].

Dentro do contexto apresentado, as plataformas de computação paralela modernas podem explorar o paralelismo em diversos níveis, formando hierarquias de paralelismo complexas, cujo tratamento pelos compiladores e artefatos tradicionais de programação tem sido considerado desafiador.

No nível de bits, o paralelismo pode ser alcançado aumentando-se o tamanho da palavra do processador. Por exemplo, um processador de 8-bits levaria 2 ciclos para fazer uma soma de 2 números de 16 bits: 1 ciclo para a soma da ordem baixa e 1 ciclo para a soma da ordem alta de cada número, enquanto um processador de 16 bits levaria apenas 1 ciclo, pois faria as duas somas simultaneamente.

No nível de instruções, o paralelismo pode ser também alcançado através de tecnologias superescalares aplicadas à arquitetura de processadores, as quais permitem a reordenação de instruções a fim de que *pipelines* de múltiplos estágios possam executar várias instruções simultaneamente. Tecnologias como VLIW (*Very Long Instruction Word*) também são aplicadas com a finalidade de fazer

com que um processador execute múltiplas instruções simultâneas, oferecendo uma maior janela de instruções para o processador, definida estaticamente pelo compilador, reduzindo a lógica de hardware necessária para a sua implementação. Os processadores vetoriais, ou SIMD (*Single Instruction Multiple Data*), também incorporam esse tipo de paralelismo, possuindo registradores vetoriais, ao invés de escalares, e instruções capazes de operar sobre vários elementos no mesmo registrador simultaneamente. Processadores atuais incorporam subconjuntos de instruções SIMD, usando tecnologias como SSE (*Streaming SIMD Extensions*), voltados a otimizar a execução de aplicações multimídia. O paralelismo SIMD também é característico de GPUs.

No nível de *threads*, o paralelismo também pode ser obtido com o uso de vários núcleos em uma mesma unidade de processamento central, onde cada núcleo é capaz de executar várias *threads* paralelamente [80].

No nível de multiprocessamento, são empregadas várias unidades de processamento em um mesmo computador, o que é conhecido como multiprocessamento. A diferença para o paralelismo a nível de *threads* é o fato de que *threads* compartilham dos recursos de uma mesmo unidade de processamento central como, por exemplo, o *cache*, enquanto processos em unidades diferentes não compartilham recursos. Porém, os modelos de programação são semelhantes, baseado em espaço compartilhado de variáveis.

No nível de aceleradores computacionais, tem sido empregadas tecnologias como GPGPU, FPGA e MIC, mencionadas anteriormente. Experimentos com aplicações reais tem demonstrado que o seu uso é capaz de oferecer incrementos de desempenho em ordens de magnitude, comparado ao uso unicamente de CPUs. Em participar, as GPUs tem desafiado os projetistas de artefatos de programação, ao introduzir várias hierarquias de paralelismo e memória que devem ser explicitamente controladas para obter o desempenho esperado (e.g. [120]).

No nível de processos em um mesmo domínio administrativo, como em *clusters* e MPPs, o paralelismo é obtido através de processos instanciados em computadores distintos, porém dentro de um mesmo domínio, os quais devem cooperar para atingir um objetivo, geralmente por passagem de mensagens.

No nível de processos em domínios administrativos distintos, os "processos" podem representar programas paralelos executando em *clusters* e/ou MPP's que fazem parte da constituição de uma grade computacional. Quando imprescindível, uma vez que a maioria das aplicações sobre grades computacionais

exploram padrões de paralelismo como *map-reduce* [47] e *bag-of-tasks* [44], a comunicação entre esses programas paralelos distribuídos emprega técnicas como invocações remotas paralelas de métodos (PRMI) [25] ou interfaces de comunicação $M \times N$ [4, 75, 133].

2.1.2 Programação Paralela

Diferente da programação sequencial, na programação paralela há a preocupação com o controle de várias linhas de instrução com o objetivo de executar uma computação de forma mais rápida, explorando o maior poder de processamento agregado de um conjunto de unidades de processamento. As linhas de instruções podem ser independentes ou não, distinguindo-se em assíncronas e síncronas, respectivamente. Neste último caso, destaca-se as que suportam o despacho de instruções do tipo SIMD (Single Instruction Multiple Data), característico dos tradicionais computadores paralelos vetoriais, e, mais recentemente, das GPUs.

Computadores vetoriais dependiam basicamente da tarefa dos compiladores, capazes de paralelizar laços automaticamente guiados por anotações de código inseridas pelo programador através de diretivas de compilação. por outro lado, GPUs exigem a programação paralela explícita, com menor auxílio do compilador, como no paralelismo assíncrono. No contexto geral, há basicamente dois modelos de programação paralela: baseado em espaço compartilhado de variáveis e baseado em memória distribuída [64].

O modelo de espaço compartilhado de variáveis é o modelo no qual todos os processos, ou *threads*, compartilham de um espaço de armazenamento comum (memória), através do qual podem se comunicar. Dependendo do local da memória onde um dado está armazenado, os processos podem levar o mesmo tempo para acessá-lo ou não, dependendo se a plataforma paralela utilizada é do tipo UMA (*Uniform Memory Access*) ou NUMA (*Non-Uniform Memory Access*).

O modelo de memória distribuída é o modelo onde cada processo tem sua própria memória, a qual os demais processos não tem acesso direto. Esse modelo exige o uso de bibliotecas de passagem de mensagens para que os processos possam se comunicar. As trocas de mensagens podem ser usadas para trocar dados, sincronizar processos ou ainda para a atribuição de trabalho. Existem principalmente dois padrões de interface para a passagem de mensagens, o *Message Passing Interface* (MPI) [116] e o *Parallel Virtual Machine* (PVM) [27]. Durante os anos 2000, o MPI consolidou-se como um padrão de fato para a programação paralela por passagem de mensagens.

Na programação paralela, parte-se do princípio de que os problemas para os quais busca-se uma solução computacional devem ser subdivididos em problemas menores, cada um dos quais pode ser resolvido independentemente. Em CAD, faz-se uso dessa técnica para obter ganhos em termos de velocidade de processamento em aplicações onde a divisão de tarefas é possível de ser realizada de forma a ocupar o potencial de desempenho do conjunto de unidades de processamento de uma plataforma de computação paralela. Porém, é muito comum que a tarefa de particionamento de um problema em subproblemas menores não seja uma tarefa fácil e resulte em um uso pouco eficiente dos recursos de processamento paralelo disponíveis. Além da execução paralela de tarefas, surgem também preocupações com a sincronização dos processos, o balanceamento de carga e o controle da localidade dos dados a fim de evitar sobrecargas de comunicação e no acesso à memória, preocupações que aumentam significativamente a dificuldade de desenvolvimento de programas eficientes, bem como a sua depuração e manutenção. Essas preocupações não existem no desenvolvimento de programas sequenciais, o que é motivação importante neste trabalho, que está particularmente interessado no tratamento da segurança de execução durante o desenvolvimento de programas paralelos.

2.2 Componentes de Software

O desenvolvimento baseado em componentes [127] é uma técnica de desenvolvimento de software que tem como premissa a separação de interesses no que diz respeito as funcionalidades de um determinado sistema, possuindo como principais características a reusabilidade e substituibilidade. Para isso, o software é constituído de um conjunto de partes independentes chamados de *componentes*, onde o critério de separação toma por referencial os interesses do software. Tais características aumentam a produtividade no desenvolvimento e na manutenção de sistemas, ditos baseados em componentes.

Não há uma definição consensual sobre o que seja um *componente*. Por esse motivo, será apresentada uma caracterização informal que atende aos propósitos desta dissertação. Um componente é uma parte funcionalmente independente e auto-implantável de um software, sujeita a composição por terceiros. A interação entre componentes se dá através de uma interface bem definida, previamente estabelecida. Componentes são definidos para oferecer um certo nível de abstração no que diz respeito ao serviço que prestam. No caso dos componentes "comerciais de prateleira" (*commercial off-the-shelf* - COTS), o engenheiro de software sabe

pouco ou nada sobre o funcionamento interno do componente. Então, ao engenheiro de software é dada apenas uma interface externa a partir da qual ele tem acesso às funcionalidades do componente. Sob essa perspectiva, os componentes são chamados *caixa-preta*, sobre os quais se conhece apenas o protocolo de interação com outros componentes mas não os detalhes internos do seu funcionamento.

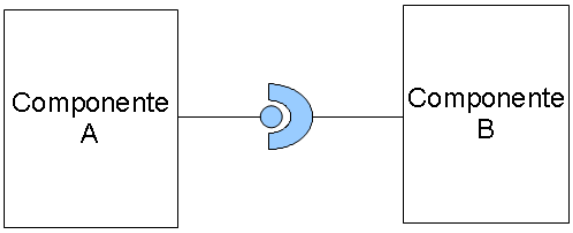


Figura 2.3: Composição de componentes

A Figura 2.3 ilustra a ligação entre dois componentes, A e B, um dos quais provê um serviço para o outro. O componente B atua como *usuário*, o que é indicado na figura pelo elemento em forma de "meia lua" ligado ao componente B. Por sua vez, o componente A atua como *provedor*, fornecendo exatamente as funcionalidades requeridas pelo componente B, denotado pela circunferência. Assim, pode-se realizar uma ligação entre os componentes A e B, de forma que que B possa utilizar as funcionalidades providas por A, por ele requeridas.

2.2.1 Desenvolvimento Baseado em Componentes em CAD

Devido aos seus requisitos peculiares, comparados aos requisitos de aplicações do domínio corporativo, aplicações CAD tradicionalmente não fazem uso das técnicas e artefatos avançados de engenharia de software, como linguagens funcionais e orientadas a objetos, desenvolvimento baseado em componentes, etc, as quais em geral preconizam o alto nível de abstração e portabilidade em detrimento a eficiência no uso dos recursos computacionais de processamento e memória de uma arquitetura específica. Entretanto, com o crescimento da complexidade dessas aplicações observado desde a última década, emergiram dificuldades na construção e manutenção de tais aplicações [107, 115], levando à busca por respostas dentre as tecnologias da engenharia de software moderna.

Nesse contexto, uma das alternativas investigadas tem sido o desenvolvimento

de modelos de componentes especialmente projetados para atender aos requisitos de aplicações de CAD [123], dentre os quais o CCA (*Common Component Architecture*) [14], o Fractal [28], o GCM (*Grid Component Model*) [21], e o Hash [35].

O modelo CCA foi desenvolvido por um consórcio envolvendo pesquisadores de laboratórios nacionais do Departamento de Energia (DoE) dos EUA e universidades, chamado *Center for Component Technology for Terascale Simulation Software* (CCTTSS). O objetivo principal desse consórcio era a definição de um conjunto mínimo de padrões de interface para que componentes pudessem se comunicar de forma eficiente e para que fosse possível construi-los através da composição de outros componentes. Outra preocupação fundamental era a "componentização" de código legado. As principais características do padrão CCA são:

- uso do padrão de projeto *provides/uses*, similar aos modelos comerciais CORBA e COM;
- especificação da interface dos componentes através da linguagem SIDL (*Scientific Interface Definition Language*), uma extensão da IDL de CORBA com estruturas de dados comuns em aplicações científicas, como matrizes multidimensionais e números complexos;
- delegação aos projetistas de *frameworks* computacionais compatíveis com o CCA a definição de formas de exploração do processamento paralelo.

Vários *frameworks* computacionais foram desenvolvidos para o suporte ao padrão CCA [123], alguns dos quais pré-existentes e que foram a ele adaptados, como o XCAT3 [77] e o SciRun2 [132]. Alguns desse *frameworks* possuem não apenas o propósito de atender requisitos de aplicações especificas, mas também de investigar a implementação eficiente de certos requisitos no padrão CCA, como o paralelismo e a distribuição. Por exemplo, o *framework* CCAffeine investigou o padrão de paralelismo SCMD (*Single Component Multiple Data*) para o processamento paralelo em clusters e multiprocessadores, além de ter servido a algumas aplicações [7]. Por sua vez, o DCA investigou a generalização do paralelismo SCMD para o caso onde componentes podem estar distribuídos, ocupando conjuntos diferentes de nós de processamento [26].

O modelo Fractal é um modelo extensível que pode ser usado com diversas linguagens de programação [6, 22]. O principal objetivo do modelo Fractal é reduzir os custos de desenvolvimento, implantação e e manutenção dos sistemas

de software. Se utiliza de padrões como a separação da interface da implementação. Concebido pelo consórcio OW2 e distribuído sob a licença LGPL, o modelo tem como características principais:

- Construção de componentes feita de forma hierárquica, ou seja, componentes podem ser combinados para formar componentes mais complexos;
- Uma determinada instância de componente pode ser usada ou compartilhada por mais de um componente. Essa característica é útil para modelar recursos compartilhados;
- Utiliza abstrações chamadas *bindings* para as conexões entre componentes, os quais podem encapsular semânticas complexas de comunicação entre componentes, desde chamadas síncronas de métodos até chamadas de procedimento remoto;
- especificação padrão organizada em níveis de conformidade, de acordo com o nível de autonomia e reconfiguração suportada para os componentes. Atualmente, há os padrões de conformidade 0, 1, 2 e 3, onde o nível 0 corresponde a componentes representados por objetos simples.

Julia e Cecilia são implementações de referência do modelo Fractal, respectivamente nas linguagens Java e C. Outras implementações consolidadas são o ProActive/Fractal, o AOKell (Java) e o Think (C). Há ainda implementações em caráter experimental, tais como FracNet (.NET), FracTalk (SmallTalk) e Julio (Python). Todas essas implementações são de código aberto e são acessíveis a partir da página do modelo Fractal na internet [6].

O modelo GCM foi idealizado para implantação sobre infraestruturas de grade computacional, tendo por base o modelo Fractal, do qual herda suas principais características. Foi desenvolvido no contexto do consórcio europeu *CoreGrid* [3], com o objetivo principal de definir um modelo de componentes para grades computacionais. Suas principais características são:

- Voltado a plataformas de grades computacionais, fornecendo um ambiente de alto nível para programação de aplicações para essa classe de plataformas.
- Extensão do modelo Fractal, herdando assim a característica de não impor nenhuma granularidade para o tamanho de um componente.

- Fornece interfaces para a definição de aspectos não-funcionais dos componentes.
- Fornece uma total transparência em relação aos recursos de uma grade computacional, que são gerenciados pelas próprias implementações do modelo, livrando, assim, o fardo do desenvolvedor.

Uma implementação do ProActive para o modelo GCM tem sido construída como implementação de referência [10].

O modelo Hash, cuja principal proposta é oferecer uma noção geral de *componente paralelo*, será tratado com maiores detalhes no Capítulo 3, devido a sua relação com as contribuições desta dissertação.

Os modelos de componentes CCA, Fractal, GCM e Hash visam aumentar a produtividade e a segurança no desenvolvimento de aplicações de CAD, propiciado pela reutilização de componentes implementados, o que reduz bastante o tempo de desenvolvimento de aplicações. Uma vez que componentes podem ser reutilizados, potencialmente seriam bastante utilizados e testados. Por esse motivo, a chance de haver erros de implementação em componentes reusáveis diminui a medida que tais componentes são reaproveitados, diminuindo assim as chances de erro nas aplicações como um todo.

Essa espécie de "validação por reputação" tem sido historicamente muito explorada no contexto da computação científica, notadamente sendo o motivo pelo qual bibliotecas de subrotinas científicas, muitas das quais com suporte ao paralelismo, são tão disseminadas dentre os cientistas e engenheiros quanto mais tradicionais forem [51]. Esse é um dos principais motivações que levam a acreditar que componentes são uma tecnologia promissora no contexto de aplicações nos domínios das ciências computacionais e das engenharias. Outra motivação importante é a possibilidade de que componentes implementem partes críticas de uma aplicação CAD especificamente para uma determinada arquitetura de computador paralelo, e que essa implementação, em geral representando alguma computação específica e de significado matemático bem-definido, possa ser disponibilizada para uso de terceiros. Por exemplo, isso poderia amortizar o esforço de implementação eficiente de uma computação sobre GPUs ou FPGAs.

2.3 Nuvens Computacionais

Nuvem computacional é o conceito por trás da ideia de oferecer recursos de computação como um serviço e não mais como um produto, possibilitado por um conjunto de tecnologias que surgiram durante os anos 2000. Nesse caso, o acesso aos recursos de hardware e software, bem como o acesso a informações em geral, por outros computadores ou dispositivos acontece através de uma rede, tipicamente a internet [12]. As nuvens computacionais possibilitariam o acesso a serviços através de uma interface *web*, independentemente da plataforma utilizada, da localização geográfica e do tempo. A disponibilização de serviços computacionais em uma nuvem pode ser comparada aos serviços de fornecimento de energia elétrica ou telefonia, onde o usuário faz uso do serviço sem o conhecimento de detalhes sobre a infraestrutura que o realiza e, se for o caso, paga de acordo com a sua utilização.

As nuvens podem ser classificadas de acordo com os serviços que prestam, como:

- **IaaS** (Infrastructure as a Service) - O serviço oferecido constitui em hardware virtualizado, blocos de armazenamento ou rede de comunicação. São serviços voltados ao armazenamento e processamento, sobre os quais o usuário pode instalar e manter suas aplicações. Sinônimo de **HaaS** (Hardware as a Service). Exemplo: Amazon EC2 e S3, DropBox;

- **SaaS** (Sofware as a Service) - O serviço oferecido é o software em si, representando uma aplicação final, podendo ser acessada através de *browsers*. Exemplo: Google Docs e Zoho;

- **PaaS** (Platform as a Service) - O serviço se constitui em uma plataforma computacional que oferece hospedagem, construção e implantação de aplicações sem o custo da compra e do gerenciamento de *software* e *hardware*. Exemplo: Google AppEngine, Microsoft Azure.

Nuvens computacionais não requerem do usuário o conhecimento sobre configurações dos recursos ou de onde está fisicamente localizado o sistema que os fornece de forma transparente e escalável, ou virtualizada. Enfim, a maior vantagem das nuvens é a possibilidade de se fazer uso de software, hardware ou ambos sem o custo de gerenciamento e sem a necessidade do conhecimento dos detalhes de infraestrutura necessária para a utilização do serviço requisitado. Outras vantagens que podemos citar são:

- A independência de plataformas e localidade permite que o usuário acesse sistemas usando um navegador *web* independentemente do dispositivo usado e permite que os usuários acessem de qualquer lugar, através da internet;
- A confiabilidade dos dados armazenados é garantida, considerando a aplicação de uma política de redundância de dados;
- O desempenho do sistema contratado pode ser monitorada através da web;
- A manutenção é mais fácil, considerando que o usuário não lida com o software ou hardware diretamente. Atualizações podem ser feitas de forma automática e sem a intervenção do usuário.

2.3.1 Nuvens Computacionais em CAD

As nuvens computacionais são entendidas como uma tecnologia complementar às grades computacionais. Enquanto as grades focalizam nos recursos de um sistema computacional, sendo seu interesse voltado a integração de recursos heterogêneos e geograficamente dispersos, as nuvens focalizam nos serviços oferecidos aos usuários, sendo seu interesse voltado a tornar transparente e simples o acesso a esses recursos. Em conjunto, tais tecnologias podem oferecer novas possibilidades para CAD, permitindo o acesso simples e homogêneo a um conjunto heterogêneo de plataformas de computação paralela e de armazenamento de dados.

A maioria dos trabalhos nessa área estão voltados para o fornecimento de plataformas de alto desempenho de forma transparente. Em [87] é apresentada uma arquitetura híbrida, onde são combinadas as vantagens do uso de nuvens computacionais, grades e arquiteturas centralizadas tradicionais de CAD, para se alcançar uma arquitetura que contenha as vantagens das 3 arquiteturas, como: alta escalabilidade de recursos, compartilhamento e heterogeneidade de recursos e alto poder de *capability computing*.

Em [53] é analisado a performance de aplicações de alto desempenho, se utilizando do MPI para comunicação, usando os recursos virtualizados de uma nuvem computacional IaaS. Em [124] são aplicados dois estudos de caso, que consistem em aplicações reais de alto desempenho, que são a Classificação de Dados de Expressões Gênica e Fluxo de Trabalho de Ressonância Magnética Funcional, para a avaliação da performance de tais aplicações no ambiente de uma nuvem computacional.

A esse contexto, acrescentamos os componentes, a fim de oferecer um arcabouço para definir as abstrações necessárias para oferecimento de aplicações e infraestrutura como recursos.

2.3.2 Componentes em Nuvens de CAD

No contexto de nuvens computacionais em CAD, foi identificado um único trabalho relacionado ao paradigma de desenvolvimento baseado em componentes, onde é discutido o uso de uma plataforma de componentes, baseado no modelo CCA, no desenvolvimento de aplicações em nuvens computacionais [86]. No capítulo 4, será discutido brevemente as diferenças entre esse trabalho e a abordagem de nuvem de componentes apresentada nesta dissertação.

2.4 Métodos Formais

Métodos formais são um conjunto de técnicas e linguagens baseadas em teorias e formalismos matemáticos para a especificação, desenvolvimento e verificação de sistemas de software e hardware, muitas vezes auxiliadas por ferramentas de software.

As ideias fundamentais de métodos formais são conhecidas e estudadas desde os anos 60. Porém, somente a partir dos anos 90 os métodos formais chegaram a um nível de maturidade suficiente no que diz respeito à sua integração às plataformas de desenvolvimento de software, despertando o interesse industrial. Atualmente, métodos formais são essenciais em aplicações reais em áreas como telecomunicações, energia, automobilística, robótica, cartões inteligentes (*smart cards*) e aviação [89].

Os métodos formais são uma ferramenta importante para oferecer garantias sobre a corretude de sistemas. São usados principalmente em sistemas de alto risco ou de tempo real, onde um erro de software pode custar dinheiro ou até mesmo vidas, como por exemplo um sistema bancário ou ainda um sistema controlador de voo. O uso de métodos formais requer maior perícia e um certo grau de esforço se comparado com os métodos comuns de desenvolvimento, porém como resultado temos um programa potencialmente livre de falhas.

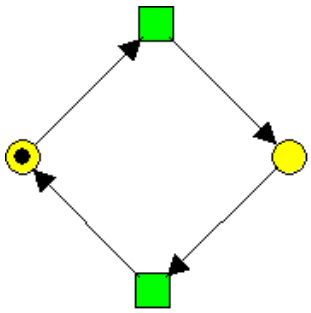
Um método formal é composto basicamente de uma linguagem de especificação formal, que é o modo pelo qual o modelo do sistema é representado, e um sistema de verificação, onde se pode fazer provas automáticas de propriedades. Alguns métodos, como a notação Z [74, 117], prezam por uma linguagem mais expressiva em detrimento da dificuldade de se desenvolver ferramentas de verificação.

No desenvolvimento de sistemas com métodos formais, as fases iniciais de

especificação e projeto do sistema são bastante custosas, porém tais custos são parcialmente compensados nas fases finais de testes e integração. A formalização do modelo pode indicar problemas sutis e sérios logo nas primeiras fases do ciclo de vida do desenvolvimento de sistemas. De outra forma, tais problemas seriam descobertos somente nos estágios de teste ou mesmo após a sua implantação.

De fato, métodos formais podem ser usados basicamente em dois níveis:

- **Especificação formal**: usada para representar um sistema de forma que suas propriedades formais sejam verificadas, antes que seja codificado em alguma linguagem de programação.
- **Derivação de programas**: Pode ser usada para produzir programas de forma automática ou semi-automática a partir de sua especificação formal.



$$SWITCH = on \rightarrow off \rightarrow SWITCH$$

Figura 2.4: Modelo de interruptor em redes de Petri e CSP respectivamente

A especificação formal é o processo de descrever, através de uma linguagem matemática com uma sintaxe e semântica definidas, as características e propriedades de um sistema, sejam elas funcionais ou não-funcionais. Atualmente, existem muitas linguagens de especificação formal, como por exemplo: CSP [67], redes de Petri [103], Z [117]. Cada linguagem modela um sistema de acordo com as suas características. CSP modela sistemas dando ênfase nas ações que os processos executam ao longo do tempo, definindo assim o comportamento dos processos através de sequências e ordens parciais de ações. Enquanto isso, Z modela sistemas dando ênfase nos estados pelo qual esse sistema pode passar e nas suas transições ao longo do tempo. Os estados são descritos através de estruturas matemáticas como conjuntos ou funções e as transições de um estado para o outro são definidas através das pré-condições e pós-condições. A Figura 2.4 mostra o modelo, em redes de Petri e CSP, de um interruptor cuja função é apenas ligar e desligar.

A utilidade da especificação formal, mesmo sem que haja a derivação de programas, vem do fato de que é através dela que são descobertos erros e inconsistências, além de se ganhar um maior entendimento das propriedades do sistema que se está especificando. Por ser desenvolvido por um linguagem com uma sintaxe e semântica bem definidas, temos um artefato que pode ser analisado de forma automática e, além disso, ainda pode ser usado como um contrato entre o cliente e o projetista do sistema.

Existem basicamente duas abordagens que podem ser utilizadas na verificação formal: a *verificação de modelos* e a *prova de teoremas*, os quais podem ser usados para analisar propriedades de segurança e de continuidade (*liveness*) de um sistema. Propriedades de segurança são aquelas que definem invariantes do sistema, ou seja, propriedades que garantem que um certo estado indesejável nunca será atingido ("nada de mal nunca irá acontecer"), como: "uma solução calculada por um certo algoritmo numérico satisfaz as restrições do problema". Já as propriedades de continuidade são propriedades que especificam que um estado desejável será atingido, como: "o algoritmo numérico certamente convergirá para uma solução correta".

2.4.1 Verificação de Modelos

A verificação de modelos [45] é uma técnica que consiste em enumerar o conjunto de estados possíveis de um modelo de um sistema e executar uma busca exaustiva nesse conjunto de estados a fim de provar uma propriedade desejada sobre o sistema. O modelo formal do sistema deve ser suficientemente abstrato de forma que o conjunto de estados seja finito. Assim, temos a garantia de que a verificação irá terminar. Ao contrário da prova de teoremas, a verificação de modelos é fácil de ser realizada e completamente automática, além de possibilitar a geração de contra-exemplos para auxiliar tarefas de depuração. A desvantagem principal dessa abordagem é o problema da explosão de estados, que ocorre em sistemas cujo modelo possui muitas variáveis que podem assumir muitos valores, resultando em uma enorme quantidade de estados que torna a busca inviável devido a quantidade de recursos computacionais necessários. Algumas estratégias podem ser usadas para amenizar o problema da explosão de estados, como a minimização semântica e a exploração da informação da ordem parcial. Como exemplos de verificadores de modelos, podemos citar o FDR2 [1], o SPIN [69] e o TLA+ [78].

2.4.2 Prova de Teoremas

A prova de teoremas [65] é uma abordagem analítica em que as propriedades e o sistema são expressos através de fórmulas em uma lógica definida por um sistema formal, as quais definem conjuntos de estados, possivelmente infinitos, que satisfazem uma certa propriedade. A prova de teoremas consiste em encontrar a prova de uma determinada propriedade, partindo dos axiomas e usando as regras de inferência que constituem a lógica. Tais provas podem ser realizadas com o auxílio de programas, chamados provadores automáticos de teoremas. Os provadores de teoremas podem ser completamente automáticos, e assim de uso para propósito geral, como o SNARK [118], ou interativos, usados para propósitos específicos, como o Isabelle/HOL [101]. A prova de teoremas recorre à indução estrutural para a prova de propriedades em um conjunto infinito de estados, ao contrário da verificação de modelos, o qual não oferece meios para lidar com tal situação. No entanto, os provadores de teorema interativos, por exigirem interação humana, exigem um maior tempo para provar propriedades, se comparados aos verificadores de modelos. Provadores de teoremas são essenciais quando a quantidade de estados possíveis do modelo formal é infinito.

2.4.3 Derivação de Programas por Refinamentos

O processo de refinamento consiste na transformação de uma especificação abstrata em uma especificação mais concreta. Tanto a especificação abstrata quanto a especificação concreta são modelos de um sistema, onde a especificação abstrata está mais perto da visão do usuário e a especificação concreta está mais perto de um programa executável. O processo de refinamento garante que a especificação concreta possui as mesmas propriedades da especificação abstrata, ou seja, tanto a especificação quanto a especificação concreta gerado a do processo de refinamento são visões de um mesmo sistema.

Os refinadores são programas que transformam a especificação abstrata de um sistema em código fonte através de vários passos, onde cada passo é a aplicação de uma ou mais *leis de refinamento*, definidas dentro de um sistema formal. Quando aplicadas, as leis de refinamento devem garantir a preservação do comportamento e das propriedades do modelo na especificação. Como exemplo de refinadores, podemos citar o CRefine [97] , que refina especificações na linguagem Circus.

2.4.4 Métodos Formais em CAD

No contexto da CAD, o uso de métodos formais para geração de código não é muito difundido. Os códigos característicos de CAD são, na sua maioria, artesanais e dependentes do sistema ou arquitetura no qual o programa irá executar, ao contrário dos códigos gerados por ferramentas de refinamento e tradução, os quais privilegiam o uso de abstrações de mais alto nível de linguagens de programação. Isso se deve ao caráter semi-automático do processo de refinamento e à necessidade de o código gerado atender as especificações originais.

Pelos motivos expostos, e dada a complexidade resultando da heterogeneidade de plataformas, como clusters, MPPs e GPUs, bem como as diversas técnicas de programação e bibliotecas de suporte ao paralelismo, como o MPI [52], openMP [98] e CUDA [55], as técnicas e ferramentas de verificação formal são mais discutidas e usadas no contexto de CAD.

Em aplicações reais, a programação visando alto desempenho e baixo uso de recursos é bastante susceptível a erros. Dessa forma, faz-se necessário o desenvolvimento de técnicas, como a verificação formal, para minimizar o erros que derivam do mal uso de técnicas e bibliotecas. Em [62], são discutidas formas de se melhorar e desenvolver a aplicação de verificação formal em aplicações reais de alto desempenho. Algumas delas são:

- O suporte de pesquisadores de métodos formais à API's utilizadas, tais como o MPI e o openMP;
- Projeto de implementação de APIs que suportam a verificação formal;
- Desenvolvimento de semânticas formais para a definição e diferenciação de programas sequenciais e paralelos;
- O ensino de ferramentas de métodos formais e suas aplicações em aplicações de alto desempenho.

De uma forma geral, aplicações no contexto de alto desempenho que são executadas um ambiente de memória distribuída são escritas usando MPI. O MPI é uma interface para comunicação por passagem de mensagens e é bastante utilizada em aplicações de alto desempenho pelo fato de que o seu projeto envolve um grande número de funções e pode ser usada em um grande número de plataformas. Por isso, se torna importante o desenvolvimento de técnicas de verificação formal que sejam

aplicadas em aplicações escritas usando o MPI. Atualmente, programas baseados em MPI são testados alimentando as funções implementadas com uma coleção de dados de entrada e analisando se algo acontece de errado, como uma saída errada ou anomalias de sincronização concorrente.

Em [102, 114, 126], são apresentadas abordagens para verificação de programas MPI usando verificadores de modelos projetados especificamente para programas MPI, como o MPI-SPIN [59] e o ISP [122]. A abordagem de verificação de modelos traz vantagens sobre a abordagem tradicional baseada em testes, pois estes não oferecem garantias da ausência de problemas em um programa. Isso é especialmente válido em programas concorrentes, onde a intercalação de instruções entre processos concorrentes pode gerar um quantidade intratável de histórias de execução. A verificação de modelos, por outro lado, pode garantir propriedades, que, pela abordagem tradicional, seriam impossíveis de serem garantidas, como a ausência de anomalias de sincronização concorrente.

2.4.5 Métodos Formais e Componentes

A descrição da arquitetura de sistemas de software tem um importante papel no desenvolvimento de sistemas complexos. Tais descrições podem ser feitas através de linguagens de especificação não formais, como a UML [110], ou linguagens de descrição de arquitetura (ADL), e tem por objetivo dar um maior entendimento do sistema e, possivelmente, auxiliar na implementação dos mesmos.

A maioria das descrições arquiteturais de sistemas são informais, como as construídas usando UML e, apesar de serem úteis no entendimento do sistema, têm o seu uso limitado pelo fato de que, como o seu significado não é preciso, determinar propriedades do sistema ou determinar se uma implementação é fiel ao seu projeto se tornam tarefas difíceis. Por outro lado, há uma vasta quantidade de trabalhos a respeito da verificação formal de propriedades de arquiteturas de software construídas a partir de linguagens de descrição de arquitetura, cuja semântica simples permite um maior grau de raciocínio sobre a estrutura de um software e a relação entre os seus componentes [40]. Muitos trabalham buscam ainda um maior grau de raciocínio com foco nas relações entre os componentes, através de conectores, ao invés de suas propriedades individuais [13, 57, 79, 113].

Em [8] é apresentado uma abordagem para descrever formalmente componentes de software e suas interações através da linguagens de descrição de arquitetura Wright [9] e da linguagem formal CSP, onde cada sistema é descrito em três

partes. Na primeira parte, são descritos os tipos de componentes e conectores. Na segunda parte, são definidos os componentes e conectores que farão parte do sistema, funcionando como instâncias dos tipos definidos na primeira parte. Finalmente, na terceira parte, é descrito como os componentes e conectores são combinados para formar o sistema.

Em [20], cada componente é caracterizado como um sistema dinâmico com uma interface pública, tendo seu estado encapsulado, onde componentes são tratados como coálgebras [111], ou seja, componentes podem ser vistos como funções $(U \times I) \rightarrow P(U \times O)$, onde U denota o espaço de estado interno, I denota o espaço de entradas e O o espaço saídas. Tal função denota como um componente, com um determinado espaço interno, reage a um estímulo de entrada de dados, produzindo uma saída de dados, e possivelmente mudando o seu estado interno. Ainda nessa abordagem, em [88], é apresentada uma base para se definir noções de refinamento de componentes, delineando-se a noção de substituição de componentes, ou seja, determinar quando é seguro substituir um componente por outro em um determinado contexto.

A seguir, será apresentado com mais detalhes o P-COM^2, que é um *framework* para desenvolvimento de programas paralelos, cuja ideia de especificação das iterações entre componentes se assemelha bastante ao do Haskell$_\#$ e do HPE, que serão discutidos no Capítulo 3, onde as iterações entre os componentes são descritas de forma exógena através de um linguagem própria com a possibilidade de verificação de propriedades através de ferramentas próprias de métodos formais.

P-COM^2

O P-COM^2 [84] é um *framework* para o desenvolvimento de programas paralelos, que se propõe a solucionar o problema da corretude de programas paralelos através da especificação das iterações de componentes usando uma *linguagem de especificação arquitetural*. As suas principais características são:

- Desenvolvimento dito evolucionário, onde uma instância de um programa evolui de um modelo de performance para uma aplicação;
- Composição automatizada de instâncias de programas, incluindo a geração de todas as estruturas paralelas;
- Adaptação de programas componente-a-componente em tempo de execução;

- Validação de pré e pós-condições e sequência de iterações em tempo de execução;
- Provas formais da corretude das iterações entre componentes baseado na técnica de verificação de modelos e de propriedades de sincronização do programa.

O P-COM^2 implementa a composição automatizada de componentes através de uma estratégia de descoberta e ligação dinâmica de componentes baseado em *interfaces associativas*. Dada a instância de um programa, o P-COM^2 busca o componente cuja a *interface de aceitação* (*accepts interface*) corresponde exatamente a *interface de requerimento* (*requires interface*) do componente que está sendo considerado em um determinado momento. Essa abordagem se assemelha à forma como o HPE faz a busca dos componentes aninhados, dada uma instanciação. Porém, o HPE generaliza tal abordagem, uma vez que caso não seja encontrado um componente cujo os parâmetros de contexto satisfaçam completamente os requisitados, o HPE generaliza o contexto, fazendo a busca em um contexto mais geral.

Assim como o Haskell$_\#$ e o HPE, o P-COM^2 especifica os seus componentes de forma exógena através da sua linguagem de especificação arquitetural. Porém, ao contrário da linguagem HCL, a linguagem proposta pelo P-COM^2 descreve pré e pós-condições e o comportamento sequencial das operações do componente de forma que tudo possa ser verificado em tempo de execução.

Além disso, a linguagem de especificação arquitetural do P-COM^2 tem uma semântica formal bem definida que pode ser traduzida para a linguagem formal CSP através de um tradutor. Assim, é possível fazer verificações de propriedades de comportamento das iterações entre os componentes estaticamente, com o auxílio de ferramentas de verificação de modelos para CSP, como o FDR. Nesse aspecto, o P-COM^2 tem bastante em comum com a ideia original do Haskell$_\#$, que é a especificação exógena das iterações entre os componentes e, com a tradução das suas respectivas linguagens para linguagens de especificação formal, a possibilidade de verificação de propriedades de tais iterações através de ferramentas como verificadores de modelos, provadores de teoremas e, no caso de existir um cálculo de refinamento para a linguagem formal, refinadores de especificação. Na abordagem desenvolvida nesta dissertação, deseja-se especificar não só as iterações, como também a computação feita pelos componentes, de modo que a verificação possa

ser feita a nível de computação.

2.4.6 Circus

Circus é uma linguagem de especificação formal que visa a formalização de um cálculo de refinamentos para programas concorrentes. Ele combina a linguagem de especificação Z, uma das mais utilizadas no meio acadêmico e industrial, com a álgebra de processos CSP (*Communicating Sequential Processes*), que tem como um de seus aspectos chave o fato de ter sido projetado em torno da noção de refinamento. Circus também incorpora a linguagem de comandos guardados de Dijsktra [48]. Assim, é possível escrever especificações, projetos e até mesmo programas utilizando Circus. Uma especificação em Circus é basicamente uma combinação de parágrafos Z, combinadas com comandos guardados de Dijkstra e combinadores de CSP.

A Notação Z

A notação Z, leia-se "zéd", é um conjunto de convenções escolhidas para tornar conveniente a modelagem de sistemas de computação usando linguagem matemática [74, 117]. Foi proposta em 1977 pelo grupo de pesquisa em programação (PRG[2]) da Universidade de Oxford.

Em Z, são modelados os estados de um sistema, representados pelo conjunto de suas variáveis, os valores que podem assumir e as operações que podem modificá-los. Um parágrafo Z consiste basicamente de duas partes. A primeira parte é a declarativa, onde é definido o nome do parágrafo, os parágrafos que serão incluídos, e as variáveis que serão usadas juntamente com os seus respectivos tipos. A segunda parte é a restritiva, onde são impostas restrições sobre as variáveis declaradas, através das invariantes, pré-condições e pós-condições, as quais estabelecem as propriedades que deverão ser satisfeitas.

(a) $MachineState \mathrel{\widehat=} [qtdDinheiro, qtdChocolate : \mathbb{N} \mid qtdChocolate \geq 0 \land qtdDinheiro \geq 0]$

(b) $InitMachine \mathrel{\widehat=} [MachineState' \mid qtdDinheiro' = 0 \land qtdChocolate' = 100]$

(c) $BuyChoc \mathrel{\widehat=} [\Delta MachineState \mid qtdChocolate > 0 \land qtdChocolate' = qtdChocolate - 1 \land qtdDinheiro' = qtdDinheiro + 25]$

Figura 2.5: Parágrafos na notação Z

[2]Sigla em inglês para *Programming Research Group*

Na Figura 2.5, que modela uma máquina de venda de chocolates simples, que recebe uma moeda de 25 centavos e devolve um chocolate, podendo armazenar no máximo 100 chocolates, podemos observar os principais elementos de um parágrafo Z. Em (a), primeiro é declarado o nome do parágrafo, que se chama *MachineState*. Então, temos a declaração das variáveis *qtdDinheiro*, que representa quanto dinheiro há na máquina em centavos, e *qtdChocolate*, que representa a quantidade de chocolate na máquina, ambas do tipo inteiro. Temos ainda duas restrições, as quais expressam que o valor das variáveis *qtdChocolate* e *qtdDinheiro* não podem ser menor que zero.

O parágrafo em (b), de nome *InitMachine*, especifica a inicialização do sistema, incluindo o parágrafo $MachineState'$, que representa o mesmo parágrafo *MachineState* porém com suas variáveis decoradas com um apóstrofo, o qual denota o novo valor de suas variáveis. Assim, é possível fazer o uso de suas variáveis ainda que herdando as suas restrições. Aqui não se faz a declaração explícita de nenhuma variável. Porém, na seção de restrições se faz uso das variáveis declaradas em $MachineState'$. Basicamente atribuímos o valor zero à variável $qtdDinheiro'$ e o valor 100 à variável $qtdChocolate'$.

Finalmente, em (c), temos a operação de compra de um chocolate, onde o cliente oferece 25 centavos à máquina e recebe um chocolate. O nome do parágrafo é *BuyChoc*, e inclui o parágrafo $\Delta MachineState$, denotando a inclusão dos parágrafos *MachineState* e $MachineState'$, e, assim, representando uma operação de mudança de estado. Aqui, não declaramos nenhuma variável adicional. Na seção de restrições, temos como única pré-condição que a variável *qtdChocolate* seja maior que zero. As pós-condições são que o novo valor da variável *qtdChocolate* seja igual ao valor antigo subtraído de 1 e o novo valor da variável *qtdDinheiro* seja o valor antigo somando a 25.

CSP (*Communication Sequential Processes*)

A álgebra de processos CSP é uma linguagem formal usada para descrever padrões de interação em sistemas concorrentes [67]. Ela foi criada por C. A. R. Hoare em 1978, como uma proposta de linguagem de programação concorrente. Porém, desde então passou por várias modificações, consolidando-se como álgebra de processos e tendo grande influência no projeto da linguagem de programação paralela OCCAM [72]. Em CSP, as ações do sistema são modeladas através de processos. Cada processo opera independente uns dos outros e se comunica com

outros processos através de canais de comunicação. Em CSP temos duas classes de primitivas:

- *eventos*, representando ações ou comunicações, os quais são indivisíveis e instantâneos;
- *processos primitivos*, representando processos de comportamento básico como o *STOP* (nada faz, representa o *deadlock*) e o *SKIP* (representa a terminação com sucesso)

Um processo em CSP pode combinar as primitivas acima por meio de diversos combinadores definidos pela linguagem, obtendo, assim, novos processos. Alguns dos combinadores principais são:

- Prefixo ($\rightarrow$): combina um evento e um processo para produzir um novo processo. Por exemplo, o processo $a \rightarrow P$ comunica o evento a com o ambiente e então se comporta como o processo P.
- Escolha determinística, ou externa ($\Box$): oferece ao ambiente a escolha entre dois processos. A escolha se dá quando o ambiente comunica a primeira ação de um dos processos. Por exemplo, o processo $(a \rightarrow P) \Box (b \rightarrow Q)$ oferece a escolha ao ambiente entre os processos $(a \rightarrow P)$ e $(b \rightarrow Q)$. Para que a escolha seja feita, basta que o ambiente comunique os eventos a ou b.
- Escolha não-determinística, ou interna ($\sqcap$): não oferece ao ambiente a escolha entre dois processos. A escolha é feita internamente ao processo, sem o conhecimento do ambiente. Por exemplo, o processo $(a \rightarrow P) \sqcap (b \rightarrow Q)$ pode se comportar como o processo $(a \rightarrow P)$ ou $(b \rightarrow Q)$. Caso o ambiente comunique a ou b ao processo, este pode entrar em *deadlock*, pois não se sabe que escolha o processo fez internamente.
- Intercalação ($|||$): combina dois processos em apenas um, que se comporta como uma intercalação arbitrária das ações dos dois processos.
- Interface Paralela ($\underset{a}{\|}$): combina dois processos em apenas um, que se comporta como uma intercalação arbitrária dos dois processo, mas com a restrição de que ele devem sincronizar no evento a. Por exemplo, o processo $(a \rightarrow P) \underset{a}{\|} (a \rightarrow Q)$ é equivalente ao evento $a \rightarrow (P \underset{a}{\|} Q)$ enquanto o evento $(a \rightarrow P) \underset{a,b}{\|} (b \rightarrow Q)$ irá entrar em *deadlock*.

- Composição Sequencial (;): combina dois processos em apenas um, que se comporta como o primeiro processo, e depois de acabado com sucesso o primeiro processo, ele se comporta como o segundo.

$$ChocMachine = coin25 \rightarrow choc \rightarrow ChocMachine$$

Figura 2.6: Modelo de máquina que vende chocolates

A Figura 2.6 apresenta uma modelagem simples em CSP para a máquina de chocolates especificada na Figura 2.5. Em CSP, note que o foco da modelagem é nas ações e nas comunicações do sistema concorrente, ao invés de seus estados. Primeiramente, é declarado o nome do processo, chamado *ChocMachine*, que representa o comportamento da máquina. Então, é descrito o seu comportamento. Primeiramente, ela aceita uma moeda de 25 centavos, representada pela ação *coin25*. Então, ela devolve um chocolate para o usuário, ato representado pela ação *choc*. Finalmente, ela volta a fazer tudo novamente, executando recursivamente a o processo *ChocMachine*. Perceba que tal máquina, por tratar-se de uma *computação reativa*, requer interação com o ambiente, ou seja, é preciso que a máquina interaja com um processo cliente a cada interação para o sistema produzir algum efeito visível, mas nunca atingindo um estado de terminação.

$$Client = coin25 \rightarrow waitChoc \rightarrow choc \rightarrow SKIP$$

Figura 2.7: Modelo de cliente para a máquina da Figura 2.6

A Figura 2.7 apresenta a descrição do comportamento de um possível cliente, o qual insere uma moeda de 25 centavos, ato representado pela ação *coin25*, espera até que a máquina lhe forneça o chocolate, ato representado pela ação *waitChoc* e, finalmente, recebe o chocolate adquirido, ato representado pela ação *choc*. Poderíamos combinar os dois processos, *ChocMachine* e *Client*, em um único, utilizando o combinador de interface paralela, $ChocMachine \underset{coin25,choc}{\parallel} Client$, o qual representa o processo resultante da interação entre o cliente e a máquina.

A Linguagem de Comandos Guardados de Dijkstra

A linguagem de comandados guardados (GCL[3]), criada por Edsger Dijkstra em 1975, consiste em um conjunto de comandos para representação de estruturas de

[3] *Guarded Command Language*.

repetição e de seleção que permitem a construção de programas não-determinísticos. A GCL é constituída principalmente de 5 construtores, descritos nos próximos parágrafos.

O *construtor de atribuição* é feito pelo operador :=, onde $a := b$ faz com que a variável a assuma o valor da variável b.

O *construtor de concatenação* é feita pelo operador ; onde o comando st_1; st_2 se comporta como o comando st_1 e depois como o comando st_2

O *construtor de seleção* tem a forma

if

$$
\begin{aligned}
& g_1 \to P_1 \\
& [] g_2 \to P_2 \\
& \vdots \\
& [] g_n \to P_n
\end{aligned}
$$

fi

A expressão g_i é uma expressão boleana e P_i representa uma ou mais instruções. Diz-se que g_i é a guarda de P_i, de onde se deriva o nome da linguagem. Se nenhuma guarda for verdadeira, então nenhuma instrução nessa construção será executada. Se apenas uma guarda for verdadeira, apenas as instruções correspondentes a esta guarda serão executados. Se mais de uma guarda for verdadeira, a escolha será feita de forma não-determinística. Abaixo é apresentado um exemplo dessa construção, em que, dados dois números arbitrários representados pelas variáveis x e y, armazena-se em m o maior dentre eles:

if

$$
\begin{aligned}
& x \geq y \to m := x \\
& y \geq x \to m := y
\end{aligned}
$$

fi

O *construtor de repetição* tem a forma

do

$$
\begin{aligned}
& g_1 \to P_1 \\
& [] g_2 \to P_2
\end{aligned}
$$

$$\vdots$$
$$[\!] g_n \rightarrow P_n$$

od

e age como uma construção de seleção repetidamente até que nenhuma guarda seja verdadeira, ou seja, enquanto houver uma ou mais guardas verdadeiras ela irá se comportar como um construtor de seleção indefinidamente. Quando nenhuma das expressões nas guardas avaliar para verdadeiro, o programa passa para a próxima instrução, após a ocorrência do construtor de repetição.

Como exemplo, tem-se o seguinte programa que calcula o máximo divisor comum (MDC) de dois números arbitrários representados pelas variáveis lógicas X e Y, cujos valores são maiores do que zero. O programa termina quando os valores das variáveis x e y são iguais, representando o MDC entre X e Y:

$x, y := X, Y;$
do

$$x > y \rightarrow x := x - y$$
$$[\!] y > x \rightarrow y := y - x$$

od

O *construtor de especificação* foi introduzido em 1988 por Carroll Morgan [91], sendo da forma $w : [pre, pos]$. Denota de forma abstrata um fragmento de programa em que, caso a pré-condição descrita por pre seja satisfeita, o programa executará e, quando terminar, deixará o programa em um estado em que a pós-condição descrita por pos será satisfeita, e somente os valores de variáveis que pertencem à w podem ter sido modificados. Por exemplo, a instrução de especificação a seguir faz a mesma tarefa do programa acima (cálculo do MDC):

$$x, y : \left[\begin{array}{c} (X > 0) \wedge (Y > 0), \\ (x \mid X) \wedge (x \mid Y) \wedge (x = y) \wedge (\forall\, q \in \mathbb{N} \bullet (q \mid X) \wedge (q \mid Y) \Rightarrow (x \geq q)) \end{array} \right]$$

A instrução de especificação denotada por $\{pre\}$ representa o mesmo que $: [pre, true]$, e a instrução de especificação denotada por $w : [pos]$ representa o mesmo que $w : [true, pos]$.

O *construtor de chamada de função* foi incorporado em 1987, por Back [16], o qual introduziu 3 tipos de passagem de parâmetros para funções:

- Por valor - a variável passada como parâmetro mantêm o seu valor inicial após o fim da execução da chamada da função e é indicado pela palavra **val**;
- Por resultado - a variável passada como parâmetro não tem o seu valor usado na função, porém tem o seu valor alterado, de forma que, após o término da função, o seu valor permanece alterado. É indicado pela palavra reservada **res**;
- Por referência - a variável passada como parâmetro tem o seu valor usado na função e tem o seu valor alterado, de forma que, após o término da função o seu valor permanece alterado. É indicado pela palavra reservada **var**.

A linguagem Circus utiliza essas definições para os parâmetros de funções. Porém, a palavra reservada **var** é substituída por **vres**, uma vez que **var** já denota a introdução de um bloco de variável.

A linguagem Circus incorpora todas as construções da GCL, exceto a construção de repetição, que pode ser facilmente substituída pelo operador de recursão do CSP (μ) ou o operador de iteração de sequência ($\fatsemi$).

Elementos de uma especificação Circus

Na Figura 2.8, é apresentada a gramática abstrata da linguagem de especificação Circus. Como pode-se observar, uma especificação Circus é formada por uma sequencia de parágrafos, onde cada parágrafo pode ser um parágrafo Z, uma definição de canal, uma definição de conjunto de canais ou ainda uma definição de processo.

Uma definição de canal representa a declaração de um único canal de comunicação, consistindo de um nome para o canal e o tipo dos elementos que serão transmitidos. Quando um canal for usado apenas para sincronização, ele terá apenas o seu nome. Um exemplo de declaração de canal capaz de transmitir valores no domínio dos inteiros é

channel write : $\mathbb{Z}$

Conjuntos de canais que foram definidos previamente, podem ser definidos com a construção **chanset**. Tal definição consiste em um nome para o conjunto e uma expressão que determina os seus membros. Um exemplo de declaração de um conjunto de canais, denominado *states*, é

channel $top, bottom, right, left$
chanset $states == \{\!| top, bottom, right, left |\!\}$

Program	::=	CircusPar*
CircusPar	::=	Par | **channel** CDecl | **chanset** N == CSExp |
	|	**process** N $\widehat{=}$ Proc
CDecl	::=	SimpleCDecl | SimpleCDecl; CDecl
SimpleCDecl	::=	N^+ | N^+ : Exp | Schema-Exp
Proc	::=	**begin** PPar* • Action **end** | N | Proc; Proc | Proc □ Proc
	|	Proc ⊓ Proc | Proc ⟦CSExp⟧ Proc | Proc ||| Proc | Proc \ CSExp
	|	Decl ⊙ Proc | Proc⟦Exp^+⟧ | Process[$N^+ := N^+$]
	|	Decl • Proc | Proc(Exp^+) | [N^+]Proc | Proc[Exp^+]
PPar	::=	Par | N $\widehat{=}$ Action
Action	::=	Schema-Exp | CSPAction | Command
CSPAction	::=	*Skip* | *Stop* | *Chaos* | Comm → Action | Pred & Action
	|	Action; Action | Action □ Action | Action ⊓ Action
	|	Action ⟦CSExp⟧ Action | Action ||| Action
	|	Action \ CSExp | μ N • Action | Decl • Action | Action(Exp^+)
Comm	::=	N CParameter*
CParameter	::=	? N | ? N : Predicate | ! Expression | . Expression
Command	::=	N^+ : [Pred, Pred] | $N^+ := Exp^+$
	|	**if** GActions **fi** | **var** Decl • Action | **con** Decl • Action
GActions	::=	Pred → Action | Pred → Action □ GActions

Figura 2.8: Gramática do Circus [43]

Uma definição de processo é formada por um nome e uma especificação. Uma especificação de processo define os seus estados e o seu comportamento. É formada por uma sequência de parágrafos, que podem ser parágrafos Z ou ações, e uma ação principal, delimitados pelas palavras reservadas **begin** e **end**. Um processo também pode ser uma combinação de dois ou mais processos através de combinadores CSP.

process $Tribonacci \mathrel{\widehat=}$ **begin**

$TribState \mathrel{\widehat=} [x, y, z : \mathbb{N}]$

$InitTribState \mathrel{\widehat=} [TribState' \mid x' = 0 \land y' = z' = 1]$
$InitTrib \mathrel{\widehat=} out!0 \rightarrow out!1 \rightarrow InitTribState$

$UpdTribState \mathrel{\widehat=} [\Delta TribState \mid y' = z \land x' = y \land z' = z+y+x]$
$OutTrib \mathrel{\widehat=} \mu X \bullet out!z \rightarrow UpdTribState;\ X$

• $InitTrib;\ OutTrib$

Figura 2.9: Processo gerador da sequência de Tribonacci

Na Figura 3.5, encontra-se definido o processo *Tribonacci*, o qual gera a sequência Tribonacci, uma variação da sequência de *Fibonacci*. A ação principal de um

processo indica o comportamento que o processo assumirá. Nesse exemplo, o processo *Tribonacci* se comportará primeiro como *InitTrib* e depois como *OutTrib*. A ação *InitTrib* envia pelo canal *out* os números 0 e 1, e então comporta-se como *InitTribState* para inicializar o estado do processo, atribuindo o valor 0 à x e à y e o valor 1 à z. A ação *OutTrib*, por sua vez, é definida recursivamente com o operador CSP μ. Ela envia pelo canal *out* o valor da variável z, e então comporta-se como *UpdateTribState* para atualizar o estado do processo, fazendo com que z seja igual ao seu valor anterior adicionado de x e y, y seja o valor anterior de z e x seja o valor anterior de y.

Refinamento no Circus

O processo de refinamento em especificações Circus consiste em remover gradualmente não-determinismos e incertezas, transformando assim uma especificação abstrata em uma especificação mais concreta, e, eventualmente, em código executável. Uma especificação abstrata pode não determinar totalmente algumas escolhas de projeto, e essa é uma das diferenças entre uma especificação abstrata e um código executável.

$$MovementAbs = (up \sqcap down) \rightarrow (left \sqcap right)$$
$$MovementCrt = (up \Box down) \rightarrow right$$

Figura 2.10: Processos CSP

Na Figura 2.10, temos dois processos CSP. O processo *MovementAbs* especifica uma ação de movimento, onde uma peça em um tabuleiro pode se mover primeiramente para cima ou para baixo e depois para a direita ou para a esquerda. No processo *MovementCrt* não há escolhas não-determinísticas. Primeiramente, oferece a escolha ao ambiente entre as ações de se mover para cima ou para baixo e depois ele se move para a direita. Todas as observações feitas em *MovementCrt* também são válidas em *MovementAbs*. Por esse motivo, em CSP, diz-se que *MovementCrt* refina *MovementAbs*, fato representado por $MovementAbs \sqsubseteq MovementCrt$.

A semântica do Circus é baseada nas *teorias unificadas da programação* (UTP[4]) [68], onde se faz uso de variáveis booleanas auxiliares para descrever as observações sobre o comportamento de um processo. Nas teorias unificadas, temos as variáveis *okay*, *wait*, *trace* e *refusal*, dentre outras, que são explicadas na Tabela 2.1, assim

[4]Da sigla em inglês para Unified Theories of Programming

Símbolo	Significado
$okay$	indica se processo que acabou de executar está em um estado estável ou divergiu
$okay'$	indica se processo que está executando está em um estado estável ou divergiu
$wait$	indica se o processo anterior já chegou no estado de terminação
$wait'$	indica se o processo que está executando já chegou no estado de terminação
tr	registra todos os eventos que ocorreram até a última observação
tr'	registra todos os eventos que ocorrerão até a próxima observação
ref	guarda o conjunto de todos os eventos que podiam ser recusados na última observação
ref'	guarda o conjunto de todos os eventos que podem ser recusados na próxima observação

Tabela 2.1: Descrição das variáveis da TUP

como as suas versões decoradas. Por exemplo, considere o processo $P \mathrel{\widehat{=}} (a \Box b) \rightarrow SKIP$. Ele poderia ser descrito por $(\neg\ okay \land tr \text{ prefix } tr') \lor (okay \land okay' \land ((wait \land wait' \land tr' = tr \land ref' = ref) \lor (\neg\ wait \land ((wait' \land tr' = tr \land a \notin ref' \land b \notin ref') \lor (\neg\ wait' \land (tr' = tr^\frown\langle a\rangle \lor tr' = tr^\frown\langle b\rangle)))))$. Assim, em Circus, se existem as ações A_1 e A_2 no mesmo espaço de estados, A_1 refina A_2 se, e somente se, as observações de A_2 são permitidas em A_1, ou seja, $A_1 \sqsubseteq_{\mathcal{A}} A_2$ sse $[A_2 \Rightarrow A_1]$, onde os colchetes representam o quantificador universal no alfabeto de A_1, que deve ser igual ao de A_2.

Sejam $P.st$ e $P.act$ o estado local e a ação principal do processo P, respectivamente. Em Circus, um processo P é refinado por outro processo Q se, e somente se, $(\exists\, P.st;\ P.st' \bullet P.act) \sqsubseteq (\exists\, Q.st;\ Q.st' \bullet Q.act)$ [43].

Em [60], descreve-se a implementação de uma ferramenta para tradução baseado em Circus, chamada JCircus, a qual traduz uma especificação concreta Circus para código na linguagem Java, utilizando a biblioteca JCSP para controle de concorrência.

Capítulo 3

Trabalhos Anteriores

Aplicações de Computação de Alto Desempenho caracterizam-se pela existência de computações que demandam um longo período de tempo para sua conclusão, tendo em vista exigirem uma enorme carga de processamento e/ou um grande número de operações de entrada e saída. Se tais computações retornam respostas erradas ou abortam no meio da execução, o trabalho executado nesse tempo seria desperdiçado, muitas vezes devido a erros de programação facilmente solucionáveis. Portanto, a especificação e análise do comportamento de componentes paralelos é uma tarefa importante na identificação de problemas potenciais durante a execução, bem como na busca prévia de soluções para esses problemas. Este capítulo insere o trabalho de pesquisa da qual trata esta dissertação no contexto dos trabalhos precedentes do grupo de pesquisa, os quais buscaram respostas a essas questões.

Inicialmente, é apresentada a linguagem Haskell$_{\#}$, uma extensão paralela da linguagem funcional Haskell. Haskell$_{\#}$ usa expressões de comportamento para a especificação do comportamento de seus componentes, as quais podem ser traduzidas para redes de Petri [104]. Dessa forma, pode-se modelar o comportamento de componentes paralelos e suas interações, de modo a oferecer garantias de propriedades de segurança e progresso durante a execução, tais como a ausência de *deadlock* e gargalos de sincronização.

Depois disso, será discutido o modelo de componentes Hash e sua implementação de referência, a plataforma de componentes paralelos HPE. O modelo Hash tem suas origens no modelo de coordenação de processos funcionais usado no Haskell$_{\#}$. Plataformas de componentes que seguem o modelo Hash, como o HPE, herdam do Haskell$_{\#}$ o uso de expressões de comportamento para especificar o comportamento dos componentes. No entanto, assim como no Haskell#,

expressões de comportamento não são suficientes para descrever a computação realizada pelos componentes. Em Haskell#, o uso de Haskell na programação dos processos funcionais, os elementos computacionais básicos de Haskell# (componentes primitivos), torna viável a especificação formal do comportamento computacional de componentes. No entanto, o HPE permite o uso de linguagens de diferentes paradigmas, como o imperativo, na qual o uso de métodos formais é considerado mais difícil. Por esse motivo, a *certificação formal* do contrato de componentes, objeto desta dissertação, não poderia ser feita à nível de computação em sistemas de programação do modelo Hash.

Finalmente, é descrita uma visão geral da nuvem de componentes, enfatizando os seus serviços e intervenientes, a qual trata-se de uma evolução desses trabalhos no que diz respeito à certificação formal do contrato de componentes, citando possibilidades para o desenvolvimento formal na nuvem e possíveis soluções para os problemas encontrados na abordagem para o HPE no que diz respeito à certificação de componentes #.

3.1 Haskell$_{\#}$

Haskell$_{\#}$ é uma extensão paralela para a linguagem Haskell, voltada para plataformas de computação paralela distribuídas [33]. Haskell é a principal representante dentre as linguagens funcionais puras e de semântica não-estrita [121]. Haskell$_{\#}$ pode ser entendida como uma linguagem para coordenação de processos funcionais escritos na linguagem Haskell, separando interesses relativos a descrição das computações (meio de computação) dos interesses relativos a coordenação dessas computações em uma execução paralela (meio de coordenação). Essa propriedade torna possível a implementação sobre qualquer compilador Haskell disponível, utilizando bibliotecas de passagens de mensagens do padrão MPI para comunicação e sincronização entre processos funcionais. Seus últimos protótipos foram implementados sobre o compilador GHC (Glasgow Haskell Compiler) [105].

A linguagem Haskell$_{\#}$ apresenta duas características principais, que são:

- Separação do código que descreve a coordenação do código que descreve a computação. A computação é descrita em código Haskell, encapsulado em processos funcionais, que são as unidades básicas para composição de programas. O nível de coordenação é descrito pela linguagem HCL (Haskell$_{\#}$ Configuration Language), que descreve como os módulos funcionais são coordenados para o funcionamento do programa a fim de implementar o seu

interesse.

- Configurações HCL podem ser traduzidas para redes de Petri [83], permitindo assim a análise formal de propriedades comportamentais, em nível de coordenação de processos funcionais, usando ferramentas de análise redes de Petri, como CPN [76], INA [109], TINA [24] e PEP [63].

Desde sua introdução, houve 3 versões da linguagem Haskell$_{\#}$, buscando atingir melhor conciliação entre expressividade na expressão de padrões de paralelismo e hierarquia de processos mais forte.

Primeira Versão (primitivas explícitas de passagem de mensagens): utilizava chamadas explícitas das primitivas de comunicação, como *send* (!) e *receive* (?=), o que permitia a mistura entre as primitivas de sincronização características do paralelismo com o código da computação em si, quebrando, assim, o princípio da hierarquia de processos, também tornando difícil a tradução de programas Haskell$_{\#}$ para redes de Petri [82].

Segunda Versão (semântica estrita de processos funcionais): aboliu as primitivas de passagem de mensagens explícitas. Porém, os processos funcionais tinham uma semântica estrita o que não permitia que processos funcionais intercalassem comunicação e computação. Isso dificultava a implementação de alguns padrões de interação entre processos conhecidos e bem difundidos [83].

Terceira Versão (*Lazy streams*): resolveu o problema de expressividade da segunda versão usando *Lazy streams*, as quais consistem de listas lazy cujos elementos são lidos ou escritos de acordo com a demanda [31]. Uma tradução para redes de Petri dessa versão também foi definida [29].

Um programa Haskell$_{\#}$ tem sua coordenação descrita pela linguagem de configuração HCL. Cada configuração descreve um conjunto de componentes, os quais pode ser simples ou compostos.

Componentes simples ou compostos são formados por um conjunto de unidades, cada qual possuindo um conjunto de portas de entrada e de saída associadas a um tipo. Tais portas são conectadas em uma configuração usando canais de comunicação síncronos, ponto-a-ponto e unidirecionais. Portanto, uma porta de entrada deve sempre ser ligada a uma porta de saída de tipo compatível. Os tipos são tipos da linguagem Haskell. Através das portas, as unidades de um componente comunicam-se para realização de uma tarefa paralela. Portas podem ser simples,

permitindo transmitir um único valor, ou *streams*, as quais permitem a transmissão de uma sequência de valores do tipo da porta.

Componentes simples, também chamados módulos funcionais, são formados por unidades associadas a uma função Haskell. Unidades de módulos funcionais são também chamadas de processos funcionais. O conjunto de processos funcionais de um programa Haskell$_{\#}$ constitui o seu meio de computação. As portas de entrada de um processo funcional são mapeadas aos argumentos da função Haskell, enquanto as portas de saída são mapeadas aos elementos da tupla retornada pela função. Portas *stream* são associadas a listas Haskell, as quais são listas *lazy* por natureza. Isso permite que processos funcionais comuniquem-se através de *lazy streams* [31].

Componentes compostos são formados por um conjunto de unidades compostas a partir de uma combinação de unidades de componentes aninhados, os quais representam *ações*. A combinação dessas ações é realizada usando uma expressão de comportamento [106] que se utiliza de combinadores de expressões regulares sincronizadas que possuem poder expressivo equivalente à redes de Petri [61, 73]. Expressões de comportamento são também chamadas de *protocolos* das unidades.

As *ações primitivas* de um protocolo são:

- $p?$ indica uma ativação de uma porta de entrada p, a fim de receber um item de dados;
- $p!$ indica uma ativação de uma porta de saída p, a fim de enviar um item de dados;
- wait *sem* representa a primitiva *wait*, ou P, no semáforo sem;
- signal *sem* representa a primitiva *signal*, ou V, no semáforo sem;
- skip representa uma ação sem efeito.

Por sua vez, os combinadores usados para construção de ações compostas, a partir de ações primitivas e outras ações compostas, são:

- seq especifica uma composição sequencial de ações;
- par especifica uma composição paralela de ações;
- alt representa uma escolha arbitrária entre um conjunto de ações;
- repeat especifica a execução repetida de uma ação.

As portas de uma unidade de componente composto são herdadas das portas das unidades que a compõem. Portanto, em tempo de execução, um programa Haskell$_{\#}$ é uma composição de processos funcionais conectados através de *lazy streams*.

Programas Haskell$_{\#}$ podem ser traduzidos automaticamente e diretamente para redes de Petri [41]. Com a separação da computação e coordenação no Haskell$_{\#}$, a tradução para redes de Petri é feita apenas a nível de coordenação. A prova de propriedades funcionais no nível de computação é possível, pois Haskell é uma linguagem funcional pura e não-estrita. As portas correspondem aos lugares e as operações de comunicação correspondem às transições em uma rede de Petri. Uma marca em um lugar significa que a porta correspondente está ativa. A ação de disparo corresponde à conclusão de uma comunicação e, quando isso acontece, a marca segue para o lugar correspondente a próxima porta ativa.

Tal característica da linguagem Haskell$_{\#}$, citada no parágrafo anterior, é bastante importante e útil porque as redes de Petri são um formalismo bastante utilizado na análise de diversas propriedades de concorrência e não-determinismo. Assim, podemos analisar diversas características das interações, tais como ausência de *deadlock*, e outras propriedades comportamentais de um programa Haskell$_{\#}$ através de ferramentas robustas e livremente disponíveis, como CPN [76], INA [109], TINA [24] e PEP [63].

Em [39] e [34], são apresentados estudos de caso de implementação de programas e avaliação de desempenho sobre Haskell$_{\#}$.

3.2 O Modelo de Componentes Hash

O modelo de componentes Hash [35], também nomeado pelo símbolo #, tem suas origens nos trabalhos realizados com a plataforma Haskell#. O modelo # foi definido a partir do modelo de coordenação de Haskell#, porém supondo que computações podem ser descritas em uma linguagem arbitrária, não necessariamente do paradigma funcional, suportada por um sistema de programação compatível com o modelo Hash. O resultado é um modelo para plataforma de componentes paralelos onde os componentes são intrinsecamente paralelos e possivelmente compostos hierarquicamente a partir de outros componentes ditos aninhados, o que o distingue dos outros modelos de componentes. Podemos afirmar que a linguagem Haskell$_{\#}$ é um sistema de programação compatível com o modelo Hash, onde a computação é descrita pela linguagem Haskell e a coordenação é descrita por expressões de comportamento especificadas com a linguagem HCL.

Cada componente-# deve implementar um interesse da aplicação, devendo sua granularidade ser definida pela plataforma de componentes em questão. Assim, o programador deve realizar a composição do programa paralelo a partir dos interesses de aplicação, ao invés dos processos. A composição a partir dos processos é comum nas práticas convencionais de desenvolvimento de programas paralelos. A composição a partir do interesses de aplicação oferecida por uma plataforma de componentes baseada em componentes # compatibiliza a programação paralela com as modernas práticas de engenharia de software [35].

Um componente-# é composto de um conjunto de partes chamadas *unidades*, assim como os componentes de Haskell#. Em tempo de execução, cada unidade deve executar em uma unidade de processamento distinto da plataforma de computação paralela de memória distribuída. Um componente pode ser formado pela composição hierárquica de um conjunto de componentes, ditos componentes aninhados, de tal forma que uma unidade é formada por um conjunto de fatias, onde cada fatia é uma unidade de um componente aninhado. De fato, cada unidade de componente aninhado deve ser fatia de alguma unidade do componente resultante da composição. Essa forma de compor componentes é chamada de *composição por sobreposição* [36].

A Figura 3.1(a) ilustra um exemplo abstrato de configuração de um componente-# de forma a melhorar a compreensão a respeito da composição por sobreposição. O componente **A** contém três unidades (a_1, a_3 e f_3), os quais representam seu papel em unidades de processamento de uma arquitetura distribuída, e três componentes diretamente aninhados (**B**, **C** e **D**), dos quais depende para realizar o seu interesse. As unidades a_1 e a_3 são formadas por fatias, definidas como unidades herdadas dos componentes aninhados **B**, **C** e **D**. A unidade f_3 é diretamente herdada de **B**. O componente aninhado **E** é compartilhado entre **B** e **C**, podendo representar, por exemplo, uma estrutura de dados que deve ser manipulada por computações representadas pelos componentes **B** e **C**. A Figura 3.1(b) ilustra a mesma configuração, porém usando a notação comum para dependências entre componentes hierarquicamente compostos. Os retângulos em cada componente representam suas unidades. Unidades com cores iguais em componentes diferentes estão executando sobre o mesmo nó de processamento, o que é mais claro de observar através da notação da Figura 3.1(a).

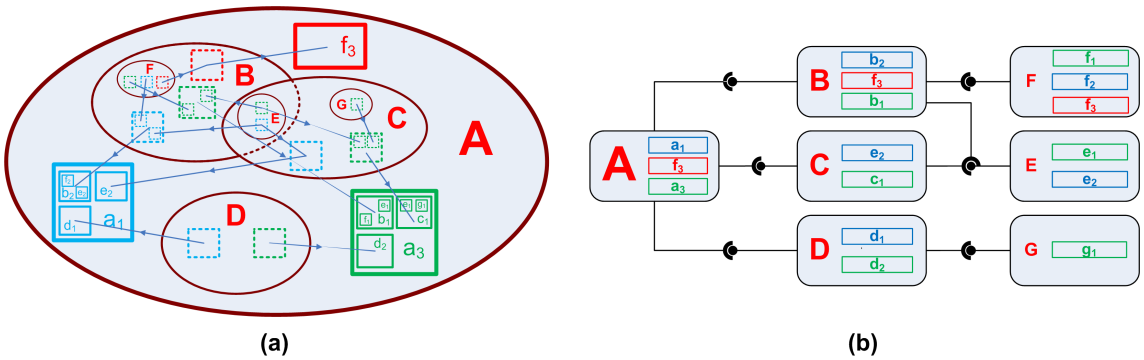


Figura 3.1: Componente # [38]

3.2.1 Sistemas de Programação Hash

A natureza concreta das unidades de um componente-# é definida por cada sistema de programação que implementa o modelo #, de acordo com um conjunto de *espécies de componentes* suportadas.

Definimos como um *sistema de programação* # qualquer plataforma para desenvolvimento de aplicações baseadas em componentes com as seguintes características:

- cada componente é formado por um conjunto de partes chamadas ***unidades***, onde cada unidade deve ser implantada em uma unidade de processamento da plataforma paralela;
- componentes podem ser combinados para formar novos componentes através da ***composição por sobreposição***;
- cada componente pertence à uma ***espécie de componente*** dentre um conjunto finito de espécies de componentes suportadas.

Espécies de componentes constituem um outro conceito que distingue o modelo # de outros modelos de componentes. Cada espécie de componentes inclui aqueles componentes # que possuem modelos de ciclo de vida, implantação e conexão comuns, sendo interpretados em termos dos mesmos artefatos de software, o que define a natureza concreta de suas unidades. No caso do primeiro sistema de programação #, o Haskell#, há um única espécie de componentes, representando os *módulos funcionais*.

Como comparação, plataformas de componentes usualmente definem uma única espécie de componentes e um conjunto de conectores pré-definidos. Sistemas de

programação # suportam várias espécies de componentes e conectores são tratados como componentes # de espécies específicas. Assim, desenvolvedores podem programar seus próprios conectores de acordo com as características da aplicação e da plataforma de execução. Espécies de componentes podem ainda ser usadas para promover a interoperabilidade entre diferentes sistemas de programação # ou com plataformas de componentes de outros modelos.

3.3 O HPE (*Hash Programming Environment*)

O HPE (*Hash Programming Environment*) é um sistema de programação # voltado a plataformas de *cluster computing* [32]. Trata-se de um ambiente de desenvolvimento a partir do qual todo o ciclo de vida de componentes # pode ser controlado, desde sua configuração até sua implantação e execução em um *cluster*. A arquitetura do HPE é composta de três componentes independentes:

- o Front-End, onde os desenvolvedores gerenciam as etapas ciclo de vida de componente-#, as quais incluem a sua configuração, registro em uma biblioteca, implantação em um *cluster* e execução. Atualmente, encontra-se implementado como um *plugin* para a plataforma de desenvolvimento Eclipse, utilizando o *framework* GEF (Graphical Editing Framework) [90] para implementação de um editor visual de configurações;

- o Core, o qual gerencia uma biblioteca de componentes #, distribuída em vários locais (*locations*);

- o Back-End, o qual lida com a plataforma de execução paralela sobre a qual os componentes # são implantados e executarão. De fato, trata-se de uma plataforma de componentes paralelos propriamente dita que segue o padrão CCA. Em outras palavras, define-se como um framework CCA com suporte a componentes paralelos distribuídos [30].

O HPE suporta atualmente as seguintes espécies de componentes, porém havendo a possibilidade de expansão para mais espécies:

- *Aplicações* denotam aplicações finais, formadas pela composição hierárquica de componentes das demais espécies, que podem ser executadas em um computador paralelo.

- *Computações* representam computações paralelas, onde cada unidade representa um processo que tem um papel na realização da computação.

- *Estruturas de Dados* representam estruturas de dados manipuladas por computações paralelas.
- *Comunicadores* representam interfaces de comunicação entre as unidades de componentes da espécie computação, encapsulando padrões de interação entre processos cuja lógica pode ser encapsulada em uma unidade de software.
- *Plataformas* representam as plataformas de computação paralela, cujas unidades representam suas unidades de processamento. Através delas, uma unidade de um componente das demais espécies pode acessar informações sobre a unidade de processamento física sobre a qual encontra-se localizada, de forma a mudar seu comportamento de acordo com suas características dinâmicas.
- *Topologias* representam topologias de processos sob as quais as unidades de um componente encontram-se organizadas, de forma que uma unidade pode conhecer a quais outras unidades ela encontra-se logicamente associada em uma computação ou padrão de comunicação. De acordo com a escolha de uma topologia, um comunicador apropriado pode ser escolhido para implementar o padrão de comunicador requerido em uma dada situação.
- *Qualificadores* denotam aspectos não-funcionais dos componentes das demais espécies.

Em um trabalho recente, produto da dissertação de mestrado de Cenez Araújo de Resende, o desempenho de aplicações sobre a plataforma HPE foi avaliado, usando-se as aplicações simuladas SP, BT, LU e FT do NAS Parallel Benchmarks (NPB) [19,49], as quais tiveram suas versões Java convertidas para C#, paralelizadas usando MPI.NET e refatoradas em componentes # [108]. O desempenho da versão baseada em componentes foi comparado ao desempenho da sua versão monolítica, evidenciando que a sobrecarga da refatoração em componentes sobre o desempenho de programas paralelos reais é pouco significativo, especialmente tendo em vista os benefícios advindos do uso de componentes, também bastante explorados nesse trabalho, uma vez que vários interesses antes espalhados no código das aplicações foram isolados em componentes.

3.3.1 O Sistema de Tipos HTS (*Hash Type System*)

Um componente abstrato descreve um contrato que deve ser atendido por componentes concretos que o implementam para um determinado *contexto de*

implementação. Atualmente, tal contrato consiste em um contrato sintático, ou seja, que consiste basicamente nas definições dos tipos que precisam ser respeitados pelos componentes concretos que implementam tal contrato. Portanto, os componentes concretos são os componentes # de fato, por serem aqueles que estarão de fato implantados e executarão na plataforma de execução gerenciada pelo *Back-End*.

Um componente abstrato define de forma completa o interesse a ser implementado por seus componentes concretos, embora não ainda de maneira formal, abstraindo-se de um *contexto de implementação* que deve ser determinado por cada componente concreto. Para isso, cada componente abstrato tem os seus *parâmetros de contexto formais*, cada um descrito por um *nome*, uma *variável* e um *tipo*. Por exemplo, seja um componente abstrato da espécie computação chamado SparseLinearSystemSolver com o parâmetro de contexto denominado *matrix_type* do tipo SparseMatrixType, cujo valor atual é atribuído a variável `MAT`. Isso é escrito da seguinte forma:

$$\text{SparseLinearSystemSolver}\,[\textit{matrix\_type} = \texttt{MAT} : \text{SparseMatrixType}]\,.$$

O componente abstrato SparseLinearSystemSolver representa o contrato de componentes # cuja função é resolver sistemas lineares esparsos usando algoritmos específicos conforme o tipo da matriz esparsa, o que é determinado pela escolha do parâmetro de contexto *matrix_type* associado a variável `MAT`, dentre subtipos do componente abstrato SparseMatrixType, o qual restringe as escolhas para o parâmetro de contexto *matrix_type*. Por exemplo, poderíamos supôr a existência dos componentes abstratos BlockTridiagonalMatrixType, PentadiagonalMatrixType e RandomSparseMatrixType, respectivamente representando matrizes esparsas bloco-tridiagonais, pentadiagonais e cujos elementos não-nulos encontram-se uniformemente e aleatoriamente distribuídos na matriz.

Componentes concretos implementam *instanciações*. Uma instanciação é definida como um par composto por um componente abstrato e um conjunto de parâmetros de contexto atuais compatíveis com seus parâmetros de contexto formais. A seguir alguns exemplos possíveis de instanciações envolvendo o componente abstrato SparseLinearSystemSolver:

- SparseLinearSystemSolver [*matrix_type* = BlockTridiagnoalMatrixType];
- SparseLinearSystemSolver [*matrix_type* = PentadiagonalMatrixType];

- SparseLinearSystemSolver [*matrix_type* = RandomSparseMatrixType].

Instanciações definem suposições específicas que um componente concreto faz para implementar o interesse do componente abstrato de forma otimizada ou mais adequada para o contexto. Portanto, componentes concretos que implementam as instanciações acima implementam algoritmos específicos para resolver sistemas lineares esparsos cujos padrões de distribuição dos elementos não-nulos na matriz obedecem as restrições impostas pelas escolhas de subtipos de SparseMatrixType para suprir o parâmetro de contexto *matrix_type*.

Instanciações são também os tipos associados aos componentes aninhados de um componente concreto. Durante a execução, quando um componente-# é carregado seus componentes aninhados devem também ser carregados a partir de suas respectivas instanciações. Para isso, é empregado um algoritmo de resolução capaz de achar o "melhor" componente concreto que implementa uma instanciação. Por "melhor", entende-se o componente concreto cuja instanciação é um subtipo mais próximo da instanciação requisitada, representando um conjunto de suposições de implementação menos gerais em relação as suposições especificadas pela instanciação requisitada. O cálculo realizado pelo algoritmo de resolução percorre os subtipos da instanciação requisitada em uma ordem predeterminada, generalizando os parâmetros de contexto atuais sempre que um componente concreto para a instanciação corrente não é encontrado no ambiente de componentes concretos implantados. Dessa forma, se um componente concreto possui um componente aninhado do tipo SparseLinearSystemSolver [*matrix_type* = BlockTridiagnoalMatrixType], o algoritmo de resolução é capaz de achar um componente concreto que implementa essa instanciação, ou generalizá-la a fim de obter um componente menos específico. Por exemplo, poderia ser buscado um componente concreto que implementa SparseLinearSystemSolver [*matrix_type* = SparseMatrixType], ou seja, que usaria um algoritmo de solução ideal quando não é conhecido o padrão de distribuição dos não-nulos na matriz. Caso nenhum componente concreto seja compatível, um erro de ligação ocorre em tempo de execução.

3.3.2 *Hash Configuration Language* (HCL)

A configuração dos componentes no HPE é descrita através de uma linguagem de descrição de arquitetura chamada Hash Configuration Language (HCL)[1]. A configuração de um componente-# é uma especificação onde são descritos todos os aspectos de um

[1] A sigla comum a Haskell# Configuration Language não é acaso. A Hash Configuration Language foi de fato inspirada nessa linguagem.

componente-#, tais como a sua espécie, as suas unidades, os seus componentes aninhados e os seus parâmetros de contexto. Através de configurações HCL, podemos descrever *componentes abstratos*, os quais são implementados concretamente, em *componentes concretos*, através de uma linguagem hospedeira, com o auxílio do *Front-End*. Atualmente, é suportada a linguagem C# como hospedeira, uma vez que o *Back-End* encontra-se implementado sobre a plataforma Mono.

```
computation MatVecProduct(a, x, v)
        [platform = C : Platform, topology = Top : CartesianTopology]
         partition_strategy_a = Da : MatPartition[topology = Top],
         partition_strategy_x = Dx : VecPartition[topology = Top],
         partition_strategy_v = Dv : VecPartition[topology = Top]]
begin
   data a : ParallelMatrix[platform = C, partition = Da]
   data x : ParallelVector[platform = C, partition = Dx]
   data v : ParallelVector[platform = C, partition = Dv]
   topology t : Top

   parallel unit calculate
   begin
      slice a.matrix
      slice x.vector
      slice v.vector
      slice t.unit
   end
end
```

Figura 3.2: Exemplos de configurações de componentes em HCL

A Figura 3.2 apresenta um exemplo de configuração de um componente abstrato escrito em HCL, os quais representam a multiplicação de uma matriz por um vetor. O cabeçalho da configuração declara a espécie do componente (*Computação*), o seu nome (MatVecProduct), os componentes aninhados públicos, e os seus parâmetros de contexto. No corpo da configuração, são especificados os três componentes aninhados públicos, de nomes a, x e v, representando os operandos da operação, e o componente aninhado (privado) t, o qual representa a topologia em que se encontram organizadas as unidades, a qual afetará como os vetores e matrizes encontram-se distribuídos entre as unidades e qual o papel de cada uma delas na computação.

Os componentes aninhados a, x e v são da espécie *Estrutura de Dados*, e tipados pelos componentes abstratos ParallelMatrix e ParallelData, com o seus parâmetros devidamente supridos. O componente aninhado t é tipado com a variável *Top*, podendo ser associado, na instanciação do componente abstrato, a qualquer subtipo do componente

abstrato CartesianTopology, o qual representa topologias cartesianas. Note que os tipos das variáveis de contexto *Da*, *Dx* e *Dv* também são restringidos pela escolha da topologia, uma vez que a forma de particionamento da matrizes e vetores depende de como os processos estão logicamente dispostos.

Um componente MatVecProduct possui uma única unidade paralela, chamada *calculate*, representando portanto uma computação do tipo SPMD (*Single Program Multiple Data*). Cada unidade contém quatro fatias chamadas *aslice*, *xslice*, *vslice* e *tslice*, de tal forma que cada fatia provém de uma unidade de um componente aninhado. Através desta última, uma unidade pode conhecer a posição relativa de uma unidade em relação às demais. Por exemplo, caso o valor atribuído ao parâmetro de contexto *topology* em uma instanciação seja LinearTopology, será necessário apenas o acesso a uma propriedade que defina o ranque do processo representado por um único valor inteiro entre 0 e $N-1$, onde N é o número de unidades instanciadas na execução, como em MPI.

No HPE, a coordenação dos componentes é descrita da mesma forma que em Haskell$_{\#}$, usando expressões de comportamento. Já a parte computacional é descrita usando a linguagem hospedeira. Através de uma linguagem de especificação formal, deseja-se descrever tanto a parte da coordenação quanto a parte computacional do componente, tornando possível gerar o código fonte do componente. Especificar o comportamento de um componente-# é útil, dentre outras coisas, para a garantia da ausência de falhas na composição de componentes e, uma vez que se prove uma propriedade sobre um componente, podemos usar tal propriedade para provar outras propriedades sobre outros componentes que sejam compostos por esse componente. Mesmo com as expressões de comportamento, o HPE ainda não realiza a verificação de comportamento dos componentes, e não leva isso em consideração na escolha dos componentes que irá usar na execução de uma aplicação.

Para especificar formalmente um componente-#, é necessária uma linguagem de especificação formal que capture não apenas as mudanças de estados de um componente, mas também as ações que o componente executa ao longo do tempo. A linguagem Z é uma linguagem de especificação formal largamente utilizada tanto na indústria como na academia, e fundamenta-se na descrição dos estados de um sistema. Já a linguagem de especificação formal CSP, bastante utilizada na descrição de sistemas paralelos e concorrentes, se preocupa em descrever as ações de um determinado sistema. Vários esforços se propõem a juntar as características das linguagens Z e CSP em uma única linguagem, como por exemplo, as linguagens CSP-Z [58], TCOZ [85] e Circus [129, 130].

A linguagem Circus foi escolhida frente aos requisitos apresentados no contexto deste trabalho, uma vez que, além de oferecer suporte à especificação comportamental e funcional de um componente-#, está associada a um cálculo de refinamentos que permite a derivação

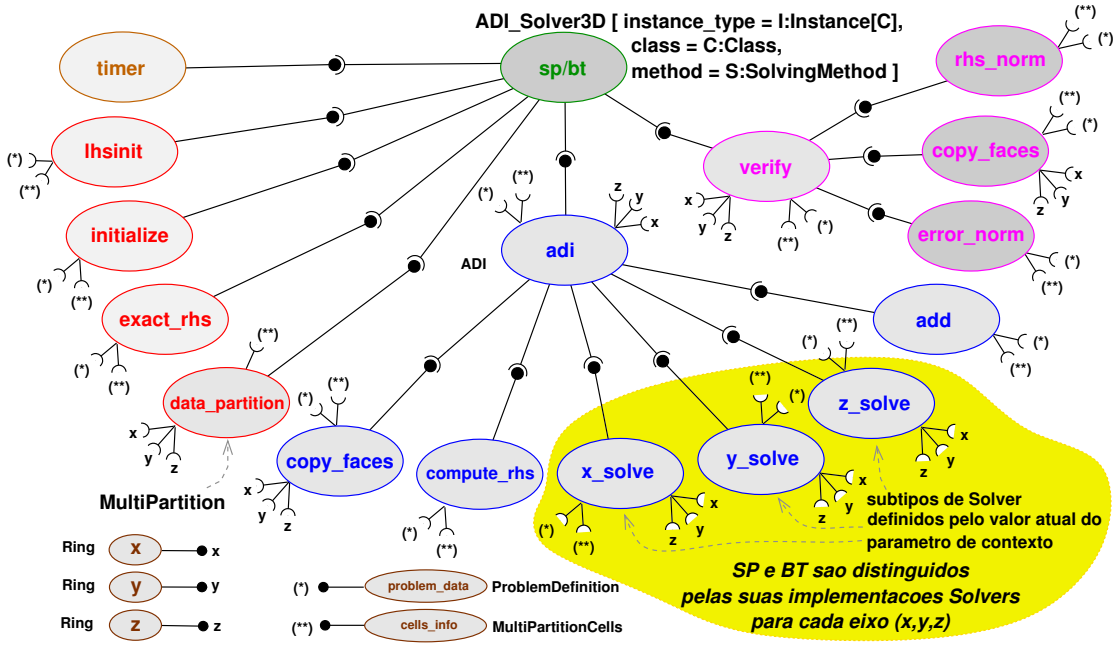


Figura 3.3: Arquitetura dos componentes SP e BT [108]

de código de programas a partir de suas especificações [97, 112]. Em [37] é descrita uma extensão sintática da linguagem Circus para especificar formalmente componentes-#, a qual será detalhada no Capítulo 4.

3.3.3 Exemplo de Aplicação

O *NAS Parallel Benchmarks* (NPB) [17] consiste em um conjunto de *benchmarks* e aplicações, desenvolvidas pela divisão NAS (*Advanced Supercomputing Division*) da NASA, agência espacial dos EUA, usadas para medir o desempenho de computadores paralelos. Atualmente, há três versões em várias linguagens, sendo C, Fortran e Java as principais, e utilizando diferentes modelos de programação paralela, como o modelo de passagem de mensagens, através de MPI [52], e o modelo de memória compartilhada, através de OpenMP [98] e Java Threads.

Em [108], é apresentada a implementação baseada em componentes no HPE de 3 aplicações simuladas (SP, BT e LU) e 1 benchmark (FT) do NPB. Ao todo, foram produzidos 116 componentes abstratos e 147 componentes concretos. Utilizando componentes do modelo HASH, foi possível fatorar as versões monolíticas originais desses programas, especificamente as versões Java e Fortran/MPI, tanto com relação a aspectos funcionais quanto aspectos não-funcionais, dentre os quais a topologia dos processos, padrões de comunicação, estratégia de distribuição das estruturas de dados e a cronometragem da execução, garantido um grau de modularidade que não é possível de ser atingido por outros modelos de componentes, uma vez que não possuem a capacidade

de distingui-los em espécies e de representar componentes intrinsicamente paralelo, isto é, distribuídos nas unidades de processamento na plataforma de computação paralela.

É importante ressaltar que, como os *benchmarks* SP e BT implementam variações do mesmo método para solução de sistemas não-lineares, foi possível o compartilhamento de grande parte dos componentes, ao contrário de suas versões monolíticas, que os tratam como programas completamente distintos. Na Figura 3.3, é ilustrado tal compartilhamento de componentes. É possível observar que SP e BT diferenciam-se apenas pela implementação particular dos componentes resolvedores (**x_solve**, **y_solve** e **z_solve**).

Outro aspecto importante é o encapsulamento do padrão de comunicação da aplicação LU em um esqueleto de computação paralela [46] encapsulado no componente WAVEFRONT, ilustrando uma importante técnica de programação naturalmente suportada por plataformas de componentes baseadas no modelo HASH, como o HPE, especialmente devido ao suporte ao HTS, como demonstrado no artigo.

Com relação à avaliação de desempenho, a metodologia adotada teve o objetivo de comparar, com rigor estatístico, as versões monolíticas dos *benchmarks*, usando C# e MPI.NET, com as suas versões baseadas em componentes, concentrando-se exclusivamente na sobrecarga resultante da fatoração em componentes. Os resultados foram favoráveis. A sobrecarga média, que decorre do uso de componentes, foi de 0.98%, nunca ultrapassando 7.5%. Em muitos cenários avaliados, a versão baseada em componentes foi considerada ligeiramente mais rápida do que a versão monolítica, embora as causas disso não tenham sido aprofundadas.

Em um trabalho posterior, nosso grupo de pesquisa realizou a avaliação de desempenho de máquinas de execução virtual, incluindo a plataforma Mono utilizada pelo HPE, comparando-as entre si e com a execução nativa, mostrando que o desempenho pode ser comparável ao código nativo se observadas certas regras de codificação, as quais variam entre uma ou outra implementação da máquina virtual, mesmo que sejam do mesmo padrão, no caso Java ou CLI. Curiosamente, para a máquina OracleJVM [100], a principal recomendação é exatamente a fatoração do código em subrotinas, o que é garantido pela fatoração em componentes, embora cause sobrecarga na máquina Java da IBM [70].

Os códigos fontes e planilhas de resultados desse estudo estão publicamente disponíveis em [92].

3.4 A Nuvem de Componentes

Nuvens computacionais [12] oferecem um modelo de utilização para plataformas de computação paralela que entende-se como promissor para atender às necessidades dos usuários das aplicações que demandam por Computação de Alto Desempenho [94, 125].

Apesar disso, ainda há importantes desafios conceituais para que plataformas de nuvens computacionais tornem-se alternativas viáveis para o suporte a aplicações de CAD. Dentre essas dificuldades, nos preocupa a ausência de paradigmas de composição de aplicações que ofereçam o nível de abstração adequado a esse contexto, tendo em vista a heterogeneidade das plataformas de computação paralela que é refletida nos modelos de programação que conseguem explorar de forma eficiente o seu desempenho. Essa constatação tem se tornado mais marcante com o recente projeto de plataformas equipadas com aceleradores computacionais [80], atualmente representados por *Unidades de Processamento Gráfico* (GPU[2]) [55, 120] e *Arranjos de Portas Programáveis em Campo* (FPGA[3]) [66], que somam-se à consolidação do uso de múltiplas hierarquias de paralelismo em uma mesma plataforma com a consolidação de processadores de múltiplos núcleos [80].

Esta dissertação está inserida no contexto da concepção, projeto e implementação, dentro do nosso grupo de pesquisa, de uma nuvem computacional cujo serviço provido será uma infraestrutura para o desenvolvimento de programas paralelos usando componentes de software que implementam os interesses das aplicações dos clientes da nuvem e que encontram-se implantados sobre essas plataformas de modo a explorar o seus recursos de processamento de forma eficiente. O serviço dessa nuvem combina aspectos de PaaS (*Platform-as-a-Service*), SaaS (*Software-as-a-Service*) e IaaS (*Infrastructure-as-a-Service*). Como **PaaS**, ela proveria um ambiente para o desenvolvimento e certificação de componentes e aplicações (software). Tais recursos devem ser vistos de maneira transparente através da abstração de componentes paralelos baseados no modelo Hash (componentes-#). Os programas seriam desenvolvidos através da composição de componentes que estão implantados na nuvem e a sua execução também se daria na nuvem. Como **SaaS**, ela proveria aplicações finais para os usuários, compostas através de componentes, para utilização de usuários especialistas de domínios específicos para resolver problemas de seu interesses. Como **IaaS**, ela proveria o acesso a plataformas de computação paralela, participantes da nuvem, para execução de suas aplicações. Tais plataformas estariam oferecidas também por meio da abstração dos componentes-#.

Em [86] é discutido o uso do modelo de componentes para fornecer um ambiente para o desenvolvimento e execução de aplicações em nuvens computacionais. Nesse trabalho é dito que plataformas de nuvens IaaS são, em vários aspectos, similares aos modelos de componentes distribuídos tradicionais, como o CCA. Nela uma máquina virtual pode ser vista também como um componente, podendo ser executado na nuvem. Dessa forma, uma nuvem de componentes pode ser vista como um recipiente de componentes de diversos tipos, fornecendo ambientes de virtualização através dos componentes que representam

[2]Do inglês, *Graphic Processing Units*.
[3]Do inglês, *Field-Programmable Gate Arrays*.

máquinas virtuais. Nossa abordagem vai mais longe, pois combina não só aspectos de IaaS e SaaS, mas também de PaaS, fornecendo um ambiente para o desenvolvimento, implantação e execução de componentes, que podem representar plataformas de execução (IaaS), aplicações (SaaS) ou outras espécies de componentes.

3.4.1 Intervenientes

Identificamos quatro perfis de usuários (intervenientes[4]) na nuvem de componentes:

- **Provedores de componentes**: são os usuários que desenvolvem componentes-# que implementam interesses diversos com o intuito de que seus componentes sejam usados para o desenvolvimento de aplicações pelos provedores de aplicações ou para o desenvolvimento de componentes por outros provedores de componentes;
- **Provedores de aplicações**: são os usuários que desenvolvem aplicações a partir da composição de componentes cujo objetivo é a execução em uma plataforma paralela eficientemente de forma a satisfazer as necessidades dos usuários de aplicações;
- **Mantenedores de Plataformas**: são os usuários que mantém plataformas de computação paralela participantes da nuvem, sobre as quais componentes devem estar implantados e aplicações podem ser instanciadas dinamicamente;
- **Usuários de aplicações**: são os usuários que visam a utilização das aplicações desenvolvidas na nuvem para a solução de problemas em domínios específicos de seu interesse, sobre uma determinada plataforma. Não envolve programadores profissionais, ao contrário dos usuários provedores, mas especialistas de domínios específicos, como cientistas e engenheiros.

A Figura 3.4 ilustra essa relação entre os intervenientes supracitados.

Os provedores de componentes são os intervenientes que proveriam, na nuvem, os componentes necessários à construção de aplicações, que poderiam ser componentes abstratos ou componentes concretos. Na sua maioria são programadores profissionais, especialistas em programação paralela para plataformas de computação paralela específicas. Eventualmente, podem ser programadores especialistas de um domínio específico, como cientistas computacionais e engenheiros, os quais dominam detalhes sobre a funcionalidade dos componentes, mas ainda assim possuindo conhecimento forte sobre programação paralela. Os provedores de componentes também fazem uso de outros componentes já existentes na nuvem, para desenvolver os seus próprios componentes.

Os provedores de aplicação são os responsáveis por prover aplicações já prontas na nuvem. Em sua maioria, esses intervenientes são empresas ou grupos de desenvolvedores

[4]Do inglês *stakeholders*.

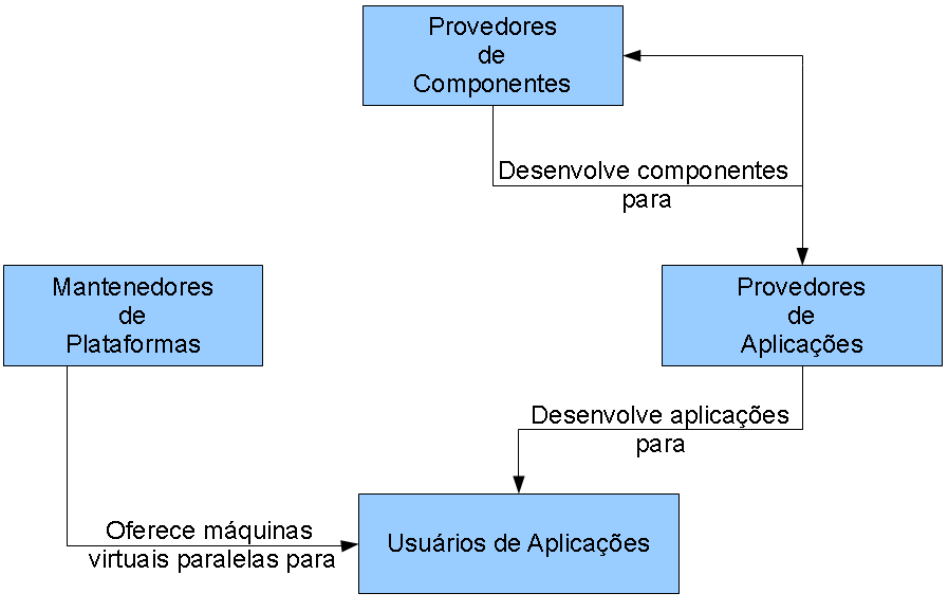


Figura 3.4: Relação entre os intervenientes

que desenvolvem softwares para domínios específicos, fazendo uso da composição de diversos componentes já existentes na nuvem para a construção de aplicações que seriam executadas na própria nuvem.

Os usuários de aplicação são os intervenientes que buscam soluções na nuvem para os problemas nas suas áreas de atuação. Esses usuários se utilizam de aplicações desenvolvidas na nuvem pelos provedores de aplicação para solucionar os mais diversos problemas como aplicações para previsão do tempo ou aplicações para simulações diversas.

Por fim, mantenedores de plataformas são responsáveis por implantar os componentes que devem executar em suas plataformas, oferecendo configurações de máquinas paralelas virtuais acessíveis para os usuários de aplicações.

3.4.2 Certificação Formal de Contratos de Componentes

Na nuvem de componentes, é desejável que provedores de componentes e aplicações e usuários de aplicações tenham alguma garantia sobre as funcionalidades dos componentes utilizados na nuvem, ou seja, tenham informações sobre a confiabilidade de um componente no que diz respeito às suas funcionalidades. Dessa forma, intervenientes não teriam surpresas desagradáveis no que diz respeito ao comportamento de um determinado componente. Com essas garantias, os riscos de que um componente ou aplicação se comporte de maneira imprevista diminui, pois o comportamento de cada componente usado na construção de outros componentes ou aplicações estaria bem definido. Existem algumas

formas de atingir esse objetivo:

- Um sistema de pontuação dada a cada componente, onde o usuário poderia pontuar o componente usado comparando o que o contrato descreve em relação ao que o componente deveria fazer e o que o componente concreto desenvolvido na realidade faz. Quanto maior a pontuação, mais confiável seria o componente;
- Uma série de testes desenvolvidos para componentes que implementam um determinado componente abstrato, cada teste serviria para testar, dependendo da espécie do componente, o comportamento do componente e/ou a sua saída. Dependendo dos testes poderíamos mensurar a confiabilidade do componente;
- Um processo de derivação de componentes, onde, através de um contrato formal de componente, que consiste em uma especificação escrita em uma linguagem de especificação formal, executaríamos uma sequência de passos de refinamento até atingirmos a especificação concreta do componente concreto desejado. Dessa forma tem-se certeza de que o componente concreto satisfaz o seu contrato. E, posteriormente, uma possível verificação da especificação concreta, por parte da nuvem e/ou dos usuários, a fim de verificar certas propriedades.

Dentre essas propostas de soluções, esta dissertação de mestrado investiga a terceira solução. Nesse caso, haveria uma linguagem de especificação que faria o papel da linguagem formal para a descrição dos contratos. Os contratos serviriam para a derivação de componentes concretos e posteriormente para que a nuvem saiba se um componente concreto implementa um determinado contrato, o que seria útil também para que a resolução de componentes concretos que implementem um determinado contexto leve também em consideração o comportamento do componente concreto e não apenas os seus tipos. No próximo capítulo, é apresentado o esboço de uma metodologia que contemple essa solução, enfatizando as questões em aberto que desejamos tratar em nosso trabalho.

Capítulo 4

Um Sistema de Contratos Formais para Componentes Paralelos do Modelo Hash

Dentre os intervenientes da nuvem de componentes, os *usuários de aplicações* podem ser vistos como os clientes finais, os quais fazem uso de aplicações oferecidas pelos *provedores de aplicações* e de plataformas de computação paralela mantidas pelos *mantenedores de plataformas* para a solução dos problemas do seu interesse. Os usuários de aplicação podem demandar certas garantias de segurança de execução aos provedores de aplicações, com respeito às aplicações disponibilizadas. Dentre outras motivações, deseja-se evitar o desperdício de tempo e recursos caso ocorra um erro em tempo de execução, bem como no processo de depuração da aplicação na correção de erros. Por sua vez, os provedores de aplicações podem também demandar garantias aos provedores de componentes no que diz respeito ao comportamento dos componentes disponibilizados para construção das suas aplicações.

Nesse contexto, é desejável que os provedores de aplicações e de componentes forneçam garantias quanto aos seus serviços (componentes) disponibilizados. Para isso, é proposta nesta dissertação uma metodologia de desenvolvimento que tem por objetivo oferecer maiores garantias de que um componente produzido obedeça a propriedades estabelecidas pelo seu contrato, uma generalização da noção componente abstrato utilizada no HPE. Dessa forma, provedores de aplicações e de componentes produziriam aplicações e componentes que são seguros por satisfazer propriedades especificadas pelo seu contrato. Além disso, de forma semi-automática, usuários de aplicações poderão verificar os componentes quanto a outras propriedades que considerarem relevantes, de forma *ad hoc*, sem que necessariamente essas propriedades estejam previstas e especificadas no contrato.

Na metodologia proposta, parte-se do contrato do componente, que é uma especificação formal do mesmo, e chega-se à sua especificação concreta através de um processo de refinamento. Esse processo consiste na aplicação de diversas leis de refinamento, as quais transformam uma especificação de entrada em uma especificação mais concreta e preservam as propriedades da especificação de entrada. Dessa forma, depois da aplicação de várias leis de refinamento, pode-se chegar a uma especificação totalmente concreta e, por transitividade, sabe-se que tal especificação obedece a especificação abstrata inicial. Uma especificação concreta é uma especificação que pode ser traduzida diretamente para uma linguagem de programação sem a necessidade de nenhuma outra transformação ou passos adicionais de refinamento. Por essa razão, é importante que a linguagem de especificação formal escolhida possibilite o processo de refinamento, pois assim é possível ter alguma segurança de que a especificação concreta obedece a sua especificação abstrata de partida.

A partir da especificação concreta, obtém-se uma implementação em uma linguagem de programação desejada através de um processo de tradução. No contexto de CAD, é desejável aplicarem-se regras de tradução convenientes para a arquitetura e para a linguagem alvo, fazendo um melhor uso dos recursos disponíveis, melhorando assim o desempenho do componente de forma individual. Levando isso em consideração, deve-se tratar requisitos específicos de aplicações nos domínios tradicionais de CAD, como, por exemplo, o uso de matrizes e vetores como estruturas de dados fundamentais, o suporte a números complexos e de ponto flutuante e a ordenação de laços para melhorar a localidade de acesso aos dados nas hierarquias de memória segundo técnicas disseminadas entre programadores no contexto de CAD. Tais requisitos, dependentes da plataforma de computação alvo, não são usualmente tratados por linguagens de especificação formal voltadas ao refinamento.

Além da garantia de cumprimento de contratos na implementação de componentes concretos, um outro problema importante é a escolha dinâmica e automática de componentes concretos que serão ligados a uma aplicação a partir de uma instanciação do contrato. Na Seção 3.3.1, foi discutido como o HPE trata esse requisito por meio da busca de componentes concretos segundo o *contexto* do componente abstrato, definido por seus parâmetros de tipo. Como contribuição desta dissertação, levou-se em consideração também o comportamento de um componente concreto, ou seja, um componente concreto é escolhido para atender a um contrato instanciado somente se ele respeita as propriedades declaradas pelo contrato estabelecido pelo componente abstrato que ele declara implementar e se esse contrato é compatível com o contrato instanciado.

A Figura 4.1 mostra como se daria o processo de derivação e verificação de um componente. Primeiramente, tem-se o contrato de um componente abstrato, o qual é refinado para um determinado contexto de implementação até obter-se uma especificação

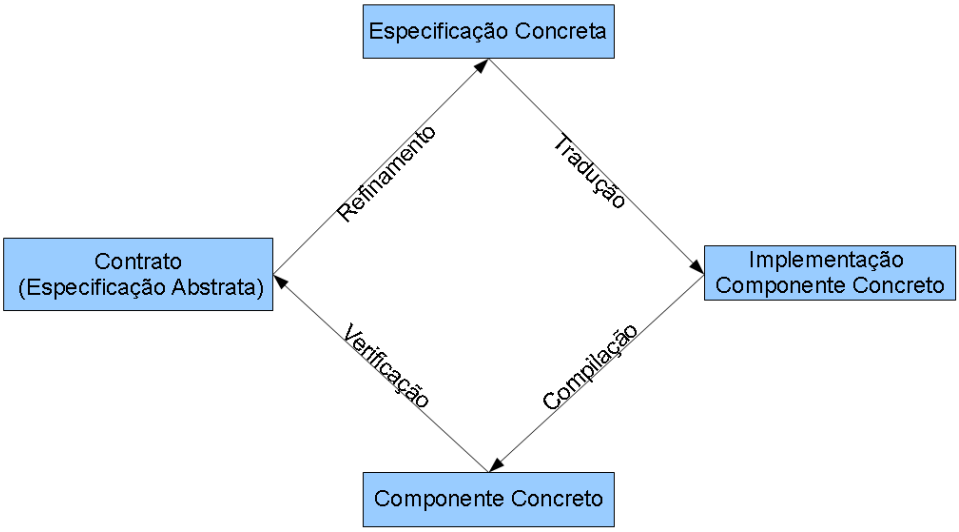


Figura 4.1: Diagrama do ciclo de derivação e verificação de componentes na nuvem

concreta que deverá ser traduzida para a linguagem de programação alvo, constituindo o componente concreto pretendido. O componente concreto pode então ser compilado de tal forma que seja possível haver verificações futuras no intuito de saber se tal componente satisfaz um determinado contrato, o que exigirá que o código objeto do componente carregue informações de sua especificação contratual .

O processo proposto envolve os provedores de componentes e de aplicações, pois são eles os responsáveis pelo desenvolvimento de *software* na nuvem de componentes. Caberia à nuvem de componentes a responsabilidade de verificar se componentes e aplicações respeitam seus contratos, dando-se preferência a componentes que ofereçam alguma garantia de que foi implementado de acordo com o seu contrato, ou seja, aqueles que foram desenvolvidos segundo a metodologia proposta neste trabalho. Nesse caso, seria ainda necessária a proposta de um processo de certificação de componentes e/ou desenvolvedores, o qual não é tratado especificamente nesta dissertação.

Enfim, a linguagem de especificação formal adequada para os requisitos da metodologia proposta neste trabalho deve ser capaz de especificar tanto aspectos funcionais dos componentes paralelos quanto o seu comportamento, levando em consideração requisitos de aplicações e plataformas de computação paralela no contexto de CAD. Além disso, é indispensável que tenha o suporte de um cálculo de refinamento, para que seja possível seguir o processo de refinamento proposto. Ou seja, a partir do contrato do componente

descrito na linguagem de especificação escolhida, chega-se a uma especificação mais concreta e, eventualmente, ao código em uma linguagem de programação.

4.1 Especificação de Componentes HASH usando Circus/HCL

No HPE, unidades de componentes das espécies computação e comunicador são ações que executam operações em uma determinada ordem parcial. A fim de descrever em que ordem as unidades de um componente devem ser ativadas, utiliza-se as expressões de comportamento herdadas da linguagem Haskell$_{\#}$. Assim, o comportamento de um #-componente é definido de forma exógena, separando os interesses coordenação de processos dos interesses de computação realizados por esses processos. Tais expressões de comportamento são o bastante apenas para descrever a coordenação dos componentes. Porém, uma vez que no HPE a linguagem que descreve a parte computacional dos componentes não é mais Haskell, o que nos permitia a aplicação do raciocínio formal à nível de computação, é preciso uma linguagem de especificação formal que descreva os componentes a nível de computação, seja qual for a linguagem empregada na sua descrição.

O ponto de partida para este trabalho de pesquisa foi a escolha da linguagem Circus para estender a especificação formal de #-componentes com a descrição de seus aspectos funcionais [37]. A linguagem Circus foi escolhida porque permite a descrição, de maneira simples, de aspectos comportamentais e funcionais de programas concorrentes, permitindo assim a análise de propriedades de segurança e continuidade através de provadores de teoremas publicamente disponíveis e de uso disseminado, como o ProofPower-Z [131], além de oferecer um cálculo de refinamento para geração de código. Tal integração traz vantagens mútuas, pois o Circus oferece ao HPE a possibilidade de especificação de #-componentes juntamente com a possibilidade de refinamento de tais especificações para código C# e o HPE oferece ao Circus uma forma de composição de especificações de forma à possibilitar especificações mais modulares.

A Figura 4.2 apresenta a gramática da linguagem Circus com as extensões #. Tais extensões precisaram ser feitas para uma melhor adequação à composição por sobreposição, agora aplicada a especificações Circus. A linguagem Circus com as extensões # será doravante chamada de Circus/HCL.

Além das extensões descritas na Figura 4.2, também são acrescentados alguns açúcares sintáticos, descritos a seguir:

- O tipo $Array_k(T)$, onde T é um tipo primitivo próprio da linguagem Z, denota um array k-dimensional de elementos do tipo T e é definido como: $Array_k(T) == \mathbb{N} \times \mathbb{N} ... \times \mathbb{N} \pfun T$;
- A função $\_[\_, \_ ..., \_] == Array(T) \times \mathbb{N} \times \mathbb{N} ... \times \mathbb{N} \to T$, se utilizada como expressão, corresponde a função de indexação de um array k-dimensional e é

Program	::=	HHeader HJoin* CircusPar*	1 – *#-component declaration*
·HHeader	::=	Kind HASHId **where**	2 – *kind of the #-component*
·Kind	::=	**computation** | **synchronizer** | **data**	
·HJoin	::=	**inner component** $\overline{N}$ HRangeSet$^?$: HASHId	3 – *declaring an inner component*
· · HRangeSet	::=	[HRange$^+$]	4 – *indexed notation*
· · HRange	::=	HExpr . . . HExpr	
·HASHId	::=	$\overline{N}$ HParams$^?$ HPublic$^?$	5 – *Reference of a #-component*
· · HParams	::=	$\langle$HExpr$^+\rangle$	6 – *parameters of the #-component*
· · HPublic	::=	$\left(\overline{N}^+\right)$	7 – *public inner components*
CircusPar	::=	. . . | **process** $N \triangleq$ HSlice* **Proc**	8 – *declaring a unit (process)*
·HSlice	::=	**slice** $\overline{N}$ HIndex$^?$.$\underline{N}$	9 – *declaring a slice of the unit*
· · HIndex	::=	[N$^+$]	10 – *indexed notation*
Proc	::=	. . . | $\underline{N}$	11 – *reference to a slice (process)*
Action	::=	**action**$^?$ . . . | $\underline{N}$!	12 – *reference to the slice action*
Schema-Ref	::=	. . . | $\underline{N}$::	13 – *reference to the slice state*

Figura 4.2: Gramática do Circus com extensões # [37]

declarado implicitamente antes de qualquer especificação. Se utilizada para se atribuir um valor a uma determinada posição do array, corresponde a um açúcar sintático para a operação de sobrescrever (*override*), onde $array[i_1, ..., i_n] = x$ corresponde a operação $array \oplus \{(i_1, ..., i_n) \mapsto x\}$;

- Muitas aplicações científicas reais fazem uso de números complexos na solução de problemas da engenharia e física, dentre outros, como por exemplo na medição de correntes elétricas ou na análise de estresse de estruturas como pontes ou construções. O tipo *Complex* denota um número complexo e é definido como: $Complex == \mathbb{R} \times \mathbb{R}$. O primeiro valor da dupla corresponde a parte real e o segundo a parte imaginária do número complexo. Todas as operações referentes aos números reais são sobrecarregadas para os complexos, levando em consideração as propriedades dos números complexos.

Esses açúcares sintáticos servem para representar estruturas fundamentais na construção de aplicações no contexto de CAD, especificamente aquelas no contexto científico e de engenharia, que são aplicações que contêm processamento numérico intensivo. Eles também serão úteis na Seção 4.2, na subseção que trata sobre tradução de uma especificação concreta para um linguagem de programação e no Capítulo 5, que trata dos estudos de caso.

4.1.1 Relação entre #-componente e especificação Circus/HCL

Uma especificação Circus/HCL representa um #-componente, onde seus processos denotam as *unidades* do #-componente, e a ação principal de cada unidade denota o seu

protocolo, definido pela combinação dos processos que representam suas fatias usando os combinadores de processos de CSP. Assim como em HCL, também é válido em Circus/HCL:

- as *ações* publicadas por uma unidade são introduzidas por declarações **action** e somente podem ser referenciadas no protocolo da unidade (ações que não são precedidas pela palavra-reservada **action** não são visíveis externamente, mas apenas referenciadas em ações e no protocolo da unidade);
- as *condições* de uma unidade são introduzidas por declarações **condition**, e podem ser referenciadas nas ações e protocolo da unidade;
- ações de uma fatia que não são referenciadas no seu protocolo, podem ser usadas livremente na unidade;
- ações de uma fatia que são referenciadas no seu protocolo devem ser usadas no protocolo da unidade obedecendo a restrição imposta pelo protocolo da fatia.

Canais, em Circus/HCL, são #-componentes primitivos da espécie *comunicador* que encapsulam operações primitivas de passagem de mensagens. Para isso, tais componentes contém uma unidade denominada *receive*, através do qual são realizados operações de recebimento de mensagens pelo canal, e uma unidade chamada *send*, para operações de envio de mensagem. Unidades paralelas são representadas como processos replicados através do operador $|||$ do CSP, o que faz sentido semanticamente, uma vez que as unidades executam independentemente umas das outras no HPE.

A operação de *junção* (*join*) recebe como operandos especificações Circus de componentes, e denota uma única especificação contendo a união de todos os processos e canais dos dois operandos. Nas figuras 4.3(a) e 4.3(b), são representados esquemas arbitrários de especificações de componentes-, denominados U e V. Na Figura 4.3(c), está representada a junção das duas especificações, em uma nova especificação que inclui as unidades e canais das especificações originais.

Em uma especificação Circus/HCL, a operação de *junção* é definida pelo conjunto de declarações **inner component** da especificação, a qual tem como efeito a junção das especificações dos componentes aninhados que declaram.

A operação de *entrelaçamento* (*fold*) recebe dois operandos, que são processos Circus denotando unidades de um componente e denota um novo processo que contém a união dos estados dos dois operadores e uma combinação de suas ações. A Figura 4.4 apresenta dois exemplos de entrelaçamento, um dos quais ilustra o compartilhamento de uma fatia. Nesse exemplo, as unidades U (Figura 4.4(a)) e S (Figura 4.4(b)) são fatias das unidades Q (Figura 4.4(c)) e R (Figura 4.4(d)), respectivamente. Dessa forma, Q e R incluem os estados e as ações de suas fatias U e S. Na Figura 4.4(e), a unidade P é obtida através

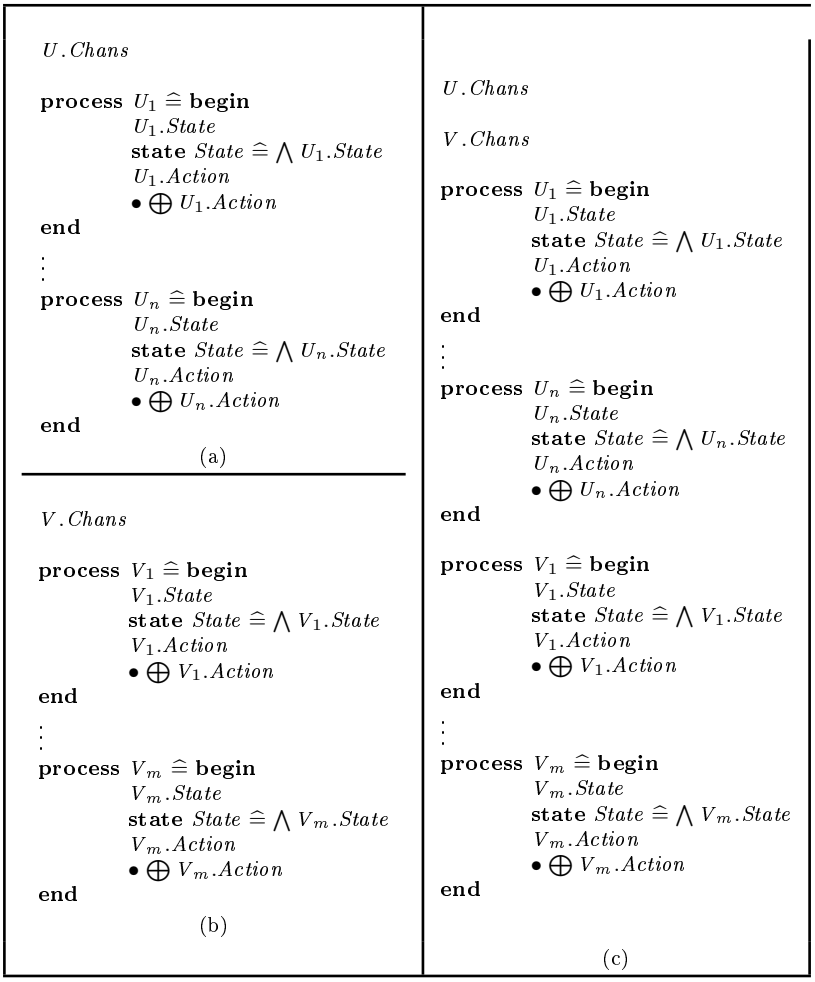


Figura 4.3: Operação de Junção de Especificações Circus [37]

do entrelaçamento entre as unidades Q e R, sem o compartilhamento de fatias. Assim, a unidade P inclui os estados e ações das fatias Q e R, e suas ação principal é uma combinação das ações principais de suas fatias. Na Figura 4.4(f), Q e R compartilham a fatia S, ou seja, todos os estados e ações de S são compartilhados por Q e R. Para que isso faça sentido, assume-se que $S.State \subseteq U.State$ e $S.Action \subseteq U.Action$, ou seja S é também fatia de U. No resultado do entrelaçamento, P contém todos os estados e ações de $R - S$ e Q, onde S está incluído.

A Figura 4.5 ilustra a hierarquia de unidades resultantes do exemplo de entrelaçamento da Figura 4.4, onde as setas sólidas ligam uma unidade às fatias nela diretamente declaradas e as setas pontilhadas ligam uma unidade a suas fatias transitivamente ligadas.

Em uma especificação Circus/HCL, a operação de entrelaçamento corresponde a declaração de unidades (declaração **process**), mais especificamente concretizada pela declaração de suas fatias, por meio do conjunto de declarações **slice**, e da especificação do seu protocolo, a partir da combinação das ações das fatias. Em uma especificação

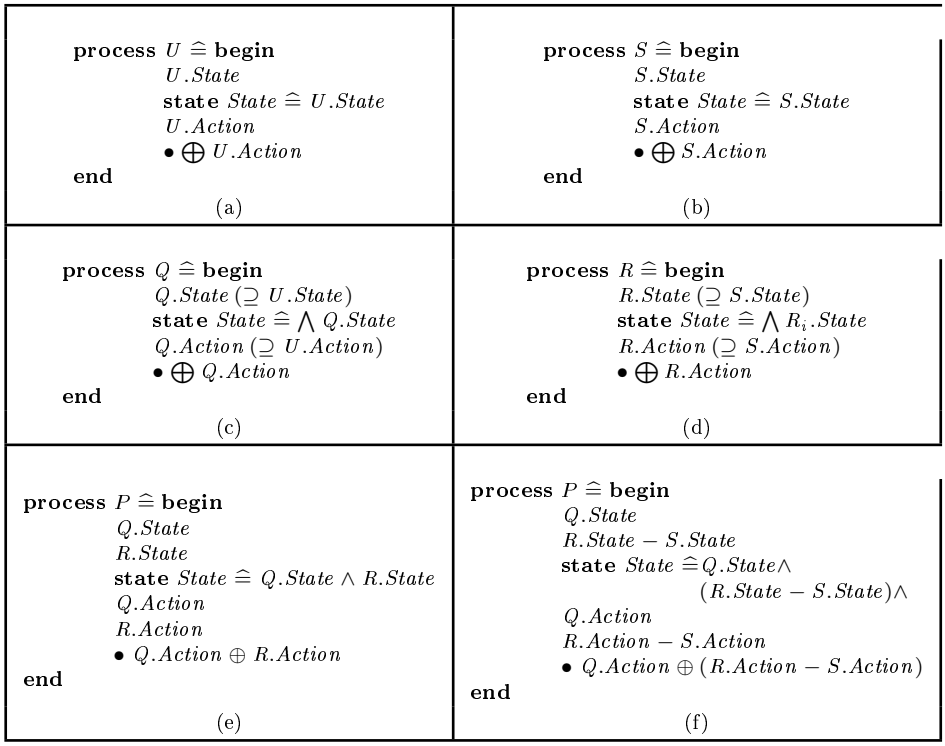


Figura 4.4: Operação de Entrelaçamento entre Processos Circus [37]

Circus/HCL bem formada, todas as unidades dos componentes aninhados participam de alguma operação de entrelaçamento, ou seja, são fatias de alguma unidade do componente onde estão aninhados. O entrelaçamento com compartilhamento diz respeito ao compartilhamento de componentes entre componentes aninhados em uma configuração, que ocorre, por exemplo, quando computações, encapsuladas em componentes, processam as mesmas instâncias de estruturas de dados encapsuladas em um mesmo componente.

Unidades paralelas também podem ser usadas em um entrelaçamento, desde de que as suas faixas de índices sejam iguais e que as variáveis que são usadas na indexação tenham

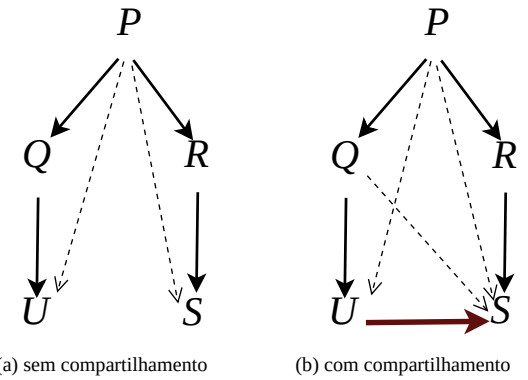


Figura 4.5: Hierarquia de Unidades no Entrelaçamento da Figura 4.5

nomes iguais ou que sejam renomeadas para um nome comum. Por exemplo, sejam as unidades paralelas

$$\textbf{process } P_1 \mathrel{\widehat=} |||i : \{0...N-1\} \bullet MainAction_1$$

e

$$\textbf{process } P_2 \mathrel{\widehat=} |||j : \{0...N-1\} \bullet MainAction_2$$

. Aplicando-se a operação de entrelaçamento entre P_1 e P_2, poderia-se obter

$$\textbf{process } P_3 \mathrel{\widehat=} |||k : \{0...N-1\} \bullet MainAction_1[i/k] \Box MainAction_2[j/k]$$

, onde $MainAction_1[i/k]$ é uma ação igual a $MainAction_1$, porém com todas as ocorrências de i substituídas por k.

Em componentes-#, algumas vezes é preciso dividir uma unidade paralela em duas ou mais unidades paralelas. Por exemplo, seja a unidade paralela

$$\textbf{process } P \mathrel{\widehat=} |||i : \{0...N-1\} \bullet MainAction$$

. É possível dividi-la em duas unidades paralelas, assim definidas:

$$\textbf{process } P_1 \mathrel{\widehat=} |||i : \{0...N/2\} \bullet MainAction \text{ e}$$

e

$$\textbf{process } P_2 \mathrel{\widehat=} |||i : \{N/2+1...N-1\} \bullet MainAction$$

, onde $\{0...N/2\} \cup \{N/2+1...N-1\} = \{0...N-1\}$.

4.1.2 Exemplos

A Figura 4.6 mostra a especificação Circus do componente BROADCAST da espécie comunicador no HPE. O propósito desse componente é distribuir um número, que inicialmente está na unidade de ranque 0 da unidade paralela, para todas as demais unidades. No cabeçalho, tem-se a espécie do componente, o seu nome e o valor N, que representa a cardinalidade arbitrária das unidades paralelas. Depois, declaram-se os canais utilizados no componente e, por último, as unidades do componente. A unidade *recunit* é uma unidade paralela, ou seja, representa N unidades individuais que possuem a mesma especificação e atuam em paralelo. As unidades pertencentes a uma unidade paralela executam independentemente umas das outras, se comunicando através do canal c. Na definição da unidade paralela, tem-se que cada unidade armazena um número k. A unidade de ranque 0 da unidade paralela manda o seu número, que neste exemplo vale 3, para todas as outras unidades, enquanto as outras unidades esperam pelo número. Tal componente

synchronizer BROADCAST$\langle N \rangle$ **where**

channel $c : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

process $recunit \mathrel{\widehat=} \|_c\ i : \{0..N-1\} \bullet$

begin

state $State \mathrel{\widehat=} [k : \mathbb{N}]$
$updSt \mathrel{\widehat=} [State', v? : \mathbb{N} \mid k' = v?]$
$ldSt \mathrel{\widehat=} [State, v! : \mathbb{N} \mid v! = k]$
action $broadcast \mathrel{\widehat=}$ **var** $v : \mathbb{N} \bullet$

$(ldSt;\ |||j : \{1..N-1\} \bullet c.0.j!v \rightarrow SKIP)$
$\lhd i = 0 \rhd$
$(c.0.i?v \rightarrow updSt)$

$\bullet\ broadcast$

end

Figura 4.6: Contrato do componente BROADCAST, da espécie comunicador

synchronizer REDUCESUM$\langle N \rangle$ **where**

channel $c : \mathbb{N} \times \mathbb{N} \times \mathbb{R}$

process $reduce \mathrel{\widehat=} \|_c\ i : \{0..N-1\} \bullet$

begin

state $State \mathrel{\widehat=} [k : \mathbb{R}]$
$updState \mathrel{\widehat=}$ **val** $aux : \mathbb{R} \bullet [\Delta State, aux? : \mathbb{R} \mid k' = k + aux?]$
action $reduceSum \mathrel{\widehat=}$ **var** $aux : \mathbb{R} \bullet$

$(i = 0\ \&\ |||\, j : 1..N-1 \bullet c.j.0?aux \rightarrow updState(aux);$
$|||\, j : 1..N-1 \bullet c.0.j!k)$
$\Box$
$(i \neq 0\ \&\ c.i.0!k \rightarrow c.0.i?aux \rightarrow updState(aux))$

$\bullet\ reduceSum$

end

Figura 4.7: Contrato do componente REDUCESUM, da espécie comunicador

computation VECVECPRODUCT$\langle N \rangle$ (u, v, r, i) **where**

inner component u: VECTOR $\langle N \rangle$
inner component v: VECTOR $\langle N \rangle$
inner component r: REDUCESUM $\langle N \rangle$
inner component i: REALDATA $\langle N \rangle$

process $product \mathrel{\widehat=} \Big|\Big|\Big| i : \{0..N-1\} \bullet$
begin

slice $u.vector[i]$
slice $v.vector[i]$
slice $r.reduce[i]$
slice $i.real[i]$

state $State \mathrel{\widehat=} [u::; v::; r::; i:: | u::dim = v::dim]$
$computeLocal \mathrel{\widehat=} [\Delta State \mid r::k' = \sum_{j=0}^{v::dim-1} v::v[j] \times u::v[j]]$
$updValue \mathrel{\widehat=} [\Delta State \mid i::k' = r::k]$
$\bullet$ $computeLocal;\ r!;\ updValue$

end

Figura 4.8: Contrato do componente VECVECPRODUCT, da espécie computação

poderia ser um componente aninhado de uma especificação maior para fazer o *broadcast* de um número entre as unidades de uma unidade paralela.

Na Figura 4.7, o contrato do componente abstrato REDUCESUM encapsula uma operação coletiva de comunicação entre processos, muito comum em programas paralelos. Suponha um conjunto de processos no qual cada processo detém um número. A operação soma os números dos processos envolvidos e acumula o resultado em um processo, dito raiz, e distribui o resultado para todos os demais processos.

Na Figura 4.8, o contrato VECVECPRODUCT encapsula a operação de produto entre dois vetores, onde cada processo contem uma partição de ambos os vetores. Tais partes são correspondentes em ambos os vetores, de modo que a operação fará o produto entre as partições correspondentes localmente dos vetores e, depois, fazendo o uso do componente REDUCESUM, irá somar as partes contidas em cada processo e redistribuí-las novamente para cada um. Na linguagem Circus/HCL, a expressão $slice :: v$ corresponde à variável v pertencente à fatia $slice$. O componente VECVECPRODUCT possui três componentes aninhados além do componente REDUCESUM. São dois componentes do tipo VECTOR, que encapsulam os vetores a serem multiplicados, e um componente do tipo REALDATA que armazenará o resultado da operação. Os contratos dos componentes VECTOR, REALDATA são ilustrados na Figura 4.9.

data VECTOR$\langle N \rangle$ **where**
$\mid dim : \mathbb{N}$
process $vector \mathrel{\widehat=} \|\|\| i : \{0..N-1\} \bullet$
begin

state $State \mathrel{\widehat=} [v : Array_1(\mathbb{R})]$
$\bullet\ SKIP$

end

data REALDATA$\langle N \rangle$ **where**
process $real \mathrel{\widehat=} \|\|\| i : \{0..N-1\} \bullet$
begin

state $State \mathrel{\widehat=} [k : \mathbb{R}]$
$\bullet\ SKIP$

end

Figura 4.9: Contratos dos componentes VECTOR and REALDATA

4.1.3 Especificações Circus/HCL como Contratos de Componentes

Na Seção 3.4, foram explicado a infraestrutura e os papéis desempenhados no contexto da nuvem de componentes. No início deste capítulo, foi discutido então o arcabouço geral do processo de certificação de software na nuvem de componentes, a partir do requisito da nuvem de componentes em oferecer aos seus intervenientes garantias em relação aos componentes quanto às suas funcionalidades definidas em contratos. Dentro desse contexto, propõe-se o uso de contratos formais para especificação dos componentes abstratos no HPE. Com contratos formais assim definidos, consegue-se expressar não apenas os compromissos sintáticos de tipos, que é hoje suportado pelo HPE, mas também o padrão de comportamento que os componentes concretos devem implementar. Dentro desse arcabouço, é possível obter um componente concreto a partir de uma especificação concreta derivada por refinamentos sucessivos a partir de um contrato aplicado a um determinado contexto de implementação, tomando por base a abstração de *contexto de implementação* do sistema de tipos HTS.

A Figura 4.10 mostra como se daria o ciclo de derivação e certificação de um componente-# usando a linguagem Circus/HCL.

A Seção 4.2 tem por objetivo explicar em detalhes o processo pelo qual será derivado um componente concreto, partindo de um contrato, ou seja, partindo da especificação do componente abstrato o qual ele implementa. Para isso, será utilizada a linguagem formal Circus/HCL, que é uma extensão da linguagem Circus, fazendo o papel da linguagem de especificação formal utilizada nos contratos, a qual é capaz de especificar os aspectos de coordenação e de computação de componentes-#, bem como suporta o processo refinamento requisitado.

A Seção 4.3 explica como é possível usar a especificação concreta de um componente para verificar se tal componente satisfaz um determinado contrato. A certificação pode ser feita de forma automática através do provador ProofPower-Z, ou de forma manual.

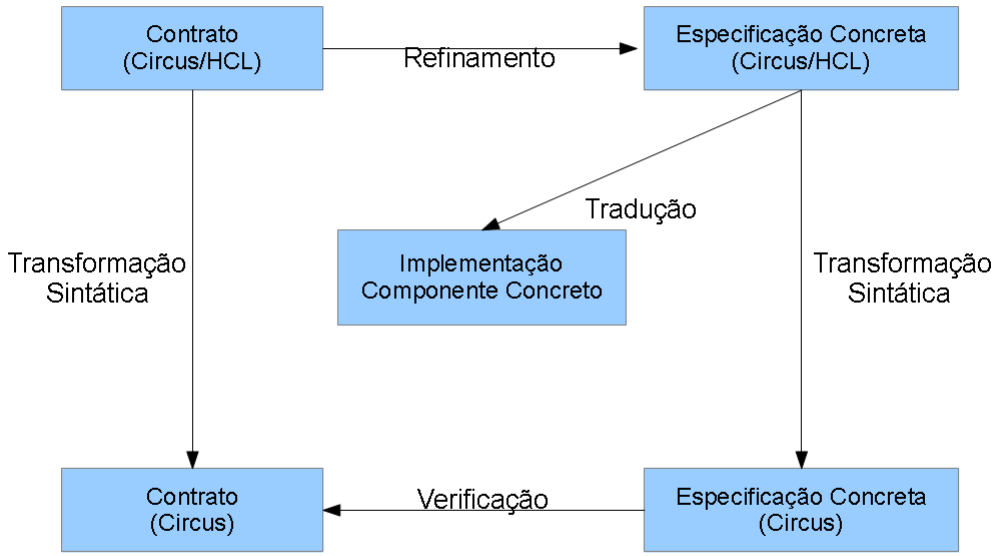


Figura 4.10: Diagrama do ciclo de derivação e verificação de componentes na nuvem utilizando Circus

4.2 Derivação Formal de Componentes usando Circus/HCL

O processo de derivação de componentes na nuvem, fazendo uso da linguagem Circus/HCL para a descrição dos contratos de componentes abstratos, consiste em dois passos, ao final dos quais deseja-se oferecer garantias de que o componente derivado satisfaz às propriedades do seu contrato.

No primeiro passo, parte-se de um contrato, que se trata de uma especificação abstrata descrita na linguagem Circus/HCL onde estão especificadas as propriedades e funcionalidades dos componentes que o implementam. A seguir, aplica-se passos de refinamento ao contrato, onde cada passo torna o contrato em uma especificação mais concreta. Aplicam-se passos de refinamento até que o contrato possa ser classificado como uma especificação concreta.

Uma especificação concreta Circus/HCL é uma especificação onde não existem ações descritas na notação Z, ou seja, todos as ações estão descritas através das linguagens CSP e GCL. Além disso, nem todos as construções da GCL são permitidas. Em uma especificação concreta, não pode haver o uso de instruções de especificação. Além disso, todas as guardas não fazem uso de quantificadores, ou seja, em guardas só são usados os operadores de conjunção ($\wedge$), disjunção ($\vee$) e negação ($\neg$).

O segundo passo consiste em traduzir a especificação concreta de um componente para uma linguagem alvo. Para isso, o desenvolvedor do componente, ou da aplicação, precisa traduzir cada instrução na GCL ou em CSP em uma instrução na linguagem alvo. Dada a simplicidade da GCL, juntamente com a semântica de canais da linguagem CSP, a tarefa de tradução não é uma tarefa difícil, consistindo basicamente de substituições de instruções.

4.2.1 Refinamento do Contrato

O passo de refinamento de contrato pode ser visto como o passo vital no processo de derivação de um componente, pois é nele em que acontece de fato a transformação da especificação abstrata em uma especificação concreta que pode ser traduzida para uma linguagem de programação. O passo de refinamento consiste na aplicação sucessiva de leis de refinamento. Cada aplicação de uma lei leva à especificação de uma estrutura mais abstrata para uma especificação mais concreta, que obedece às mesmas invariantes, pré-condições e pós-condições. Em uma especificação Circus/HCL concreta, assim como em uma especificação Circus concreta, não há parágrafos Z, as guardas devem ser expressões booleanas e não há não-determinismos na especificação, que pode ser introduzido pelo uso do operador de não-determinismo da linguagem CSP, por exemplo. Também não há guardas de comandos expressas com quantificadores.

As leis de refinamento usadas neste trabalho são um subconjunto de um grande conjunto de leis, as quais foram previamente provadas quanto ao fato de que preservam as propriedades da especificação original, ou seja, suas invariantes, pré-condições e pós-condições [42, 60, 95].

No processo de refinamento, o uso das leis usadas em um refinamento depende da intenção do desenvolvedor quanto à especificação concreta e ao código final desejado. Dessa forma, as leis utilizadas neste trabalho foram escolhidas de forma que a tradução para a especificação concreta seja simples e o código gerado seja o mais otimizado possível.

Leis de Refinamento

Para os propósitos deste trabalho, faz-se uso de seis leis de refinamento: *Conversão Básica*, *Composição Sequencial*, *Introdução de Atribuição*, *Alternativa/Guarda - Troca*, *Introdução de Alternativa* e *Fortalecimento da Pós-Condição*. Esta seção apresenta formalmente tais leis de refinamento e definição, bem como discute a intuição por trás de cada uma delas. A seguir, se é dito que $A \sqsubseteq B$ isso significa que B refina A e se é dito que $A = B$ significa que $A \sqsubseteq B$ e $B \sqsubseteq A$ e a expressão $exp[new/old]$ é equivalente a expressão exp com todas as ocorrências de old substituídas por new.

A *Lei de Conversão Básica* transforma um esquema na notação Z em uma instrução de especificação da GCL, uma vez que grande parte das leis de refinamento estão definidas em termos dessas instruções.

Lei L.1 Conversão Básica

$(\Delta S;\ di?;\ do!\ \mid\ p) = \alpha d, \alpha do! : [inv \land \exists\, d', do! \bullet inv' \land p, inv' \land p]$

Onde $S \mathrel{\widehat=} (d\ \mid\ inv)$.

Essa lei declara que, dado um esquema que possui variáveis de entrada e saída, uma pós-condição e faz-se uma mudança no estado do processo, estado esse que possui variáveis declaradas e uma invariante. A instrução de especificação que equivale a esse esquema tem por variáveis que podem ser mudadas as variáveis declaradas no estado e as variáveis de saída. Tem por pré-condição a invariante do estado e a existência de algum valor para as variáveis que serão modificadas tal que tais variáveis satisfaçam a pós condição. Isso é fundamental, pois se não houvesse nenhum valor possível das variáveis de saída que satisfizesse a pós-condição, então tal instrução de especificação não poderia ser implementada. Por fim, ela tem por pós-condição a invariante e a pós-condição declarada no esquema.

A *Lei da Composição Sequencial* tem por objetivo dividir uma instrução de especificação em duas. Ela é útil, quando precisa-se compor uma instrução de especificação complexa de ser implementada em várias mais simples.

Lei L.2 Composição Sequencial

$w{:}[pre, pos] = w{:}[pre, mid[w'/w]];\ w{:}[mid, pos]$

Restrições

- mid não tem variáveis livres decoradas
- Nenhuma variável livre de pos pertence a w

Esta lei declara que uma instrução de especificação pode ser dividida em duas, onde a pós-condição da primeira e a pré-condição da segunda consiste em um objetivo intermediário. Desta forma, após a conclusão da primeira instrução de especificação, que tem como pré-condição a pré-condição da instrução de especificação original, tem-se que a sua pós-condição é satisfeita, como tal pós-condição é pré-condição para a segunda instrução de especificação e a pós-condição da segunda instrução de especificação é a própria pós-condição da instrução de especificação original, tem-se que a pós-condição da instrução de especificação original é satisfeita, que é exatamente o mesmo resultado cuja instrução de especificação original se propõe a fazer.

A *Lei de Introdução de Atribuição* tem por objetivo substituir uma instrução de especificação por uma atribuição, que satisfaz a pré e pós condições da instrução de especificação. Essa lei é amplamente utilizada em substituições simples, cuja pós-condição da instrução de especificação consiste apenas em atribuições.

Lei L.3 Introdução de Atribuição

$w, vl{:}[pre, pos] \sqsubseteq vl := el$

Restrições

- $pre \Rightarrow pos[el/vl'][\_/']$

Esta lei declara que se uma atribuição de valores faz com que a pré-condição implique a pós-condição então ela pode substituir a instrução de especificação. Intuitivamente, se uma atribuição cumpre com o propósito de uma instrução de especificação, que é, se tem-se as pré-condições, após a execução da instrução de especificação teremos a pós-condição, então tal atribuição pode substituir tal instrução de especificação, e essa lei formaliza tal intuição.

A *Lei de Alternativa/Guarda - Troca* tem por objetivo apenas transformar guardas da linguagem CSP em guardas na GCL e vice-versa. As guardas na linguagem CSP são mais complicadas de se implementar do que na GCL, pelo fato de haver operações de sincronização envolvidas, desta forma, no que diz respeito a tradução para uma linguagem de programação, é mais conveniente trabalhar com a GCL.

Lei L.4 Alternativa/Guarda - Troca

if $g_1 \rightarrow A_1 \,[]\, g_2 \rightarrow A_2$ **fi** $= g_1 \,\&\, A_1 \,\Box\, g_2 \,\&\, A_2$

Restrições

- $g_1 \vee g_2$
- $g_1 \Rightarrow \neg\, g_2$

A lei basicamente substitui o operador de alternativa & do CSP pela construção **if** da GCL. As restrições se dão pelo fato de uma pequena diferença na semântica das duas operações. Na construção **if** da GCL, se mais de uma guarda é satisfeita, a escolha é feita de forma não-determinística, enquanto com o operador de alternativa & do CSP, por estar sujeita ao operador determinístico de escolha externa $\Box$, a escolha é determinística. Já se nenhuma das guardas é satisfeita a construção **if** da GCL diverge, enquanto com operador de alternativa & entra em *deadlock*. As restrições garantem que apenas uma das duas guardas será satisfeita.

A *Lei de Introdução de Alternativa* tem o objetivo de inserir a estrutura de alternativa da GCL partindo-se de uma instrução de especificação.

Lei L.5 Introdução de Alternativa

$w : [pre, pos] \sqsubseteq$ **if** $[]_i g_i \rightarrow w : [g_i \wedge pre, pos]$ **fi**

Restrições

- $pre \Rightarrow \bigvee_i g_i$

Esta lei declara que, se uma instrução de especificação tem sua pré-condição satisfeita, tal instrução de especificação pode ser substituída por uma estrutura de alternativa.

Para isso, também é preciso garantir que uma das guardas da alternativa será satisfeita, pois, caso contrário, a ação divergiria. A restrição proposta nesta lei garante isso. Intuitivamente, tem-se que uma instrução de especificação $w : [pre, pos]$ é refinada por outra $w : [g_i \land pre, pos]$, se e somente se for possível garantir g_i. Neste caso, como $w : [g_i \land pre, pos]$ é guardado pela guarda g_i, tal condição é satisfeita.

Por fim, a *Lei de Fortalecimento da Pós-Condição* tem o objetivo de tornar mais específica a pós-condição de uma instrução de especificação.

Lei L.6 Fortalecimento de Pós-Condição

$w : [pre, pos] \sqsubseteq w : [pre, npos]$

Restrições

- $pre \land npos \Rightarrow pos$

Esta lei declara que, se as pré-condições de duas instruções de especificações forem as mesmas e a conjunção da pré-condição e pós-condição da segunda instrução de especificação garantir que a pós-condição da primeira será satisfeita, então a segunda pode ser substituída pela primeira. Intuitivamente, tem-se que se uma instrução de especificação C dá todas as garantias relativas a pós-condição de outra instrução de especificação A, e opcionalmente dá mais garantias, então a instrução de especificação C pode substituir a instrução de especificação A.

A definição D.1 denota nada mais do que uma abreviação.

Definição D.1 $(x : T \bullet A)(e) \mathrel{\widehat=} A[e/x]$

Uma chamada de função com passagem de parâmetro, onde a função está descrita da forma acima, nada mais é do que a substituição do valor da variável do parâmetro pelo valor passado. Nesse caso, não pode ser atribuído nenhum valor ao parâmetro, ele apenas pode ser usado em expressões.

Exemplos de Refinamento

Na linguagem Circus/HCL, deve-se preocupar-se apenas com o refinamento das ações e estados referentes à linguagem Circus, ou seja, as extensões criadas não devem ser refinadas. Nesta seção, serão usados os contratos dos componentes ReduceSum e VecVecProduct, respectivamente apresentados nas figuras 4.7 e 4.8.

Apesar do contrato de ReduceSum deixar claro a funcionalidade a ser implementada, a especificação não é concreta, pois ainda existe o uso do esquema *updState* que deve ser eliminado no processo de refinamento. Assim, para o esquema

$$updState \mathrel{\widehat=} \mathbf{val}\ aux : \mathbb{R} \bullet [\Delta State, aux? : \mathbb{R} \mid k' = k + aux?]$$

, aplicam-se as leis L.1, para converter um esquema em uma instrução de especificação, e L.3, para transformar uma instrução de especificação em uma atribuição, obtendo-se:

$$updState \mathrel{\widehat=} \textbf{val}\ aux : \mathbb{R} \bullet aux : \mathbb{R} \bullet k := k + aux$$

. A obrigação de prova é trivial, consistindo apenas em $true \Rightarrow k + aux = k + aux$.

Pode-se ainda usar a lei L.4 para transformar as guardas em alternativas. Dessa forma, o trecho

$$(i = 0\ \&\ |||\ j : \{1..N-1\} \bullet c.j.0?aux \rightarrow updState(aux);$$
$$|||\ j : \{1..N-1\} \bullet c.0.j!k)$$
$$\Box$$
$$(i \neq 0\ \&\ c.i.0!k \rightarrow c.0.i?aux \rightarrow updState(aux))$$

, após a aplicação dessa lei, torna-se

$$(\textbf{if}\ i = 0 \rightarrow |||j : \{1..N-1\} \bullet c.j.0?aux \rightarrow updState(aux);$$
$$|||\ j : \{1..N-1\} \bullet c.0.j!k)$$
$$[]\ i \neq 0 \rightarrow c.i.0!k \rightarrow c.0.i?aux \rightarrow updState(aux)\ \textbf{fi}$$

.

Assim, tem-se a especificação concreta resultante:

$$\textbf{synchronizer}\ \textsc{ReduceSum}\langle N\rangle\ \textbf{where}$$
$$\textbf{channel}\ c : \{0..N-1\} \times \{0..N-1\} \times \mathbb{R}$$
$$\textbf{process}\ reduce \mathrel{\widehat=} \Big|\Big|_{c}\ i : \{0..N-1\} \bullet$$
$$\textbf{begin}$$
$$\textbf{state}\ State \mathrel{\widehat=} [k : \mathbb{R}]$$
$$updState \mathrel{\widehat=} \textbf{val}\ aux : \mathbb{R} \bullet k := k + aux$$
$$\textbf{action}\ reduceSum \mathrel{\widehat=} \textbf{var}\ aux : \mathbb{R} \bullet$$
$$(\textbf{if}\ i = 0 \rightarrow |||j : \{1..N-1\} \bullet c.j.0?aux \rightarrow updState(aux);$$
$$|||\ j : \{1..N-1\} \bullet c.0.j!k)$$
$$[]\ i \neq 0 \rightarrow c.i.0!k \rightarrow c.0.i?aux \rightarrow updState(aux)\ \textbf{fi}$$
$$\bullet\ reduceSum$$
$$\textbf{end}$$

O contrato do componente VecVecProduct faz uso de alguns esquemas próprios da linguagem Z, os quais devem ser eliminados para torná-lo uma especificação concreta.

Primeiramente, considere o esquema computeLocal, o qual inclui o esquema $\Delta State$ e tem como pós-condição

$$r::k' = \sum_{j=0}^{v::dim-1} v::v[j] \times u::v[j].$$

Seja $inv \mathrel{\widehat=} u::dim = v::dim$ e $p \mathrel{\widehat=} r::k' = \sum_{j=0}^{v::dim-1} v::v[j] \times u::v[j]$ e w o conjunto de todas as variáveis definidas pelos esquemas $u::$, $v::$, $r::$ e $i::$. Pela lei de L.1, tem-se que o esquema computeLocal pode ser convertido na instrução de especificação

$$w : [inv \wedge \exists\, u::', v::', r::', i::' \bullet inv' \wedge p, inv' \wedge p]$$

Pode-se notar que a proposição

$$\exists\, u::v', u::dim', v::v', v::dim', r::k', i::k' \bullet inv' \wedge p$$

é verdadeira, bastando para isso escolher valores convenientes para as variáveis. Com isso, é possível reduzir a instrução de especificação para

$w : [inv, inv' \wedge p]$, ou seja, $w : \left[\begin{array}{c} inv, \\ inv' \wedge r::k' = \sum_{j=0}^{v::dim-1} v::v[j] \times u::v[j] \end{array} \right]$

Neste ponto é preciso definir dois processos para dar continuidade ao refinamento. Sejam:

$$I \mathrel{\widehat=} w : [inv, inv' \wedge r::k' = 0] \text{ e}$$

$$A \mathrel{\widehat=} x : \mathbb{N} \bullet w : \left[\begin{array}{l} inv \wedge r::k = \sum_{j=0}^{x-1} v::v[j] \times u::v[j], \\ inv' \wedge r::k' = \sum_{j=0}^{x} v::v[j] \times u::v[j] \end{array} \right]$$

Com $v::dim$ aplicações da lei L.2, tem-se que:

$$w : [inv, inv' \wedge p] = I \fatsemi j : 0..u::dim - 1 \bullet A(j).$$

Basta então refinar os processos I e $A(j)$ para tornar o esquema *computeLocal* em um processo concreto. Para isso, basta fazer uso da lei L.3 por duas vezes e da definição D.1. Aplica-se a lei L.3 para o processo I, refinando-o para $r::k := 0$, deixando uma obrigação de prova trivial. Fazendo-se uso da definição D.1, e então aplicando novamente a lei L.3 no corpo do processo, tem-se que:

$$A(j) \sqsubseteq (r::k := r::k + v::v[x] \times u::v[x])[j/x]$$

A aplicação da lei L.3 deixa a obrigação de prova A.1:

$$r::k = \sum_{j=0}^{x-1} v::v[j] \times u::v[j] \wedge inv$$
$$\Rightarrow$$
$$r::k + v::v[x] \times u::v[x] = \sum_{j=0}^{x} v::v[j] \times u::v[j] \wedge inv$$

Com isso, o esquema *computeLocal* foi refinado até um processo concreto. Foi mostrado que:

$$computeLocal$$
$$\sqsubseteq$$
$$I \fatsemi j : 0..u::dim - 1 \bullet A(j)$$
$$=$$
$$r::k := 0 \fatsemi j : 0..u::dim - 1 \bullet r::k := r::k + v::v[j] \times u::v[j]$$

□

Para finalizar o refinamento do componente, resta apenas refinar o processo *updValue*. Para isso, aplica-se novamente a lei L.3 para refiná-lo para $i::k := r::k$, deixando uma obrigação de prova trivial. Repare que a ação r! não deve ser refinada, pois esta é a ação principal do componente ReduceSum, e, portanto, está encapsulada nesse componente. Dessa forma, tem-se a seguir a especificação concreta do componente VecVecProduct:

computation VecVecProduct$\langle N \rangle$ (u, v, r, i) **where**

inner component u: Vector $\langle N \rangle$
inner component v: Vector $\langle N \rangle$
inner component r: ReduceSum $\langle N \rangle$
inner component i: RealData $\langle N \rangle$

process $product \mathrel{\widehat=} \Big|\Big|\Big| i : \{0..N-1\} \bullet$
begin

slice $u.vector[i]$
slice $v.vector[i]$
slice $r.reduce[i]$
slice $i.real[i]$
state $State \mathrel{\widehat=} [u::, v::, r::, i:: | u::dim = v::dim]$
$computeLocal \mathrel{\widehat=}$
$\quad r::k := 0 \fatsemi j : 0..u::dim - 1 \bullet r::k := r::k + v::v[j] \times u::v[j]$
$updValue \mathrel{\widehat=} i::k := r::k$
$\bullet\ computeLocal;\ r!;\ updValue$

end

O passo de refinamento é importante no processo de derivação do componente, tornando uma especificação abstrata em uma concreta. A especificação concreta é assim chamada porque está pronta para ser traduzida para uma linguagem de programação alvo. No segundo passo do processo de derivação, que é o passo de tradução, será realizada a tradução da especificação concreta para a linguagem C#, a linguagem atualmente suportada pelo HPE para a descrição e implementação de componentes-#.

4.2.2 Tradução da Especificação Concreta

O passo de tradução é o último passo para chegar à implementação de um componente concreto que obedece o contrato a partir do qual foi derivado. Esse passo consiste na

aplicação de regras de tradução sobre a especificação concreta, a fim de alcançar-se uma implementação em uma linguagem de programação desejada.

Nesta seção, será apresentada uma estratégia, dentre as várias possíveis, para tradução da especificação concreta. Será usada como linguagem alvo a linguagem C#, que é a linguagem atualmente suportada pela plataforma HPE.

Na plataforma HPE, um componente concreto consiste em uma classe que herda da classe descrita pelo componente abstrato ao qual ele implementa. A classe implementa métodos do tipo *get* e *set* que permitem acessar os objetos que representam as fatias provenientes das unidades dos componentes aninhados, além de um conjunto de métodos que implementam as interfaces das portas do padrão CCA e requisitadas pelo próprio HPE. A classes que representam as unidades dos componentes das espécies *computação*, *comunicadores* e *aplicação* possuem métodos que representam as suas ações, as quais, como visto anteriormente, devem ser chamadas conforme o protocolo estabelecido para a unidade. Esta seção tratará apenas da construção desses métodos, os quais são definidos pelo programador do componente. Os demais são definidos ou automaticamente gerados pela própria plataforma HPE.

Seja TRAD a função de tradução dos elementos de uma especificação Circus, definida como uma função injetora $TRAD : S_{circ} \rightarrow S_{c\#}$, onde S_{circ} é o conjunto das construções sintáticas da linguagem Circus, e $S_{c\#}$ o conjunto das construções sintáticas relativas a linguagem C#. A função TRAD mapeia um elemento do conjunto S_{circ}, que representa o elemento a ser traduzido, a um elemento do conjunto $S_{c\#}$, que representa a sua tradução. TRAD é uma função de transformação sintática. Por esse motivo, não leva em conta possíveis erros semânticos cometidos na especificação. Para as leis que envolvem a criação de variáveis, as variáveis criadas devem ser novas, pois dessa forma é garantida uma implementação livre de conflitos de nomes de variáveis no mesmo escopo.

Estruturas de Dados

A maioria das estruturas de dados suportadas pela notação Z podem ser traduzidas para tipos da linguagem C#. A linguagem C# possui diversos tipos numéricos, dentre os quais o **ulong**, que representa um inteiro sem sinal cujo valor varia de 0 à 18.446.744.073.709.551.615, o **long**, que representa um inteiro com sinal cujo valor pode variar entre -9.223.372.036.854.775.808 à 9.223.372.036.854.775.807, e o **double**, que representa valores de ponto flutuante capazes de representar uma faixa discreta de números reais no intervalo $[\pm 5,0 \times 10^{-324}, \pm 1,7 \times 10^{+10308}]$. Além disso, a linguagem C# possui tipos básicos para representar caracteres, strings e booleanos.

Dentre os tipos estruturados suportados pela linguagem C#, destacam-se os vetores multidimensionais reais, ou matrizes retangulares, cujos valores são armazenados em endereços de memória consecutivos. O uso de vetores desse tipo é muito comum em

aplicações científicas, mas não é suportado pela linguagem Java, a linguagem orientada a objeto baseada em máquina virtual de uso mais disseminado. Java suporta apenas vetores aninhados (*jagged arrays*), também vulgarmente conhecidos como "*arrays* de *arrays*". Na Seção 4.2.4, são discutidos alguns trabalhos sobre o desempenho de código intensivo no acesso a arrays multidimensionais em linguagens baseadas em máquinas de execução virtual.

A seguir, é definida a tradução de alguns dos tipos numéricos mais usados:

$TRAD(\mathbb{N}) = \textbf{ulong}$
$TRAD(\mathbb{Z}) = \textbf{long}$
$TRAD(\mathbb{Q}) = TRAD(\mathbb{I}) = TRAD(\mathbb{R}) = \textbf{double}$

Além disso, é definida a tradução para o tipo $Array_k$, o açúcar sintático cuja função principal é sinalizar ao tradutor a intenção do programador sobre o uso de um vetor de k dimensões:

$TRAD(Array_k(T)) = TRAD(T)[,\ldots,]$, onde o número de vírgulas corresponde à $k-1$.

Logo, o tipo $Array_k(T)$ é definido como uma função $\mathbb{N} \times \mathbb{N}... \times \mathbb{N} \pfun T$. Caso fosse aplicado diretamente o tipo $\mathbb{N} \times \mathbb{N}... \times \mathbb{N} \pfun T$ com a intenção de definir um vetor multidimensional, o tradutor poderia interpretá-lo como uma função, ao invés de um vetor. Além disso, é possível escolher o tipo de implementação de vetores de acordo com uma máquina de execução virtual alvo a fim de otimizar o desempenho do programa gerado, como descrito na Seção 4.2.4. Na regra apresentada, como trata-se da linguagem C# sobre a plataforma Mono, decidiu-se utilizar matrizes retangulares como exemplo.

Expressões

As expressões em Z são construções que avaliam para um determinado valor. A seguir, TRAD será determinada para algumas das principais expressões presentes na linguagem Z. Seja e uma expressão de qualquer tipo, têm-se que:

$TRAD(e) = e$, se e é constante de algum tipo primitivo, por exemplo, 3.1415 ou a string "formal"
$TRAD((e)) = (TRAD(e))$

Seja e_1 e e_2 expressões de tipos numéricos, têm-se que:

$TRAD({e_1}^n) =$System.Math.Pow$(TRAD(e_1), TRAD(n))$
$TRAD(\sqrt[n]{e_1}) =$System.Math.Pow$(TRAD(e_1), 1.0/(TRAD(n)))$

$$TRAD(\lfloor e \rfloor) = \text{System.Math.Floor}(TRAD(e))$$
$$TRAD(\lceil e \rceil) = \text{System.Math.Ceiling}(TRAD(e))$$
$$TRAD(e_1 + e_2) = e_1 + e_2$$
$$TRAD(e_1 - e_2) = e_1 - e_2$$
$$TRAD(e_1 \times e_2) = e_1 \ * \ e_2$$
$$TRAD(e_1/e_2) = ((\textbf{double})\, TRAD(e_1))/TRAD(e_2)$$
$$TRAD(|\ e\ |) = \text{System.Math.abs}(TRAD(e))$$

Os operadores da notação Z a seguir estão definidos apenas para os tipos $\mathbb{Z}$ e $\mathbb{N}$. Seja $e_1, e_2 \in \mathbb{Z}$ ou $\mathbb{N}$ têm-se que:

$$TRAD(e_1\ div\ e_2) = TRAD(e_1)/TRAD(e_2)$$
$$TRAD(e_1\ mod\ e_2) = TRAD(e_1)\%TRAD(e_2)$$
$$TRAD(max\{e_1, e_2\}) = \text{System.Math.Max}(TRAD(e_1), TRAD(e_2))$$
$$TRAD(min\{e_1, e_2\}) = \text{System.Math.Min}(TRAD(e_1), TRAD(e_2))$$

Sejam e_1 e e_2 expressões de tipos cujos valores podem ser comparados. A função TRAD para expressões de comparação é definida abaixo:

$$TRAD(e_1 = e_2) = TRAD(e_1) == TRAD(e_2)$$
$$TRAD(e_1 > e_2) = TRAD(e_1) > TRAD(e_2)$$
$$TRAD(e_1 < e_2) = TRAD(e_1) < TRAD(e_2)$$
$$TRAD(e_1 \geq e_2) = TRAD(e_1) >= TRAD(e_2)$$
$$TRAD(e_1 \leq e_2) = TRAD(e_1) <= TRAD(e_2)$$
$$TRAD(e_1 \neq e_2) = TRAD(e_1)\ != TRAD(e_2)$$

Sejam e_1 e e_2 comparações, ou seja, expressões que avaliam para verdadeiro ou falso. A função TRAD para expressões booleanas é definida abaixo:

$$TRAD(\neg\ e) = !TRAD(e)$$
$$TRAD(e_1 \wedge e_2) = TRAD(e_1)\ \&\&\ TRAD(e_2)$$
$$TRAD(e_1 \vee e_2) = TRAD(e_1)\ ||\ TRAD(e_2)$$

Construções do Modelo Hash

As construções próprias do Modelo Hash que foram adicionadas na linguagem Circus são traduzidas de acordo com a plataforma de componentes, como o HPE. Seja FirstUpper uma função para auxílio na tradução. Ela recebe um nome de uma variável como argumento e retorna o mesmo nome com a primeira letra em maiúsculo. No caso do HPE, a função TRAD é definida abaixo:

$$TRAD(slice!) = \begin{cases} slice.synchronize() & \texttt{, se slice for da espécie comunicador} \\ slice.compute() & \texttt{, se slice for da espécie computação} \end{cases}$$

$TRAD(slice.action!)$ = slice.action(), onde *slice* é o nome de uma fatia e *action* é o nome de uma ação dessa fatia.
$TRAD(slice :: v)$ = slice.FirstUpper(v)
$TRAD(i) = getRank()$, onde i representa o ranque de uma unidade em uma unidade paralela
$TRAD(N) = getSize()$, onde N representa o número de unidades de uma unidade paralela

CSP

As construções CSP tratam basicamente da sincronização e comunicação entre unidades. Para implementar tais construções, faz-se uso do MPI para comunicação em memória distribuída e da API de *threads* da própria linguagem C#. O operador de intercalação ($|||$) é implementado com a classe *ThreadPool* e a classe *WaitHandle*, se ele aparece dentro de algum processo. Um reservatório de *threads*, implementada pela classe *ThreadPool* do C#, é a técnica de se reutilizar *threads* afim de se evitar a sobrecarga na criação de novas *threads*. Assim, se uma *threads* acaba de realizar uma determinada tarefa, ela é posta para dormir, até que esteja disponível outra tarefa para ser realizada. Se o operador de intercalação aparece indicando a intercalação das unidades, o próprio HPE se encarrega de tratá-la. Não é tratado neste trabalho o operador de composição paralela ($||$) no caso de ser usado dentro de uma ação, pois sua implementação é demasiadamente complicada, envolvendo threads e o MPI, e ter pouco uso levando-se em consideração a pouca utilidade que se tem em usar o paradigma de passagem de mensagens entre *threads*, além disso o HPE já se encarrega de fazer o paralelismo entre unidades.

Na tradução de comunicações, utilizam-se comunicadores do MPI para representar canais de comunicação, de modo que a cada canal de comunicação é associado um objeto comunicador do MPI.NET de mesmo nome do canal. Para a construção *SKIP* não há tradução, pois ele não representa nenhuma ação, apenas que a operação atual terminou com sucesso, muitas vezes pode ser traduzido por return. A função TRAD para as construções CSP é definida abaixo:

$TRAD(STOP)$ = Thread.Sleep(Timeout.Infinite);
$TRAD(CHAOS)$ = Environment.Exit(1);
$TRAD(comm \rightarrow P) = TRAD(comm)TRAD(P)$
$TRAD(c.i.j!x)$ = c.Send(x,j,0);
$TRAD(c.i.j?x) = x$ = c.Receive<$TRAD(T)$>(i,0);, onde T é o tipo da variável x
$TRAD(\mu F \bullet P) =$

```
Func F = null;
F = () => (TRAD(P));
F;
```

$TRAD(|||\ i : N1..N2 \bullet P) =$

```
ManualResetEvent[] doneEvents = new ManualResetEvent[N2-N1+1];
for (int counter = N1; counter <= N2; counter++){
    int i = counter;
    doneEvents[i-N1] = new ManualResetEvent(false);
    ThreadPool.QueueUserWorkItem (() => TRAD(P) doneEvents[i-N1].Set(););
}
WaitHandle.WaitAll(doneEvents);
```

$TRAD(g_1 \ \& \ c_1 \rightarrow P_1 \ \Box \ \ldots \ \Box \ g_n \ \& \ c_n \rightarrow P_n) =$

```
int idChoice = -1;
Object lockT = new Object();
ManualResetEvent[] doneEventsAux = new ManualResetEvent[n];
for (int counter = 0; i <= n-1; counter++){
    int i = counter;
    doneEvents[i] = new ManualResetEvent(false);
    if (!(TRAD(g_i)))
        continue;
    ThreadPool.QueueUserWorkItem (() =>
                    TRAD(c_i)
                    lock (lockT){
                        if (idChoice != -1)
                            TRAD(STOP)
                        idChoice=i;
                    }
                    TRAD(P_i))
                    doneEventsAux[i].Set();
                );
}
while(idChoice == -1);
ManualResetEvent[] doneEvents = new ManualResetEvent[1];
doneEvents[0] = doneEventsAux[idChoice];
WaitHandle.WaitAny(doneEvents);
```

A Linguagem de Comandos Guardados

A GCL é aquela que mais se aproxima de uma linguagem de programação, comparada ao CSP e ao Z. Por esse motivo, a tradução das suas construções é bastante simples. Para

as construções pertencentes à GCL, a função TRAD é definida abaixo:

$TRAD(v) = v$, se v é um identificador

$TRAD(\textbf{if } g_1 \rightarrow P_1 \ldots [] \ g_n \rightarrow P_n \textbf{ fi}) =$

```
if (TRAD(g1)){TRAD((P1)}
else   if (TRAD(g2)){TRAD(P2)}
⋮
else   if (TRAD(gn)){TRAD(Pn)}
else   {TRAD(CHAOS)}
```

$TRAD(v := e) = TRAD(v) = TRAD(e);$
$TRAD(Func \mathrel{\widehat=} \textbf{val } x : T \bullet P) = \textbf{void } Func(TRAD(T)\ x)\{TRAD(P)\}$
$TRAD(Func \mathrel{\widehat=} \textbf{res } x : T \bullet P) = \textbf{void } Func(\textbf{out } TRAD(T)\ x)\{TRAD(P)\}$
$TRAD(Func \mathrel{\widehat=} \textbf{ vres } x : T \bullet P) = \textbf{void } Func(\textbf{ref } TRAD(T)\ x)\{TRAD(P)\}$
$TRAD(Func \mathrel{\widehat=} \textbf{var } x : T \bullet P) = \textbf{void } Func()\{TRAD(T)\ x;\ TRAD(P)\}$
$TRAD(⨟ i : ini..fim \bullet P) = \textbf{for}(\textbf{long } i = ini;\ i <= fim;\ \text{i++})\{TRAD(P)\}$
$TRAD(P) = P();$, dado que P é uma chamada a uma função sem parâmetros.
$TRAD(P(i)) = P(i);$
$TRAD(P;\ P') = TRAD(P)\ TRAD(P')$

Como exemplo, os componentes ReduceSum e VecVecProduct serão traduzidos de acordo com as leis apresentadas. A tradução do componente concreto ReduceSum é mostrada abaixo:

```
void updState(double aux){
    k = k + aux;
}

void reduceSum(){
    double aux;
    if (getRank() == 0){
        c.Send(k,0,0);
        aux = c.Receive<double>(0,0);
        updState(aux);
    }
    else if (getRank() != 0){
        ManualResetEvent[] doneEvents = new ManualResetEvent[getSize()-1-1+1];
```

```
        for (int counter = 1; counter <= getSize()-1; counter++){
            int j = counter;
            doneEvents[j-1] = new ManualResetEvent(false);
            ThreadPool.QueueUserWorkItem (() =>
                          aux = c.Receive<double>(j,0);
                          updState(aux);
                          doneEvents[j-1].Set();
                        );
        }
        WaitHandle.WaitAll(doneEvents);

        ManualResetEvent[] doneEvents = new ManualResetEvent[getSize()-1-1+1];
        for (int counter = 1; counter <= getSize()-1; counter++){
            int j = counter;
            doneEvents[j-1] = new ManualResetEvent(false);
            ThreadPool.QueueUserWorkItem (() =>
                                    c.Send(k,j,0);
                                    doneEvents[j-1].Set();
                                );
        }
        WaitHandle.WaitAll(doneEvents);
    }
    else{
        Environment.Exit(1);
    }
}

public void synchronize(){
    reduceSum();
}
```

A tradução do componente VecVecProduct é apresentada abaixo:

```
void computeLocal(){
    r.K = 0;
    for(long j=0;j<=u.Dim-1;j++){
        r.K = r.K + v.V[j] * u.V[j];
    }
}
```

```
void updValue(){
    i.K = r.K;
}

public void compute(){
    computeLocal();
    r.synchronize();
    updvalue();
}
```

Após o passo de tradução, obtém-se a implementação de um componente concreto que implementa o contrato definido pelo seu componente abstrato, pronta para ser compilada usando o HPE.

4.2.3 Código Objeto de um Componente

Após passar pelo processo de derivação, a implementação de um componente concreto precisa ser compilada, tornando-se um código em formato binário executável. Porém, é necessário que seja possível a verificação de propriedades de componentes após a fase de compilação. Sugere-se então embutir a especificação concreta de um componente em seu código objeto, de modo que será possível a verificação de propriedades e de satisfação a um contexto de implementação.

A incorporação da especificação deve ser feita pela nuvem, após a implantação do componente. Assim, quando um usuário for implantar um componente na nuvem, ele tem a opção de, junto ao componente, fornecer a especificação do mesmo. Dessa forma, a nuvem pode fazer a incorporação da especificação no código objeto do componente da forma que desejar. Técnicas para embutir especificações e informações sobre a derivação formal de programas em códigos executáveis são bem conhecidas [56, 93].

4.2.4 Sobre Otimização do Desempenho de Programas em Máquinas Virtuais

No contexto do grupo de pesquisa do qual faz parte o autor desta dissertação, um estudo recente de avaliação de desempenho com máquinas de execução virtual dos padrões JVM (*Java Virtual Machine*) e CLI (*Common Language Infrastructure*) fornece evidências sobre a relevância da contribuição relativa à proposta de açúcares sintáticos para vetores multidimensionais proposta nesta dissertação [5].

Apesar de matrizes retangulares terem sido propostas pelo padrão CLI e suportadas pela linguagem C# com o objetivo de atender aos requisitos de programas com acesso intensivo a vetores multidimensionais, normalmente encontrados em aplicações de computação científica com requisitos de CAD, o estudo demonstra que seu desempenho não

é melhor do que o uso alternativo de vetores aninhados, tanto comparando-se o desempenho de vetores aninhados em máquinas do padrão JVM com o desempenho de matrizes retangulares em máquinas do padrão CLI, quanto comparando-se as duas alternativas em uma mesma máquina do padrão CLI. Isso é devido ao fato de que as otimizações hoje existentes em compiladores JIT (*Just-in-Time*) de ambos os padrões privilegiam código que faz uso de vetores aninhados, contexto que pode mudar na medida em que otimizações específicas para código baseado em matrizes retangulares sejam implementados em compiladores JIT de máquinas virtuais do padrão CLI.

Ainda a respeito das otimizações em compiladores JIT de máquinas virtuais, elas exigem a aplicação de certas recomendações de estruturação de código a fim de serem melhor aproveitadas, uma preocupação bastante conhecida de programadores de computação científica usando linguagens nativas, como C e Fortran. No entanto, além de tais recomendações não serem as mesmas da programação com linguagens nativas, são distintas entre máquinas virtuais, do mesmo padrão.

Por exemplo, para obter-se um melhor desempenho do compilador JIT da implementação do padrão JVM oferecido pela Oracle [100], também usada na OpenJDK [99], recomenda-se que o código esteja o máximo possível fatorado em classes e métodos, o que contradiz o senso comum usado em programação para código nativo de não encapsular trechos de código mais executados em uma computação intensiva dentro de subrotinas, a fim de evitar a sobrecarga da sua invocação. Esse comportamento foi primeiramente observado em trabalhos de avaliação de máquinas de execução virtual de outros autores [11, 119]. Surpreendentemente, esses trabalhos mostram inclusive que se essas recomendações forem cuidadosamente aplicadas e levadas ao extremo, o que contradiz o fato de estar-se trabalhando com uma linguagem com alto nível de abstração, o desempenho bruto do programa em execução virtual pode ser semelhante a sua versão em execução nativa. Porém, o trabalho realizado por nosso grupo de pesquisa mostrou que essa recomendação não é válida para o compilador JIT usado pela implementação do padrão JVM oferecido pela IBM [70, 71], pois este causa alta sobrecarga no desempenho.

Tendo em vista os fatos relacionados, a maioria das escolhas relacionadas à forma de estruturação de código científico com vistas a explorar ao máximo o desempenho de máquinas de execução virtual depende das peculiares da implementação do compilador JIT. Dessa forma, em um ambiente de derivação de programas poderiam existir vários *planos de tradução*, cada qual incluindo um conjunto de regras para tradução da especificação concreta de acordo com uma máquina de execução alvo específica, ou mesmo a possibilidade de gerar código apropriado para mais de uma máquina a partir de uma especificação mais abstrata, dentre os quais poderia ser escolhido o mais adequado para um certo ambiente de execução. Essa é uma proposta de trabalho de pesquisa que poderia ser aplicada

futuramente.

4.3 Certificação de Componentes

A certificação de componentes consiste na verificação se um componente cumpre um determinado contrato, de forma a garantir que tal componente se encaixa no contexto definido pelo contrato.

Para que seja possível comparar um contrato com a especificação concreta de um componente, ambas descritas em Circus/HCL, é necessária a transformação de ambas as especificações para a linguagem Circus, removendo-se quaisquer açúcares sintáticos. O motivo disso é o fato de que os componentes descritos em Circus/HCL não tem semântica válida na UTP (*Unified Theory of Processes*), mencionado na Seção 2.4. Portanto, sem isso não seria possível saber se um componente satisfaz um determinado contrato.

Após a especificação Circus/HCL ser transformada em uma especificação Circus é possível verificar se o contrato é satisfeito pelo componente. Para isso, pode-se usar ferramentas como o ProofPower-Z, para ajudar na automatização do processo de verificação.

4.3.1 Transformação do Circus/HCL para o Circus

Uma diferença importante das linguagens Circus/HCL e Circus é o fato de que na linguagem Circus/HCL é permitido que uma unidade altere o estado de uma de suas fatias enquanto no Circus, os estados dos processos, que representam as unidades de um componente, estão encapsulados nos próprios processos, sendo, assim, impossível para um processo alterar o estado de outro diretamente.

Por essa razão, um componente na linguagem Circus/HCL, quando transformado para a linguagem Circus, as unidades se tornam processos cuja especificação desdobra as especificações de todas as suas fatias, inclusive os seus estados, de forma que o estado da unidade transformada é a união dos estados de todas as suas fatias, juntamente com o estado da própria unidade a ser transformada. Dessa forma, as unidades, na linguagem Circus, poderão fazer alterações nos estados de qualquer fatia que possua. Por sua vez, o componente em si se torna um processo que descreve o comportamento de suas unidades através do operador de composição paralela ou intercalação, dependendo se há ou não comunicação entre as unidades.

O processo transformado que representa o componente em si será:

$$\Big\|_{channelSet} \; i : 0..N-1 \bullet nome\_da\_unidade(i)$$

O conjunto *channelSet* consiste no conjunto de todos os canais declarados nos componentes aninhados, assim como o do próprio componente.

O processo que representará as unidades do componente a ser transformado leva o nome da própria unidade que está sendo transformada. Da mesma forma o nome do processo que descreve o comportamento das suas unidades, leva o nome do componente em si. Todos os canais usados no componente e nos seus componentes aninhados são declarados, onde o nome de cada canal ganha um prefixo que consiste no nome da fatia da qual faz parte seguido de _.

O estado da unidade, como dito anteriormente, declara todas as variáveis de todas as suas fatias, onde cada variável ganha um prefixo que consiste no nome da fatia seguido de ::. O tipo de cada variável se mantêm igual ao tipo original da variável. Cada pré-condição contida nos estados das fatias da unidade também são levados para a especificação Circus da unidade.

Para cada fatia de uma unidade, também são declarados todos os parágrafos de ações que fazem parte da unidade do componente aninhado a qual ela representa. O nome do novo parágrafo ganhará o prefixo que consiste no nome da fatia seguido de _. O nome do parágrafo de uma ação será o nome da fatia seguida de _ seguida da ação. Os parágrafos de ações pertencentes à própria unidade que está sendo transformada não sofrem alteração de nomes.

Cada processo Circus que representa uma unidade do componente a ser transformado terá por parâmetro uma variável cujo valor será dado pelo processo que representa o componente. Ela representará o valor do enumerador para a unidade e suas fatias. Essa variável terá por nome o mesmo nome do enumerador apresentado na especificação Circus/HCL e será do tipo natural. Será feita uma definição axiomática na especificação Circus, que definirá a constante que representa o número de unidades enumeradas, e terá por nome o mesmo nome declarado na especificação Circus/HCL, ela será do tipo natural e a sua única restrição é ser maior que zero.

Quando se deseja transformar um contrato, deve-se usar o próprio contrato, juntamente com os contratos dos componentes aninhados. Já quando se deseja transformar um componente concreto, deve-se usar a especificação concreta deste componente, e os contratos dos componentes aninhados. Em transformações, jamais se deve usar especificações concretas de componentes aninhados.

Como exemplo, a transformação do **contrato** VecVecProduct leva a seguinte especificação Circus:

$$
\begin{array}{l}
\mid N : \mathbb{N} \\
N > 0 \\
\textbf{channel}\ r\_c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\textbf{process}\ product \mathrel{\widehat=} i : \mathbb{N} \bullet
\end{array}
$$

begin

$$\textbf{state}\ State \mathrel{\widehat=} [i::k : \mathbb{N}, r::k : \mathbb{N}, u::dim : \mathbb{N}, u::v : \mathbb{N} \pfun \mathbb{R}, v::dim : \mathbb{N}, v::v : \mathbb{N} \pfun \mathbb{R} \mid u::dim = v::dim]$$

$$r\_updState \mathrel{\widehat=} \textbf{val}\ aux : \mathbb{R} \bullet [\Delta State, aux? : \mathbb{R} \mid r::k' = r::k + aux]$$

$$r\_reduceSum \mathrel{\widehat=} \textbf{var}\ aux : \mathbb{R} \bullet$$

$$(i = 0\ \&\ |||\, j : 1..N-1 \bullet r\_c.j.0?aux \rightarrow r\_updState(aux);$$
$$|||\, j : 1..N-1 \bullet r\_c.0.j!r::k)$$
$$\Box$$
$$(i \neq 0\ \&\ r\_c.i.0!r::k \rightarrow r\_c.0.i?aux \rightarrow r\_updState(aux))$$

$$r\_MainAction \mathrel{\widehat=} r\_reduceSum$$

$$computeLocal \mathrel{\widehat=} [\Delta State \mid r::k' = \textstyle\sum_{j=0}^{v::dim-1} v::v(j) \times u::v(j)]$$

$$updValue \mathrel{\widehat=} [\Delta State \mid i::k' = r::k]$$

$$\bullet\ computeLocal;\ r\_MainAction;\ updValue$$

end

$$\textbf{process}\ VecVecProduct \mathrel{\widehat=} \Big|\Big|_{r\_c}\ i : 0..N-1 \bullet product(i)$$

4.3.2 Certificação de Componentes

A certificação de componentes consiste em verificar se um componente se encaixa em um determinado contexto. Os contextos são definidos por contratos que são especificações abstratas escritas na linguagem Circus/HCL. Os contratos podem ser vistos como extensões dos componentes abstratos no HPE, detalhando as funcionalidades, pré-condições e pós-condições dos componentes que desejarem implementá-lo.

A verificação só pode ser feita após a transformação, para a linguagem Circus, tanto da especificação concreta, representando o componente, quanto da especificação abstrata, que representa o contrato.

A linguagem Circus tem sua semântica baseada na UTP. Dessa forma, todo artefato pertencente a essa linguagem tem o seu significado traduzido para uma fórmula lógica, utilizando as variáveis próprias da UTP e, se for o caso, as variáveis relativas à operação ou ao processo em questão. Dessa forma, toda especificação de componente tem sua semântica descrita através da lógica de primeira ordem.

Para verificar se um componente, representado pela especificação concreta E, implementa um contrato C, basta provar que:

$C \sqsubseteq E$, ou seja:
$[\exists\, E.State;\ E.State' \bullet E.Act] \Rightarrow [\exists\, C.State;\ C.State' \bullet C.Act]$

As ações $E.Act$ e $C.Act$ podem agir em espaços de estados diferentes e, por isso, os estados podem não ser comparáveis. Por esse motivo apenas as ações principais das especificações são comparadas, escondendo os estados de cada uma, como se os estados fossem blocos locais de variáveis, e, como tal, sua semântica é dada pelo quantificador existencial.

O processo de se obter a semântica de um processo pode se tornar um trabalho longo e cansativo, pois para cada operador CSP ou esquema Z ou ainda instrução na GCL tem-se, como significado, uma fórmula na lógica de primeira ordem. Por isso, os processos podem se tornar em fórmulas bastante grandes, e o processo de prova da implicação, nesses casos, se torna uma tarefa extensa e cansativa. Porém existem ferramentas para ajudar na prova semi-automática de teoremas, onde se fornece a proposição que se deseja provar e, depois de alguns passos, alguns deles manuais, se chega à prova da fórmula proposta.

A ferramenta ProofPower-Z [81], é uma extensão do provador de teoremas ProofPower [15], que incorpora a teoria da linguagem Z. Em [131], é mostrado como codificar processos Circus na linguagem Z a fim de que se possa usá-los no ProofPower-Z para provar leis de refinamento e propriedades de processos de forma semi-automática. No caso da certificação de componentes, codificaria-se tanto o componente que deseja-se certificar quanto o contrato o qual deseja-se saber se o componente implementa, ambos descritos em Circus. Após isso, a ferramenta se encarregaria, possivelmente com intervenção humana, de provar a implicação.

Capítulo 5

Estudos de Caso e Prova de Conceito

Para prova de conceito sobre o processo de derivação e certificação de componentes que será proposto como resultado da dissertação, propomos a utilização de aplicações do *NAS Parallel Benchmarks* (NPB) [17], o qual consiste em um conjunto de *benchmarks* e aplicações simuladas para medir o desempenho de computadores paralelos.

A metodologia proposta consiste em especificar alguns componentes da versão baseada em componentes do NPB, a partir da descrição "papel-e-caneta"[1] do programa que encontra-se na documentação do pacote [19]. A partir dessa especificação, usaremos o processo de derivação para obter implementações concretas dos componentes, as quais serão comparados com os componentes desenvolvidos manualmente para a plataforma HPE [108]. Com isso, desejamos caracterizar as limitações da derivação de código de simulação numérica, bastante comum em aplicações CAD, desenvolvendo alternativas no processo de tradução para gerar código eficiente.

A seguir, um descrição mais detalhada sobre o pacote NPB.

5.1 NAS Parallel Benchmarks (NPB)

O NPB é um conjunto de aplicações de *benchmark* desenvolvidas pela divisão da NASA chamada *NASA Advanced Supercomputing* (NAS). Ele foi desenvolvido para a avaliação de computadores paralelos segundo os requisitos da divisão NAS. Suas três versões foram publicadas respectivamente nos anos de 1991, 1996 e 2003. Atualmente, essas versões contemplam implementações em várias linguagens de programação, tais como C, Fortran e Java, utilizando tanto o modelo de passagem de mensagens, através da utilização do MPI, quanto do modelo de memória compartilhada, através da utilização do OpenMP e Java Threads.

Atualmente o NPB se encontra na versão 3.3, consistindo de 11 programas,

[1] A própria literatura refere-se a tais especificações como "pencil-and-paper".

dentre aplicações simuladas e *benchmarks*. Dentre esses, existem versões baseadas em componentes para o HPE para 3 aplicações simuladas (SP, BT e LU) e para um benchmark (FT) [92, 108], como descrito na Seção 3.3.3. Neste trabalho, aplicaremos o processo de derivação proposto para 2 benchmarks. São estes:

- IS (*Integer Sort*): Ordenação paralela de um conjunto de inteiros utilizando o algoritmo *bucketsort*;
- CG (*Conjugate Gradient*): Usa o método das potências para encontrar o maior autovalor de uma matriz simétrica esparsa.

Cada *benchmark* define sete cargas de trabalhos pré-definidas, chamadas *classes de problemas*, para problema a ser resolvido, definido por parâmetros de entrada particulares de cada programa. As classes são denominadas S, W, A, B, C, D e E, em ordem crescente de tamanho de carga de trabalho.

O uso do NPB é justificado por serem programas já prontos e típicos da área de computação científica, profissionalmente desenvolvidos por especialistas no assunto. Escolhemos IS e CG devido a sua simplicidade e serem bastante utilizadas como "componentes" em aplicações reais de CAD, representando uma computação numérica sobre números inteiros, intensiva em movimentação de memória e operações de comunicação coletiva, e uma computação numérica sobre números de pontos flutuantes, intensiva em operações coletivas implementadas sobre operações de comunicação ponto-a-ponto.

O estudo de caso se dará em quatro etapas. Primeiramente, especificaremos componentes computacionalmente significativos das duas aplicações na linguagem Circus/HCL, de forma a preservar as propriedades originais das aplicações. Após a especificação desses componentes, aplicaremos a primeira etapa do processo proposto, que é a derivação dos componentes-# a partir das especificações. Enfim, utilizaremos as especificações das aplicações para fazermos verificações quanto as suas propriedades e sabermos se a sua especificação concreta de fato obedece ao seu contrato. Esperamos ainda identificar problemas a serem tratados em futuros trabalhos para viabilizar essa técnica para uso real na nuvem de componentes.

5.2 Integer Sort (IS)

O kernel Integer Sort (IS) consiste na operação básica de ordenação de chaves inteiras em paralelo, cujos valores não excedem um determinado valor $Bmax$. Ele utiliza o algoritmo *bucket sort* para fazer a ordenação paralela das chaves. Tal valor depende da classe de carga de trabalho utilizada. Por exemplo, se a classe de carga de trabalho for A então

$Bmax = 2^{19}$. Na prática, por se tratar de uma operação elementar, ele é bastante utilizado em aplicações de alto desempenho.

Uma sequência de chaves $\{k_0, k_2, \ldots, k_{N-1}\}$ é dita ordenada se, para todo $1 \leq i \leq N-1 <$ tem-se que $k_{i-1} \leq k_i$. O ranque de uma chave é o índice que essa chave teria se a sequência estivesse ordenada. O componente IntegerSort implementa o *bucket sort* paralelo, achando os ranques de todas as chaves.

Abaixo, o processo de derivação de componentes descrito na Seção 4.2 é aplicado ao componente IntegerSort. Relembrando os passos, inicialmente é descrito o seu contrato através da linguagem Circus/HCL. Após isso, o contrato é refinado para se tornar uma especificação concreta. Finalmente, ocorre a tradução da especificação concreta para a linguagem C#. Para o processo de verificação, tanto o contrato quanto a especificação concreta são traduzidas para a linguagem Circus. Dessa forma, elas podem ser usadas para a verificação se um componente satisfaz o contrato proposto.

5.2.1 Contrato

Na Figura 5.1 é apresentado o contrato do componente IntegerSort, baseado na descrição "caneta-e-papel" do *kernel* IS [17].

O contrato do componente IntegerSort descreve o método de ordenação *bucket sort* paralelo para ordenar os valores, onde cada processo representa um "balde" (*bucket*) onde serão jogados as chaves dentro de um certo intervalo determinado. Define uma única unidade paralela, de nome *sorter*, cujo estado inclui as seguintes variáveis:

- **k**: Guarda as chaves sorteadas para a unidade. Deve ser inicializada antes da execução do componente;
- **Np**: Número de chaves sorteadas para a unidade. Supõe-se inicializadas antes da execução do componente;
- **ord**: O valor do índice i representa a quantidade de elementos do i-ésimo "balde" (*bucket*) presentes em k;
- **rank**: O valor do índice i representa o ranque da primeira chave de valor i se o valor de $ord[i]$ for diferente de zero;
- b_0 e b_1: Armazena valor do primeiro e último valores os quais os ranques devem ser computados pelo balde.

A unidade *sorter* especifica ainda dois parágrafos de ação. O parágrafo *init* é responsável pela especificação da inicialização do estado do componente, atribuindo o valor zero a todos os elementos do *array ord*, que guardará os valores das ordens dos elementos para cada balde, bem como atribuindo os valores corretos para as variáveis b_0 e b_1.

computation IntegerSort$\langle N \rangle$ (u, v, r, i) **where**
$| \; Bmax : \mathbb{N}$
channel $c : \mathbb{N} \times \mathbb{N} \times \mathbb{R}$

process $sorter \mathrel{\widehat=} \Big|\Big|_{c} \; i : \{0..N-1\} \bullet$
begin

state $State \mathrel{\widehat=} [k{:}Array_1(\mathbb{Z}), Np{:}\mathbb{N}, ord{:}Array_1(\mathbb{Z}), rank{:}Array_1(\mathbb{N}), b_0{:}\mathbb{N}, b_1{:}\mathbb{N}]$

$init \mathrel{\widehat=} [State' \mid \forall j : 0..Bmax \bullet ord[j]' = 0 \land$
$b_0' = i \times (\lfloor \frac{Bmax}{N} \rfloor + 1) \land b_1' = (i+1) \times (\lfloor \frac{Bmax}{N} \rfloor) + i]$

action $rank \mathrel{\widehat=}$ **var** $bucket, aux : \mathbb{N} \bullet$

$\mathop{⨟}_{9} j : 0..i-1 \bullet \mu P \bullet c.j.i?aux \rightarrow$ **if** $aux = -1 \rightarrow SKIP$
$[] aux \neq -1 \rightarrow ord[aux] := ord[aux] + 1; \; P$
fi;

$\mathop{⨟}_{9} j : 0..Np-1 \bullet bucket{:}[bucket' = \lfloor \frac{k[j]}{\lfloor \frac{Bmax}{N} \rfloor + 1} \rfloor]$;
if $bucket \neq i \rightarrow c.i.bucket!k[j]$
$[] bucket = i \rightarrow ord[k[j]] := ord[k[j]] + 1$
fi;

$\mathop{⨟}_{9} j : 0..i-1 \bullet c.i.j! - 1 \rightarrow SKIP$;
$\mathop{⨟}_{9} j : i+1..N-1 \bullet c.i.j! - 1 \rightarrow SKIP$;

$\mathop{⨟}_{9} j : i+1..N-1 \bullet \mu P \bullet c.j.i?aux \rightarrow$ **if** $aux = -1 \rightarrow SKIP$
$[] aux \neq -1 \rightarrow ord[aux] := ord[aux] + 1; \; P$
fi;

if $i = 0 \rightarrow rank{:}[\forall j : b_0..b_1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + 1];$
$c.i.(i+1)!(rank[b_1] + ord[b_1]) \rightarrow SKIP$
$[] (i > 0) \land (i < N-1) \rightarrow c.(i-1).i?aux \rightarrow rank{:}[\forall j : b_0..b_1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux];$
$c.i.(i+1)!(rank[b_1] + ord[b_1]) \rightarrow SKIP$
$[] i = N-1 \rightarrow c.(i-1).i?aux \rightarrow rank{:}[\forall j : b_0..b_1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux]$
fi

$\bullet \; init; \; rank$

end

Figura 5.1: Contrato Circus/HCL do Componente IntegerSort

O parágrafo *rank* especifica o procedimento de ranqueamento das chaves. Primeiramente, cada unidade redistribui as chaves que não estão entre b_0 e b_1, e recebe chaves de outras unidades, computando-se o vetor *ord* enquanto isso. Após essa etapa, todas as unidades estão com as suas chaves, e então computa-se o ranque de cada chave, baseado no vetor *ord*.

5.2.2 Refinamento

Para que o contrato do componente IntegerSort seja refinado, é preciso que sejam refinadas todas as instruções de especificação e esquemas na linguagem Z. São elas:

i. O parágrafo de ação *init*, que corresponde ao esquema

$$[State' \mid \forall j : 0..Bmax \bullet ord[j]' = 0 \land b_0' = i \times \left(\lfloor \frac{Bmax}{N} \rfloor + 1\right) \land b_1' = (i+1) \times \left(\lfloor \frac{Bmax}{N} \rfloor\right) + i]$$

ii. A instrução de especificação

$$bucket : [bucket' = \lfloor \frac{k[j]}{\lfloor \frac{Bmax}{N} \rfloor + 1} \rfloor]$$

iii. A instrução de especificação

$$rank : [\forall j : b_0..b_1 \bullet rank[j]' = \left(\sum_{l=b_0}^{j-1} ord[l]\right) + aux]$$

iv. A instrução de especificação

$$rank : [\forall j : b_0..b_1 \bullet rank[j]' = \left(\sum_{l=b_0}^{j-1} ord[l]\right) + 1]$$

Para refinar o primeiro item, primeiramente será aplicada a lei L.1 para que o esquema Z seja convertido em uma instrução de especificação. Assim, sejam

- $p_1 \equiv \forall j : 0..Bmax \bullet ord[j]' = 0$,
- $p_2 \equiv b_0' = i \times (\lfloor \frac{Bmax}{N} \rfloor + 1)$ e
- $p_3 \equiv b_1' = (i+1) \times (\lfloor \frac{Bmax}{N} \rfloor) + i$.

Após a aplicação da lei, tem-se a instrução de especificação

$$ord, b_0, b_1 : \left[\begin{array}{c} \exists\, ord', b_0', b_1' \bullet p_1 \land p_2 \land p_3, \\ p_1 \land p_2 \land p_3 \end{array}\right].$$

Aplicando-se a lei L.2, tem-se

$$ord, b_0, b_1 : \left[\begin{array}{c} \exists\, ord', b_0', b_1' \bullet p_1 \land p_2 \land p_3, \\ p_2 \land p_3 \end{array}\right] ;\ ord, b_0, b_1 : \left[\begin{array}{c} p_2[\_/'] \land p_3[\_/'], \\ p_1 \land p_2 \land p_3 \end{array}\right].$$

Aplicando-se a lei L.3 na primeira instrução de especificação, tem-se

$$b_0, b_1 := i \times (\lfloor Bmax/N \rfloor + 1), (i+1) \times (\lfloor Bmax/N \rfloor) + i$$

o que leva à obrigação de prova trivial:

$$(\exists\, ord', b_0', b_1' \bullet p_1 \wedge p_2 \wedge p_3) \Rightarrow ((i \times (\lfloor Bmax/N \rfloor + 1) = i \times (\lfloor \tfrac{Bmax}{N} \rfloor + 1)) \wedge \\ ((i+1) \times (\lfloor Bmax/N \rfloor) + i = (i+1) \times (\lfloor \tfrac{Bmax}{N} \rfloor) + i)).$$

Aplicando $Bmax$ vezes a lei L.2 na segunda instrução de especificação, tem-se

$$\S\, j : 0..Bmax \bullet : \left[\begin{array}{c} p_2[\_/'] \wedge p_3[\_/'] \wedge \forall\, l : 0..(j-1) \bullet ord[l] = 0, \\ p_2 \wedge p_3 \wedge \forall\, l : 0..j \bullet ord[l]' = 0 \end{array} \right].$$

Aplicando a lei L.3 na instrução de especificação, tem-se $\S\ j : 0..Bmax \bullet ord[j] := 0$, deixando a obrigação de prova OP.2. Com isso tem-se, finalmente, que

$$b_0, b_1 := i \times (\lfloor Bmax/N \rfloor + 1), (i+1) \times (\lfloor Bmax/N \rfloor) + i;\ \S\, j : 0..Bmax \bullet ord[j] := 0.$$

Para refinar a instrução de especificação do segundo item, será inicialmente usada a lei L.3. Dessa forma, a instrução de especificação é refinada para a atribuição $bucket := \lfloor k[j]/(\lfloor Bmax/N \rfloor + 1) \rfloor$, deixando uma obrigação de prova trivial: $true \Rightarrow \lfloor k[j]/(\lfloor Bmax/N \rfloor + 1) \rfloor = \lfloor \frac{k[j]}{\lfloor \frac{Bmax}{N} \rfloor + 1} \rfloor$. Claramente, se vê que a expressão $\lfloor k[j]/(\lfloor Bmax/N \rfloor + 1) \rfloor$ tem o mesmo valor da expressão $\lfloor \frac{k[j]}{\lfloor \frac{Bmax}{N} \rfloor + 1} \rfloor$. Com isso, tem-se que $bucket := \lfloor k[j]/(\lfloor Bmax/N \rfloor + 1) \rfloor$.

No caso da instrução de especificação no terceiro item, aplica-se $(b_1 - b_0)$ vezes a lei L.2, obtendo-se

$$\S\, m : b_0..b_1 \bullet rank : \left[\begin{array}{c} \forall\, j : b_0..(m-1) \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux, \\ \forall\, j : b_0..m \bullet rank[j]' = (\sum_{l=b_0}^{j-1} ord[l]) + aux \end{array} \right]$$

, que é equivalente a

$$rank : [true, rank[b_0]' = aux]; \\ \S\, m : b_0 + 1..b_1 \bullet rank : \left[\begin{array}{c} \forall\, j : b_0..m-1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux, \\ \forall\, j : b_0..m \bullet rank[j]' = (\sum_{l=b_0}^{j-1} ord[l]) + aux \end{array} \right].$$

Aplicando-se a lei L.3 na instrução de especificação $rank : [true, rank[b_0]' = aux]$, obtém-se $rank[b_0] := aux$, deixando uma obrigação de prova trivial: $true \Rightarrow (aux = aux)$.

$$
\begin{array}{l}
\textbf{computation}\ \textsc{IntegerSort}\langle N\rangle\ (u, v, r, i)\ \textbf{where} \\
\mid Bmax : \mathbb{N} \\
\textbf{channel}\ c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\textbf{process}\ sorter \mathrel{\widehat=} \|_{c}\ i : \{0..N-1\} \bullet \\
\textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat=} [k{:}Array_1(\mathbb{Z}), Np{:}\mathbb{N}, ord{:}Array_i(\mathbb{Z}), key\_d{:}Array_1(\mathbb{Z}), rank{:}Array_1(\mathbb{N}), b_0{:}\mathbb{N}, b_1{:}\mathbb{N})] \\
\quad init \mathrel{\widehat=} b_0, b_1 := i \times (\lfloor Bmax/N \rfloor + 1), (i+1) \times (\lfloor Bmax/N \rfloor) + i;\ \fatsemi j : 0..Bmax \bullet ord[j] := 0 \\
\quad rank \mathrel{\widehat=} \textbf{var}\ bucket, aux : \mathbb{N} \bullet \\
\qquad \fatsemi j : 0..i-1 \bullet \mu P \bullet c.j.i?aux \rightarrow \textbf{if}\ aux = -1 \rightarrow SKIP \\
\qquad\qquad\qquad\qquad\qquad\qquad [] aux \neq -1 \rightarrow ord[aux] := ord[aux] + 1;\ P \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi}; \\
\qquad \fatsemi j : 0..Np-1 \bullet\ bucket := \lfloor k[j]/(\lfloor Bmax/N \rfloor + 1) \rfloor; \textbf{if}\ bucket \neq i \rightarrow c.i.bucket!k[j] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [] bucket = i \rightarrow ord[k[j]] := ord[k[j]] + 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi}; \\
\qquad \fatsemi j : 0..i-1 \bullet c.i.j! - 1 \rightarrow SKIP; \\
\qquad \fatsemi j : i+1..N-1 \bullet c.i.j! - 1 \rightarrow SKIP; \\
\qquad \fatsemi j : i+1..N-1 \bullet \mu P \bullet c.j.i?aux \rightarrow \textbf{if}\ aux = -1 \rightarrow SKIP \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [] aux \neq -1 \rightarrow ord[aux] := ord[aux] + 1;\ P \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi}; \\
\qquad \textbf{if}\ i = 0 \rightarrow rank[b_0] := 1;\ \fatsemi m : b_0 + 1..b_1 \bullet rank[m] := rank[m-1] + ord[m-1]; \\
\qquad\qquad\qquad c.i.(i+1)!(rank[b_1] + ord[b_1]) \rightarrow SKIP \\
\qquad [] (i > 0) \wedge (i < N-1) \rightarrow c.(i-1).i?aux \rightarrow rank[b_0] := aux; \\
\qquad\qquad\qquad\qquad\qquad\qquad \fatsemi m : b_0 + 1..b_1 \bullet rank[m] := rank[m-1] + ord[m-1]; \\
\qquad\qquad\qquad\qquad\qquad\qquad c.i.(i+1)!(rank[b_1] + ord[b_1]) \rightarrow SKIP \\
\qquad [] i = N-1 \rightarrow c.(i-1).i?aux \rightarrow rank[b_0] := aux; \\
\qquad\qquad\qquad\qquad \fatsemi m : b_0 + 1..b_1 \bullet rank[m] := rank[m-1] + ord[m-1] \\
\qquad \textbf{fi} \\
\quad \bullet\ init;\ rank \\
\textbf{end}
\end{array}
$$

Figura 5.2: Especificação Concreta Circus/HCL de IntegerSort

Usando novamente a lei L.3 na instrução de especificação

$$rank : \left[\begin{array}{c} \forall j : b_0..m-1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux, \\ \forall j : b_0..m \bullet rank[j]' = (\sum_{l=b_0}^{j-1} ord[l]) + aux \end{array} \right]$$

, fica-se com $rank[m] := rank[m-1] + ord[m-1]$, deixando a obrigação de prova OP.3. Com isso, tem-se que

$$rank[b_0] := aux;\ \fatsemi m : b_0 + 1..b_1 \bullet rank[m] := rank[m-1] + ord[m-1].$$

O refinamento da instrução do quarto item é análoga ao do terceiro item, tanto na aplicação das leis quanto nas obrigações de prova, bastando trocar a variável *aux* pela constante 1, obtendo-se $rank[b_0] := 1;\ \fatsemi m : b_0+1..b_1 \bullet rank[m] := rank[m-1]+ord[m-1]$.

Após terem sido aplicados os passos de refinamento, chega-se ao componente IntegerSort refinado na Figura 5.2.

```
void init()
{
    b_0 = getRank() * (Math.floor(Bmax/getSize()) + 1);
    b_1 = (getRank()+1) * (Math.floor(Bmax/getSize())) + getRank();
}

void rank()
{
    ulong bucket,aux;
    for (int j=0;j <= i-1;j++) {
        Func F = null;
        F = () => (aux = c.Receive<ulong>(j,0);
                   if (aux == -1) {return;}
                   else if (aux != -1) {ord[aux]++;}
                   else {Environment.Exit(1);})
        F;
    }
    for (int j=0;j <= Np-1;j++) {
        bucket[j] = Math.floor(k[j]/(Math.floor(Bmax/getSize()) + 1));
        if (bucket != getRank()) {c.Send(k[j],bucket,0);}
        else if (bucket == getRank()) {ord[k[j]]++;}
        else {Environment.Exit(1);}
    }
    for (j=0; j<=getRank()-1; j++) c.Send(-1,j,0);
    for (j=getRank()+1;j<=getSize()-1;j++) c.Send(-1,j,0);
    for (int j=getRank()+1;j <= getSize()-1;j++) {
        Func F = null;
        F = () => (aux = c.Receive<ulong>(j,0);
                   if (aux == -1) {return;}
                   else if (aux != -1) {ord[aux]++;}
                   else {Environment.Exit(1);})
        F;
    }
    if (getRank()==0) {
        rank[b_0] = 1;
        for(m=b_0+1; m<=b_1; m++)
            rank[m] = rank[m-1] + ord[m-1];
        c.Send(rank[b_1]+ord[b_1],
                                    getRank()+1, 0);
    }
    else if ((getRank() > 0) && (getRank() < getSize() -1)) {
        aux = c.Receive<ulong>(getRank()-1,0);
        rank[b_0] = aux;
        for(m=b_0+1; m<=b_1; m++)
            rank[m] = rank[m-1] + ord[m-1];
        c.Send(rank[b_1]+ord[b_1],
                                    getRank()+1, 0);
    }
    else if (getRank() == getSize()-1) {
        aux = c.Receive<ulong>(getRank()-1,0);
        rank[b_0] = aux;
        for(m=b_0+1; m<=b_1; m++)
            rank[m] = rank[m-1] + ord[m-1];
    }
    else{
        Environment.Exit(1);
    }
}

public void compute()
{
    init();
    rank();
}
```

Figura 5.3: Tradução de IntegerSort

$$
\begin{array}{l}
\mid Bmax : \mathbb{N} \\
Bmax > 0 \\
\textbf{channel } c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\textbf{process } sorter \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\quad \textbf{state } State \mathrel{\widehat=} [k{:}Array_1(\mathbb{Z}), Np{:}\mathbb{N}, ord{:}Array_1(\mathbb{Z}), rank{:}Array_1(\mathbb{N}), b_0{:}\mathbb{N}, b_1{:}\mathbb{N}] \\
\quad init \mathrel{\widehat=} [State' \mid \forall j : 0..Bmax \bullet ord[j]' = 0 \land \\
\qquad\qquad b_0' = i \times (\lfloor \frac{Bmax}{N} \rfloor + 1) \land b_1' = (i+1) \times (\lfloor \frac{Bmax}{N} \rfloor) + i] \\
\quad \textbf{action } rank \mathrel{\widehat=} \textbf{var } bucket, aux : \mathbb{N} \bullet \\
\qquad \mathop{;}_{9} j : 0..i-1 \bullet \mu P \bullet c.j.i?aux \rightarrow \textbf{if } aux = -1 \rightarrow SKIP \\
\qquad\qquad\qquad\qquad [] aux \neq -1 \rightarrow ord[aux] := ord[aux] + 1;\ P \\
\qquad\qquad\qquad\qquad \textbf{fi}; \\
\qquad \mathop{;}_{9} j : 0..Np-1 \bullet bucket{:}[bucket' = \lfloor \frac{k[j]}{\lfloor \frac{Bmax}{N} \rfloor + 1} \rfloor]; \\
\qquad\qquad\qquad \textbf{if } bucket \neq i \rightarrow c.i.bucket!k[j] \\
\qquad\qquad\qquad [] bucket = i \rightarrow ord[k[j]] := ord[k[j]] + 1 \\
\qquad\qquad\qquad \textbf{fi}; \\
\qquad \mathop{;}_{9} j : 0..i-1 \bullet c.i.j! - 1 \rightarrow SKIP; \\
\qquad \mathop{;}_{9} j : i+1..N-1 \bullet c.i.j! - 1 \rightarrow SKIP; \\
\qquad \mathop{;}_{9} j : i+1..N-1 \bullet \mu P \bullet c.j.i?aux \rightarrow \textbf{if } aux = -1 \rightarrow SKIP \\
\qquad\qquad\qquad\qquad [] aux \neq -1 \rightarrow ord[aux] := ord[aux] + 1;\ P \\
\qquad\qquad\qquad\qquad \textbf{fi}; \\
\qquad \textbf{if } i = 0 \rightarrow rank{:}[\forall j : b_0..b_1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + 1]; \\
\qquad\qquad c.i.(i+1)!(rank[b_1] + ord[b_1]) \rightarrow SKIP \\
\qquad [] (i > 0) \land (i < N-1) \rightarrow c.(i-1).i?aux \rightarrow rank{:}[\forall j : b_0..b_1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux]; \\
\qquad\qquad c.i.(i+1)!(rank[b_1] + ord[b_1]) \rightarrow SKIP \\
\qquad [] i = N-1 \rightarrow c.(i-1).i?aux \rightarrow rank{:}[\forall j : b_0..b_1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux] \\
\qquad \textbf{fi} \\
\quad \bullet\ init;\ rank \\
\textbf{end} \\
\textbf{process } IntegerSort \mathrel{\widehat=} \Big|\Big|_{c}\ i : \{0..N-1\} \bullet sorter(i)
\end{array}
$$

Figura 5.4: Contrato Circus do Componente IntegerSort

5.2.3 Tradução

A tradução do componente IntegerSort, utilizado-se das regras de tradução descritas no capítulo 4, é apresentada na Figura 5.3. Por se tratar de um componente da espécie computação, o método que dá inicio ao componente, executando a sua ação principal (protocolo), é o método compute.

5.2.4 Verificação

Nas figuras 5.4 e 5.5, são apresentadas a tradução para a linguagem Circus do contrato e da especificação concreta do componente IntegerSort, em Circus/HCL, respectivamente.

$$
\begin{array}{l}
\mid Bmax : \mathbb{N} \\
Bmax > 0 \\
\textbf{channel}\ c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\\
\textbf{process}\ sorter \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat=} [k{:}Array_1(\mathbb{Z}), Np{:}\mathbb{N}, ord{:}Array_i(\mathbb{Z}), key\_d{:}Array_1(\mathbb{Z}), rank{:}Array_1(\mathbb{N}), b_0{:}\mathbb{N}, b_1{:}\mathbb{N})] \\
\\
\quad init \mathrel{\widehat=} b_0, b_1 := i \times (\lfloor Bmax/N \rfloor + 1), (i+1) \times (\lfloor Bmax/N \rfloor) + i; \\
\qquad \mathop{⨟}_{9} j : 0..Bmax \bullet ord[j] := 0 \\
\\
\quad rank \mathrel{\widehat=} \textbf{var}\ bucket, aux : \mathbb{N} \bullet \\
\\
\qquad \mathop{⨟}_{9} j : 0..i-1 \bullet \mu P \bullet c.j.i?aux \rightarrow \textbf{if}\ aux = -1 \rightarrow SKIP \\
\qquad\qquad\qquad\qquad\qquad\qquad [] aux \neq -1 \rightarrow ord[aux] := ord[aux] + 1;\ P \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi}; \\
\qquad \mathop{⨟}_{9} j : 0..Np-1 \bullet bucket := \lfloor k[j]/(\lfloor Bmax/N \rfloor + 1) \rfloor; \\
\qquad\qquad\qquad\qquad \textbf{if}\ bucket \neq i \rightarrow c.i.bucket!k[j] \\
\qquad\qquad\qquad\qquad [] bucket = i \rightarrow ord[k[j]] := ord[k[j]] + 1 \\
\qquad\qquad\qquad\qquad \textbf{fi}; \\
\qquad \mathop{⨟}_{9} j : 0..i-1 \bullet c.i.j!{-1} \rightarrow SKIP; \\
\qquad \mathop{⨟}_{9} j : i+1..N-1 \bullet c.i.j!{-1} \rightarrow SKIP; \\
\qquad \mathop{⨟}_{9} j : i+1..N-1 \bullet \mu P \bullet c.j.i?aux \rightarrow \textbf{if}\ aux = -1 \rightarrow SKIP \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [] aux \neq -1 \rightarrow ord[aux] := ord[aux] + 1;\ P \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi}; \\
\qquad \textbf{if}\ i = 0 \rightarrow rank[b_0] := 1;\ \mathop{⨟}_{9} m : b_0 + 1..b_1 \bullet rank[m] := rank[m-1] + ord[m-1]; \\
\qquad\qquad\qquad c.i.(i+1)!(rank[b_1] + ord[b_1]) \rightarrow SKIP \\
\qquad [] (i > 0) \wedge (i < N-1) \rightarrow c.(i-1).i?aux \rightarrow rank[b_0] := aux; \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathop{⨟}_{9} m : b_0 + 1..b_1 \bullet rank[m] := rank[m-1] + ord[m-1]; \\
\qquad\qquad\qquad\qquad\qquad\qquad c.i.(i+1)!(rank[b_1] + ord[b_1]) \rightarrow SKIP \\
\qquad [] i = N-1 \rightarrow c.(i-1).i?aux \rightarrow rank[b_0] := aux; \\
\qquad\qquad\qquad\qquad \mathop{⨟}_{9} m : b_0 + 1..b_1 \bullet rank[m] := rank[m-1] + ord[m-1] \\
\qquad \textbf{fi} \\
\\
\quad \bullet\ init;\ rank \\
\\
\textbf{end} \\
\\
\textbf{process}\ IntegerSort \mathrel{\widehat=} \Big|\Big|_{c}\ i : \{0..N-1\} \bullet sorter(i)
\end{array}
$$

Figura 5.5: Especificação Concreta Circus do Componente IntegerSort

synchronizer AllGather$\langle N \rangle$ **where**
channel $c : \mathbb{N} \times \mathbb{N} \times \mathbb{R}$
process $gather \mathrel{\widehat=} \|_{c}\ i : \{0..N-1\} \bullet$
begin

state $State \mathrel{\widehat=} [v : Array_1(\mathbb{R}), beg : \mathbb{N}, end : \mathbb{N}, dim : \mathbb{N}]$
$allGather \mathrel{\widehat=}$

$\fatsemi\ j : 0..beg-1 \bullet c.j.i?v[j];$
$\fatsemi\ j : beg..end \bullet$
$(\fatsemi\ k : 0..i-1 \bullet c.i.k!v[j];\ \fatsemi\ k : i+1..N-1 \bullet c.i.k!v[j]);$
$\fatsemi\ j : end+1..dim-1 \bullet c.j.i?v[j];$

$\bullet\ allGather$

end

Figura 5.6: Contrato do Componente AllGather

5.3 Gradiente Conjugado (CG)

O kernel CG (*Conjugate Gradient*) consiste em estimar o menor autovalor de uma matriz esparsa, definida positiva e simétrica usando o método da potência inversa. Ele utiliza o método do *gradiente conjugado* para resolver o sistema de equações lineares necessários para achar o menor autovalor da matriz dada. O tamanho da matriz, número de interações e outras constantes são definidas de acordo com a carga de trabalho utilizada.

O método do *gradiente conjugado* é um método numérico iterativo usado na solução de sistemas de equações lineares onde a matriz que representa tal sistema é uma matriz definida positiva e simétrica.

A seguir, o processo de derivação é aplicado ao contrato ConjugateGradient. O seu contrato é descrito, assim como os contratos de seus componente aninhados AllGather, MatVecProduct através da linguagem Circus/HCL, como feito anteriormente com o componente IntegerSort. O componente ReduceSum também é usado como componente aninhado. Porém, como o seu contrato, refinação e tradução já foram apresentados no capítulo 4, nesta seção será apresentada apenas a transformação do seu contrato e especificação concreta para a linguagem Circus.

5.3.1 Contrato

Na Figura 5.8, é apresentado o contrato do componente ConjugateGradient, o qual possui componentes aninhados dos tipos AllGather, MatVecProduct e ReduceSum. Os contratos dos dois primeiros estão apresentados nas figuras 5.6 e 5.7, respectivamente, enquanto o do último já foi discutido como exemplo no Capítulo 4.

O componente AllGather representa uma operação de comunicação coletiva bem difundida em programas paralelos, responsável por fazer a distribuição de elementos consecutivos de um vetor entre todos os processos participantes, onde cada processo

computation MatVecProduct$\langle N \rangle$ **where**
process *product* $\widehat{=} \Big|\Big|\Big| i : \{0..N-1\} \bullet$
begin

state $State \widehat{=} [lines : \mathbb{N}, dim : \mathbb{N}, A : Array_2(\mathbb{R}), v : Array_1(\mathbb{R}), x : Array_1(\mathbb{R})]$
$computeProduct \widehat{=} x : [\forall j : 0..lines - 1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k]]$
$\bullet$ *computeProduct*

end

Figura 5.7: Contrato do Componente MatVecProduct

recebe apenas uma parte de igual número de elementos do vetor. O componente MatVecProduct faz a multiplicação entre uma matriz de dimensões *lines* $\times$ *dim* e um vetor de tamanho *dim*.

Por fim, o componente ConjugateGradient é responsável pelo método do gradiente conjugado de que trata o kernel CG. O seu estado consiste, além das variáveis contidas nos seus componentes aninhados, nas seguintes variáveis:

- $\boldsymbol{A}$, $\boldsymbol{z}$ e $\boldsymbol{x}$, representando os operandos do sistema de equações $A \times z = x$, a ser resolvido, onde A e x devem ser inicializadas antes da execução do componente;
- ***dim***, representando o valor da dimensão da matriz, também conhecido antes da execução do componente;
- ***lines***, representando o número de linhas da matriz A e dos vetores x e z armazenados em cada unidade;
- $\boldsymbol{r}$, representando o valor do resíduo ($r = x - A \times z$);
- ***residue***, representando a norma do resíduo ($residue = \sqrt{r^T \times r}$).

A contrato do componente ConjugateGradient descreve o método do gradiente conjugado onde cada unidade contem um número de linhas, representado pelo valor *lines*, da matriz A e do vetor x. O contrato consiste em dois parágrafos. O parágrafo *init* é responsável pela descrição da inicialização do estado do componente, é atribuido zero a todos o elementos do vetor z, o vetor de resíduo r é inicializado com o valor do vetor x, e as variáveis que precisam de inicialização dos componentes MatVecProduct e AllGather também são inicializadas.

O parágrafo de ação *cg* é responsável pela descrição do método do gradiente conjugado propriamente dito, computando o vetor z e o resíduo. Todas as operações referentes à matriz A ou ao vetor x são feitas de modo paralelo, pois cada unidade contem apenas algumas linhas da matriz A e do vetor x.

computation ConjugateGradient$\langle N \rangle$ **where**
inner component rs: ReduceSum $\langle N \rangle$
inner component a: AllGather $\langle N \rangle$
inner component mv: MatVecProduct $\langle N \rangle$

process $cg \mathrel{\widehat{=}} \Big|\Big|\Big| \; i : \{0..N-1\} \bullet$
begin

slice $rs.reduce[i]$
slice $a.gather[i]$
slice $mv.product[i]$

state $State \mathrel{\widehat{=}} [rs::, mv::, a::, residue{:}\mathbb{R}, dim{:}\mathbb{N}, lines{:}\mathbb{N},$
$z{:}Array_1(\mathbb{R}), x{:}Array_1(\mathbb{R}), A{:}Array_2(\mathbb{R}), r{:}Array_1(\mathbb{R})]$
$init \mathrel{\widehat{=}} lines : \left[\; (i = N-1 \land lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \lor (i \neq N-1 \land lines' = \lfloor \frac{dim}{N} \rfloor) \; \right];$
$z : [\forall j : 0..lines-1 \bullet z[j]' = 0]; \; r : [\forall j : 0..lines-1 \bullet r[j]' = x[j]];$
$mv::A : [\forall j : 0..lines-1 \bullet \forall k : 0..dim-1 \bullet mv::A[j,k]' = A[j,k]];$
$mv::dim : [mv::dim' = dim];$
$a::dim : [a::dim' = dim]$

$cg \mathrel{\widehat{=}}$ **var** $p : Array_1(\mathbb{R}); \; q : Array_1(\mathbb{R}); \; rho : \mathbb{R}; \; rhoi : \mathbb{R}; \; alfa : \mathbb{R}; \; beta : \mathbb{R} \bullet$

$rho : [rho' = \sum_{j=0}^{lines-1} r[j]^2]; \;\; rs::k := rho; \;\; rs!; \;\; rho := rs::k;$
$p : [\forall j : 0..lines-1 \bullet p[j]' = r[j]];$
$\overset{\circ}{9} \; j : 1..25 \bullet$
$a:: \left[\begin{array}{c} (i \neq N-1 \land a::beg' = lines \times i \land a::end' = lines \times (i+1) - 1 \land \\ \forall k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \lor \\ (i = N-1 \land a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \land a::end' = dim - 1 \land \\ \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim-1) \bullet a::v[k]' = p[k - lines \times i]) \end{array} \right];$
$mv::v : [\forall k : 0..dim-1 \bullet mv::v[k]' = a::v[k]]; \;\; mv.computePrhoduct!;$
$q : [\forall k : 0..lines-1 \bullet q[k]' = mv::x[k]];$
$alfa : [alfa' = \frac{\sum_{k=0}^{lines-1} p[k] \times q[k]}{rho}]; \;\; rs::k := alfa; \;\; rs!;$
$alfa := \frac{1}{rs::k}; \; z : [\forall k : 0..lines-1 \bullet z[k]' = z[k] + alfa \times p[k]];$
$rhoi := rho; \;\; r : [\forall k : 0..lines-1 \bullet r[k]' = r[k] - alfa \times q[k]];$
$rho : [rho' = \sum_{k=0}^{lines-1} r[k]^2]; \;\; rs::k := rho; \; rs!; \;\; rho := rs::k;$
$beta : [beta' = \frac{rho}{rhoi}]; \;\; p : [\forall k : 0..lines-1 \bullet p[k]' = r[k] + beta \times p[k]];$
$residue : [residue' = \sum_{k=0}^{lines-1} r[k]^2]; \;\; rs::k := residue; \;\; rs!;$
$residue : [residue' = \sqrt{rs::k}]$

$\bullet \; init; \; cg$

end

Figura 5.8: Contrato do Componente ConjugateGradient

5.3.2 Refinamento

A seguir, é apresentado o processo de refinamento dos componentes MatVecProduct e ConjugateGradient. O contrato do componente AllGather já é uma especificação concreta, e, por esse motivo, não é preciso refiná-la. Eventualmente, aplicações de leis de refinamento deixam obrigações de provas que são demonstradas no apêndice A. Algumas obrigações de prova são triviais, normalmente provenientes da aplicação da lei de refinamento L.3 em instruções de especificação do tipo $w : [v' = e]$. A demonstração de tais obrigações de prova triviais serão omitidas do apêndice A.

Refinamento de MatVecProduct

O refinamento do contrato do componente MatVecProduct restringe-se à instrução de especificação correspondente a ação *computeProduct*, como descrito a seguir.

Inicialmente, temos a instrução de especificação

$$x : [\forall j : 0..lines - 1 \bullet x[j]' = \textstyle\sum_{k=0}^{dim-1} A[j,k] \times v[k]].$$

Após a aplicação *lines* vezes da lei de refinamento L.2, obtém-se

$$\fatsemi\, m : 0..lines - 1 \bullet x : \left[\begin{array}{c} \forall j : 0..m - 1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k], \\ \forall j : 0..m \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \end{array} \right].$$

Daqui em diante, basta refinar a instrução de especificação interna à operação de iteração de sequência, obtendo-se

$$x : \left[\begin{array}{c} \forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k], \\ \forall j : 0..m \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \end{array} \right].$$

Então, aplicando-se $dim + 1$ vezes a lei L.2, obtém-se

$$x : \left[\begin{array}{c} \forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k], \\ \forall j : 0..m-1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m]' = 0 \end{array} \right] ; \fatsemi\, l : 0..dim-1 \bullet$$
$$x : \left[\begin{array}{c} \forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m] = \sum_{k=0}^{l-1} A[m,k] \times v[k], \\ \forall j : 0..m-1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m]' = \sum_{k=0}^{l} A[m,k] \times v[k] \end{array} \right].$$

É preciso ainda refinar as duas instruções de especificação resultantes da aplicação da

computation MATVECPRODUCT$\langle N \rangle$ **where**
process $product \mathrel{\widehat=} \Big|\Big|\Big| i : \{0..N-1\} \bullet$
begin

state $State \mathrel{\widehat=} [lines : \mathbb{N}, dim : \mathbb{N}, A : Array_2(\mathbb{R}), v : Array_1(\mathbb{R}), x : Array_1(\mathbb{R})]$
$computeProduct \mathrel{\widehat=} \fatsemi\, m : 0..lines-1 \bullet (x[m] := 0;\ \fatsemi\, l : 0..dim-1 \bullet x[m] := x[m] + A[m,l] \times v[l])$
$\bullet\ computeProduct$

end

Figura 5.9: Especificação Concreta de MATVECPRODUCT

última lei de refinamento. Inicialmente, tem-se a instrução de especificação

$$x : \left[\begin{array}{c} \forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k], \\ \forall j : 0..m-1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m]' = 0 \end{array} \right];$$

. Aplicando a lei L.3, e assim deixando a obrigação de prova OP.4, tem-se $x[m] := 0$.

Por fim, tem-se a instrução de especificação

$$x : \left[\begin{array}{l} \forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m] = \sum_{k=0}^{l-1} A[m,k] \times v[k], \\ \forall j : 0..m-1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m]' = \sum_{k=0}^{l} A[m,k] \times v[k] \end{array} \right].$$

Aplicando-se a lei L.3, e assim deixando a obrigação de prova OP.5, obtém-se

$$x[m] := x[m] + A[m,l] \times v[l].$$

Finalmente, é obtida a especificação concreta do componente MATVECPRODUCT, apresentada na Figura 5.9.

Refinamento de CONJUGATEGRADIENT

A seguir, refinaremos o contrato do componente CONJUGATEGRADIENT, que é bastante extenso, consistindo fundamentalmente de duas ações: *init* e *cg*, ambas contendo várias instruções de especificação. Primeiramente, será refinada a ação *init*, que consiste de 6 instruções de especificação.

A primeira instrução de especificação da ação *init* é

$$l : \left[\begin{array}{c} (i = N-1 \wedge lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \vee \\ (i \neq N-1 \wedge lines' = \lfloor \frac{dim}{N} \rfloor) \end{array} \right].$$

Aplicando-se a lei de refinamento L.5, deixando a obrigação de prova OP.6, obtém-se

$$\textbf{if } i = N-1 \rightarrow l: \left[\begin{array}{c} i = N-1, \\ (i = N-1 \wedge lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \vee \\ (i \neq N-1 \wedge lines' = \lfloor \frac{dim}{N} \rfloor) \end{array}\right]$$
$$[]\, i \neq N-1 \rightarrow l: \left[\begin{array}{c} i \neq N-1, \\ (i = N-1 \wedge lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \vee \\ (i \neq N-1 \wedge lines' = \lfloor \frac{dim}{N} \rfloor) \end{array}\right]$$
fi.

Finalmente, aplicando duas vezes a lei de refinamento L.3, deixando as obrigações de prova OP.7 e OP.8, obtém-se

$$\textbf{if } i = N-1 \rightarrow lines := dim - (N-1) \times \lfloor dim/N \rfloor$$
$$[]\, i \neq N-1 \rightarrow lines := \lfloor dim/N \rfloor$$
fi.

A segunda instrução de especificação da ação *init* é

$$z : [\forall j : 0..lines - 1 \bullet z[j]' = 0].$$

Aplicando *lines* vezes a lei de refinamento L.2, obtém-se

$$\S\, m : 0..lines - 1 \bullet z : [\forall j : 0..m - 1 \bullet z[j] = 0, \forall j : 0..m \bullet z[j]' = 0].$$

Finalmente, aplicando a lei de refinamento L.3, deixando a obrigação de prova OP.9, obtém-se $\S\, m : 0..lines - 1 \bullet z[m] := 0$.

A terceira instrução de especificação da ação *init* é $r : [\forall j : 0..lines - 1 \bullet r[j]' = x[j]]$. Aplicando-se *lines* vezes a lei de refinamento L.2, obtém-se

$$\S\, m : 0..lines - 1 \bullet r : [\forall j : 0..m - 1 \bullet r[j] = x[j], \forall j : 0..m \bullet r[j]' = x[j]].$$

Finalmente, aplicando-se a lei de refinamento L.3, deixando a obrigação de prova OP.10, obtém-se $\S\, m : 0..lines - 1 \bullet r[m] := x[m]$.

A quarta instrução de especificação da ação *init* é

$$mv::A : [\forall j : 0..lines - 1 \bullet \forall k : 0..dim - 1 \bullet mv::A[j, k]' = A[j, k]].$$

Aplicando-se *lines* vezes a lei de refinamento L.2, obtém-se

$$\fatsemi\, m : 0..lines-1 \bullet mv::A : \left[\begin{array}{c} \forall\, j : 0..m-1 \bullet \forall\, k : 0..dim-1 \bullet mv::A[j,k] = A[j,k], \\ \forall\, j : 0..m \bullet \forall\, k : 0..dim-1 \bullet mv::A[j,k]' = A[j,k] \end{array} \right].$$

Então, aplicando-se *dim* vezes a lei de refinamento L.2, obtém-se

$$\fatsemi\, m : 0..lines-1 \bullet \fatsemi\, l : 0..dim-1 \bullet$$
$$mv::A : \left[\begin{array}{c} (\forall\, j : 0..m-1 \bullet \forall\, k : 0..dim-1 \bullet mv::A[j,k] = A[j,k]) \wedge \\ (\forall\, j : 0..m \bullet \forall\, k : 0..l-1 \bullet mv::A[j,k] = A[j,k]), \\ (\forall\, j : 0..m-1 \bullet \forall\, k : 0..dim-1 \bullet mv::A[j,k]' = A[j,k]) \wedge \\ \forall\, j : 0..m \bullet \forall\, k : 0..l \bullet mv::A[j,k]' = A[j,k] \end{array} \right].$$

Finalmente, aplicando-se a lei de refinamento L.3, deixando a obrigação de prova OP.11, tem-se $\fatsemi\, m : 0..lines-1 \bullet \fatsemi\, l : 0..dim-1 \bullet mv::A[m,l] := A[m,l]$.

A quinta instrução de especificação da ação *init* é $mv :: dim : [mv :: dim' = dim]$. Aplicando-se a lei de refinamento L.3, deixando uma obrigação de prova trivial, obtém-se $mv::dim := dim$.

A sexta e última instrução de especificação da ação *init* é $a :: dim : [a :: dim' = dim]$. Aplicando-se a lei de refinamento L.3, deixando uma obrigação de prova trivial, obtém-se $a::dim := dim$.

Finalmente, após o refinamento de seis das suas instruções de especificação, obtém-se a especificação concreta da ação *init*:

$$\begin{array}{l} init \mathrel{\widehat{=}} \mathbf{if}\, i = N-1 \rightarrow lines := dim - (N-1) \times \lfloor dim/N \rfloor \\ \quad [\!]\, i \neq N-1 \rightarrow lines := \lfloor dim/N \rfloor \\ \quad \mathbf{fi}; \\ \quad \fatsemi\, m : 0..lines-1 \bullet z[m] := 0;\ \fatsemi\, m : 0..lines-1 \bullet r[m] := x[m]; \\ \quad \fatsemi\, m : 0..lines-1 \bullet \fatsemi\, l : 0..dim-1 \bullet mv::A[m,l] := A[m,l]; \\ \quad mv::dim := dim;\ a::dim := dim \end{array}$$

A seguir, será apresentado o processo de refinamento da ação *cg*, que consiste em 13 instruções de especificação.

A primeira instrução de especificação da ação *cg* é $rho : [rho' = \sum_{j=0}^{lines-1} r[j]^2]$. Aplicando-se $lines + 1$ vezes a lei de refinamento L.2, obtém-se

$$rho : [rho' = 0];\ \fatsemi\, m : 0..lines-1 \bullet rho : [rho = \textstyle\sum_{j=0}^{m-1} r[j]^2, rho' = \sum_{j=0}^{m} r[j]^2].$$

Finalmente, aplicando-se duas vezes lei de refinamento L.3, deixando uma obrigação de prova trivial e a obrigação de prova OP.12, obtém-se

$$rho := 0;\ {}^{\circ}_{9}\ m : 0..lines - 1 \bullet rho := rho + r[m]^2.$$

A segunda instrução de especificação da ação *cg* é $p : [\forall j : 0..lines - 1 \bullet p[j]' = r[j]]$. Aplicando-se $lines + 1$ vezes a lei de refinamento L.2, obtém-se

$${}^{\circ}_{9}\ m : 0..lines - 1 \bullet p : [\forall j : 0..m - 1 \bullet p[j] = r[j], \forall j : 0..m \bullet p[j]' = r[j]].$$

Finalmente, aplicando-se a lei de refinamento L.3, deixando a obrigação de prova OP.13, obtém-se ${}^{\circ}_{9}\ m : 0..lines - 1 \bullet p[m] := r[m]$.

A terceira instrução de especificação da ação *cg* é

$$a:: \left[\begin{array}{c} (i \neq N-1 \wedge a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge \\ \forall k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \vee \\ (i = N-1 \wedge a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\ \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i]) \end{array} \right].$$

Aplicando-se a lei de refinamento L.5, deixando a obrigação de prova OP.14, obtém-se

$$\begin{array}{l} \textbf{if } i = N-1 \rightarrow a:: \left[\begin{array}{c} i = N-1, \\ (i \neq N-1 \wedge a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge \\ \forall k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \vee \\ (i = N-1 \wedge a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\ \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i]) \end{array} \right] \\ [\!] i \neq N-1 \rightarrow a:: \left[\begin{array}{c} i \neq N-1, \\ (i \neq N-1 \wedge a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge \\ \forall k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \vee \\ (i = N-1 \wedge a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\ \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i]) \end{array} \right] \\ \textbf{fi}. \end{array}$$

Então, aplicando-se duas vezes a lei de refinamento L.6, deixando as obrigações de prova OP.15 e OP.16, obtém-se

$$\begin{array}{l} \textbf{if } i = N-1 \rightarrow a:: \left[\begin{array}{c} i = N-1, \\ (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\ \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i]) \end{array} \right] \\ [\!] i \neq N-1 \rightarrow a:: \left[\begin{array}{c} i \neq N-1, \\ (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge \\ \forall k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \end{array} \right] \\ \textbf{fi}. \end{array}$$

Em seguida, aplicando-se $N - (i \times \lfloor \frac{dim}{N} \rfloor) + 2$ vezes a lei de refinamento L.2, obtém-se

$$\begin{array}{l}
\textbf{if } i = N-1 \rightarrow a::\left[\begin{array}{c} i = N-1, \\ a::end' = dim - 1 \end{array}\right]; \; a::\left[\begin{array}{c} a::end = dim - 1, \\ (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1) \end{array}\right]; \\
\qquad \mathop{\S}_{9} m : (i \times \lfloor dim/N \rfloor)..(dim-1) \bullet \\
\qquad\qquad a::\left[\begin{array}{c} (a::beg = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end = dim - 1 \wedge \\ \forall\, k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m-1) \bullet a::v[k] = p[k - lines \times i]), \\ a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\ \forall\, k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m) \bullet a::v[k]' = p[k - lines \times i] \end{array}\right] \\
[\!]\, i \neq N-1 \rightarrow a::\left[\begin{array}{c} i \neq N-1, \\ (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge \\ \forall\, k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \end{array}\right] \\
\textbf{fi}.
\end{array}$$

Logo depois, aplicando-se três vezes a lei de refinamento L.3, deixando uma obrigação de prova trivial e as obrigação de prova OP.17 e OP.18, obtém-se

$$\begin{array}{l}
\textbf{if } i = N-1 \rightarrow a::end := dim-1;\; a::beg := i \times \lfloor \frac{dim}{N} \rfloor; \\
\qquad \mathop{\S}_{9} m : (i \times \lfloor dim/N \rfloor)..(dim-1) \bullet a::v[m] := p[m - lines \times i] \\
[\!]\; i \neq N-1 \rightarrow a::\left[\begin{array}{c} i \neq N-1, \\ (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge \\ \forall\, k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \end{array}\right] \\
\textbf{fi}.
\end{array}$$

Posteriormente, aplicando-se $lines + 2$ vezes a lei de refinamento L.2, obtém-se

$$\begin{array}{l}
\textbf{if } i = N-1 \rightarrow a::end := dim - 1;\; a::beg := i \times \lfloor \frac{dim}{N} \rfloor; \\
\qquad \mathop{\S}_{9} m : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim-1) \bullet a::v[m] := p[m - lines \times i] \\
[\!]\, i \neq N-1 \rightarrow a::\left[\begin{array}{c} i \neq N-1, \\ (a::end' = lines \times (i+1) - 1) \end{array}\right]; \\
\qquad\qquad a::\left[\begin{array}{c} a::end = lines \times (i+1) - 1, \\ (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1) \end{array}\right]; \\
\qquad \mathop{\S}_{9} m : (lines \times i)..(lines \times (i+1) - 1) \bullet \\
\qquad\qquad\qquad a::\left[\begin{array}{c} (a::beg = lines \times i \wedge a::end = lines \times (i+1) - 1 \wedge \\ \forall\, k : (lines \times i)..(m-1) \bullet a::v[k] = p[k - lines \times i]), \\ (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge \\ \forall\, k : (lines \times i)..m \bullet a::v[k]' = p[k - lines \times i]) \end{array}\right] \\
\textbf{fi}.
\end{array}$$

Finalmente, aplicando-se três vezes a lei de refinamento L.3, deixando uma obrigação de

prova trivial e as obrigação de prova OP.19 e OP.20, obtém-se

$$
\begin{array}{l}
\textbf{if}\, i = N-1 \rightarrow a::end := dim - 1;\ a::beg := i \times \lfloor \frac{dim}{N} \rfloor; \\
\qquad\qquad \fatsemi\, m : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim-1) \bullet a::v[m] := p[m - lines \times i] \\
[]\, i \neq N-1 \rightarrow a::end := lines \times (i+1) - 1;\ a::beg := lines \times i; \\
\qquad\qquad \fatsemi\, m : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[m] := p[m - lines \times i] \\
\textbf{fi}.
\end{array}
$$

A quarta instrução de especificação da ação *cg* é

$$mv::v : [\forall\, k : 0..dim - 1 \bullet mv::v[k]' = a::v[k]].$$

Aplicando-se *dim* vezes a lei de refinamento L.2, obtém-se

$$\fatsemi\, m : 0..dim-1 \bullet mv::v : \left[\begin{array}{c} \forall\, k : 0..m-1 \bullet mv::v[k] = a::v[k], \\ \forall\, k : 0..m \bullet mv::v[k]' = a::v[k] \end{array} \right].$$

Finalmente, aplicando-se a lei de refinamento L.3, deixando a obrigação de prova OP.21, obtém-se

$$\fatsemi\, m : 0..dim - 1 \bullet mv::v[m] := a::v[m].$$

A quinta instrução de especificação da ação *cg* é

$$q : [\forall\, k : 0..lines - 1 \bullet q[k]' = mv::x[k]].$$

Aplicando-se *lines* vezes a lei de refinamento L.2, obtém-se

$$\fatsemi\, m : 0..lines - 1 \bullet q : \left[\begin{array}{c} \forall\, k : 0..m - 1 \bullet q[k] = mv::x[k], \\ \forall\, k : 0..m \bullet q[k]' = mv::x[k] \end{array} \right].$$

Enfim, aplicando-se a lei de refinamento L.3, deixando a obrigação de prova OP.22, obtém-se $\fatsemi\, m : 0..lines - 1 \bullet q[m] := mv::x[m]$.

A sexta instrução de especificação da ação *cg* é $alfa : [alfa' = \frac{\sum_{k=0}^{lines-1} p[k] \times q[k]}{rho}]$. Aplicando-se $lines + 2$ vezes a lei de refinamento L.2, obtém-se

$$
\begin{array}{l}
alfa : [alfa' = 0]; \\
\fatsemi\, m : 0..lines - 1 \bullet alfa : [alfa = \sum_{k=0}^{m-1} p[k] \times q[k], alfa' = \sum_{k=0}^{m} p[k] \times q[k]]; \\
alfa : [alfa = \sum_{k=0}^{m} p[k] \times q[k], alfa' = \frac{\sum_{k=0}^{m} p[k] \times q[k]}{rho}].
\end{array}
$$

Finalmente, aplicando-se três vezes a lei de refinamento L.3, deixando uma obrigação de prova trivial e as obrigação de prova OP.23 e OP.24, obtém-se

$$alfa := 0;\ (\fatsemi\, m : 0..lines - 1 \bullet alfa := alfa + p[m] \times q[m]);\ alfa := alfa/rho.$$

A sétima instrução de especificação da ação *cg* é

$$z : [\forall\, k : 0..lines - 1 \bullet z[k]' = z[k] + alfa \times p[k]].$$

Aplicando-se *lines* vezes a lei de refinamento L.2, obtém-se

$$\fatsemi\, m : 0..lines - 1 \bullet z : \left[\begin{array}{c} \forall\, k : 0..m - 1 \bullet z[k] = z[k] + alfa \times p[k], \\ \forall\, k : 0..m \bullet z[k]' = z[k] + alfa \times p[k] \end{array}\right].$$

Finalmente, aplicando a lei de refinamento L.3, deixando a obrigação de prova OP.25, obtém-se $\fatsemi\, m : 0..lines - 1 \bullet z[m] := z[m] + alfa \times p[m]$.

A oitava instrução de especificação da ação *cg* é

$$r : [\forall\, k : 0..lines - 1 \bullet r[k]' = r[k] - alfa \times q[k]].$$

Aplicando-se *lines* vezes a lei de refinamento L.2, obtém-se

$$\fatsemi\, m : 0..lines - 1 \bullet r : \left[\begin{array}{c} \forall\, k : 0..m - 1 \bullet r[k] = r[k] - alfa \times q[k], \\ \forall\, k : 0..m \bullet r[k]' = r[k] - alfa \times q[k] \end{array}\right].$$

Finalmente, aplicando a lei de refinamento L.3, deixando a obrigação de prova OP.26, obtém-se $\fatsemi\, m : 0..lines - 1 \bullet r[m] := r[m] - alfa \times q[m]$.

A nona instrução de especificação da ação *cg* é $rho : [rho' = \sum_{k=0}^{lines-1} r[k]^2]$. Aplicando-se $lines + 1$ vezes a lei de refinamento L.2, obtém-se

$$rho : [rho' = 0];\ \fatsemi\, m : 0..lines - 1 \bullet rho : [rho = \textstyle\sum_{k=0}^{m-1} r[k]^2, rho' = \sum_{k=0}^{m} r[k]^2].$$

Finalmente, aplicando-se duas vezes a lei de refinamento L.3, deixando uma obrigação de prova trivial e a obrigação de prova OP.26, obtém-se

$$rho := 0;\ \fatsemi\, m : 0..lines - 1 \bullet rho := rho + r[m]^2.$$

A décima instrução de especificação da ação *cg* é $beta : [beta' = \frac{rho}{rhoi}]$. Aplicando-se a lei de refinamento L.3, deixando uma obrigação de prova trivial, obtém-se $beta := rho/rhoi$.

computation ConjugateGradient$\langle N \rangle$ **where**

inner component rs: ReduceSum $\langle N \rangle$
inner component a: AllGather $\langle N \rangle$
inner component mv: MatVecProduct $\langle N \rangle$

process $cg \mathrel{\widehat{=}} \Big|\Big|\Big| \; i : \{0..N-1\} \bullet$
begin

slice $rs.reduce[i]$
slice $a.gather[i]$
slice $mv.product[i]$

state $State \mathrel{\widehat{=}} [rs::, mv::, a::, residue : \mathbb{R}, dim : \mathbb{N},$
$lines : \mathbb{N}, z : Array_1(\mathbb{R}), x : Array_1(\mathbb{R}), A : Array_2(\mathbb{R}), r : Array_1(\mathbb{R})]$

$init \mathrel{\widehat{=}}$ **if** $i = N-1 \rightarrow lines := dim - (N-1) \times \lfloor dim/N \rfloor$
$[\!] \; i \neq N-1 \rightarrow lines := \lfloor dim/N \rfloor$
fi;
$\mathop{⨟}_{9} m : 0..lines-1 \bullet z[m] := 0$; $\mathop{⨟}_{9} m : 0..lines-1 \bullet r[m] := x[m]$;
$\mathop{⨟}_{9} m : 0..lines-1 \bullet \mathop{⨟}_{9} l : 0..dim-1 \bullet mv::A[m,l] := A[m,l]$;
$mv::dim := dim$; $a::dim := dim$

$cg \mathrel{\widehat{=}}$ **var** $p : Array_1(\mathbb{R})$; $q : Array_1(\mathbb{R})$; $rho : \mathbb{R}$; $rhoi : \mathbb{R}$; $alfa : \mathbb{R}$; $beta : \mathbb{R} \bullet$

$rho := 0$; $\mathop{⨟}_{9} m : 0..lines-1 \bullet rho := rho + r[m]^2$;
$rs::k := rho$; $rs!$; $rho := rs::k$;
$\mathop{⨟}_{9} m : 0..lines-1 \bullet p[m] := r[m]$;
$\mathop{⨟}_{9} j : 1..25 \bullet$
if $i = N-1 \rightarrow a::end := dim - 1$; $a::beg := i \times \lfloor \frac{dim}{N} \rfloor$;
$\mathop{⨟}_{9} m : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim-1) \bullet a::v[m] := p[m - lines \times i]$
$[\!] \; i \neq N-1 \rightarrow a::end := lines \times (i+1) - 1$; $a::beg := lines \times i$;
$\mathop{⨟}_{9} m : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[m] := p[m - lines \times i]$
fi;
$\mathop{⨟}_{9} m : 0..dim-1 \bullet mv::v[m] := a::v[m]$;
$mv!$; $\mathop{⨟}_{9} m : 0..lines-1 \bullet q[m] := mv::x[m]$;
$alfa := 0$; $(\mathop{⨟}_{9} m : 0..lines-1 \bullet alfa := alfa + p[m] \times q[m])$;
$alfa := alfa/rho$; $rs::k := alfa$; $rs!$;
$alfa := \frac{1}{rs::k}$;
$\mathop{⨟}_{9} m : 0..lines-1 \bullet z[m] := z[m] + alfa \times p[m]$;
$rhoi := rho$; $\mathop{⨟}_{9} m : 0..lines-1 \bullet r[m] := r[m] - alfa \times q[m]$;
$rho := 0$; $\mathop{⨟}_{9} m : 0..lines-1 \bullet rho := rho + r[m]^2$;
$rs::k := rho$; $rs!$; $rho := rs::k$;
$beta := rho/rhoi$;
$\mathop{⨟}_{9} m : 0..lines-1 \bullet p[m] := r[m] + beta \times p[m]$;
$residue := 0$; $\mathop{⨟}_{9} m : 0..lines-1 \bullet residue := residue + r[m]^2$; $rs::k := residue$; $rs!$;
$residue := \sqrt{rs::k}$

$\bullet \; init$; cg

end

Figura 5.10: Especificação Concreta de ConjugateGradient

A décima-primeira instrução de especificação da ação *cg* é

$$p : [\forall\, k : 0..lines-1 \bullet p[k]' = r[k] + beta \times p[k]].$$

Aplicando-se $lines + 1$ vezes a lei de refinamento L.2, obtém-se

$$⨟\, m : 0..lines - 1 \bullet p : \left[\begin{array}{c} \forall\, k : 0..m-1 \bullet p[k] = r[k] + beta \times p[k], \\ \forall\, k : 0..m \bullet p[k]' = r[k] + beta \times p[k] \end{array} \right].$$

Finalmente, aplicando-se a lei de refinamento L.3, deixando a obrigação de prova OP.28, obtém-se $⨟\, m : 0..lines - 1 \bullet p[m] := r[m] + beta \times p[m]$.

A décima-segunda instrução de especificação da ação *cg* é

$$residue : [residue' = \textstyle\sum_{k=0}^{lines-1} r[k]^2].$$

Aplicando-se $lines + 1$ vezes a lei de refinamento L.2, obtém-se

$$residue : [residue' = 0];$$
$$⨟\, m : 0..lines - 1 \bullet residue : [residue = \textstyle\sum_{k=0}^{m-1} r[k]^2, residue' = \sum_{k=0}^{m} r[k]^2].$$

Finalmente, aplicando-se duas vezes a lei de refinamento L.3, deixando uma obrigação de prova trivial e a obrigação de prova OP.29, obtém-se

$$residue := 0;\ ⨟\, m : 0..lines - 1 \bullet residue := residue + r[m]^2.$$

A décima-terceira e última instrução da ação *cg* é $residue : [residue' = \sqrt{r::k}]$. Aplicando-se a lei de refinamento L.3, deixando uma obrigação de prova trivial, obtém-se $residue := \sqrt{r::k}$.

Finalmente, juntando as especificações concretas das ações *init* e *cg*, tem-se a especificação concreta completa do componente ConjugateGradient na Figura 5.10.

5.3.3 Tradução

A tradução dos componentes AllGather, MatVecProduct e ConjugateGradient, utilizado-se das regras de tradução descritas no Capítulo 4, são apresentadas nas figuras 5.11, 5.12 and 5.13. A tradução do componente ReduceSum será omitida, pois já foi apresentada no capítulo 4.

```
void allGather()
{
    for(int j = 0;j<=beg-1;j++)
        v[j] = c.Receive<double>(j,0);

    for(int j = beg;j<=end;j++)
    {
        for(int k = 0;k<=getSize()-1;k++)
            c.Send(v[j],k,0);

        for(int k = getSize+1;k<=N-1;k++)
            c.Send(v[j],k,0);
    }
    for(int j = end+1;j<=dim-1;j++)
        v[j] = c.Receive<double>(j,0);
}

public void synchronize()
{
    allGather();
}
```

Figura 5.11: Tradução de AllGather

```
void computeProduct()
{
    for(int m = 0;m<=lines-1;m++)
    {
        x[m] = 0;
        for(int l = 0;l<=dim-1;l++)
        {
            x[m] = x[m] + A[m,l] * v[l];
        }
    }
}

public void compute()
{
    computeProduct();
}
```

Figura 5.12: Tradução de MatVecProduct

```
void init()
{
    if (getRank() == getSize()-1){
       lines = dim - (getSize()-1) * System.Math.Floor(((double)dim)/getSize());
    }
    else if (getRank() != getSize()-1){
       lines = Math.floor(((double)dim)/getSize());
    }
    else{ Environment.Exit(1); }
    for(int m = 0; m <= lines-1; m++) z[m] = 0;
    for(int m = 0; m <= lines-1; m++) r[m] = x[m];
    for(int m = 0; m <= lines-1; m++)
        for(int l = 0; m <= dim-1; m++) mv.A[m,l] = A[m,l];
    mv.Dim = dim;
    a.Dim = dim;
}

void cg()
{
    double[] p, q;
    double rho = 0, rhoi, alfa, beta;
    for(int m = 0; m <= lines-1; m++) rho += System.Math.Pow(r[m],2);
    rs.K = rho;
    rs.synchronize();
    rho = rs.K;
    for(int m = 0; m <= lines-1; m++) p[m] = r[m];
    for(int j = 1; j <= 25; j++)
    {
        if (getRank() == getSize()-1){
            a.End = dim-1;
            a.Beg = getRank() * System.Math.Floor(dim/n);
            for(m=i * System.Math.Floor(dim/n);m<=dim-1;m++)
                a.V[m] = p[m-lines*getRank()];
        }
        else if (getRank() != getSize()-1){
            a.End = lines * (getRank()+1) - 1;
            a.Beg = lines * getRank();
            for(m=lines * getRank();m<=lines * (getRank()+1) - 1;m++)
                a.V[m] = p[m-lines*getRank()];
            }
        else{ Environment.Exit(1); }
        for(m = 0; m <= dim-1; m++) mv.V[m] = a.V[m];
        mv.compute();
        for(m = 0; m <= lines-1; m++) q[m] = mv.X[m];
        alfa = 0;
        for(m = 0; m <= lines-1; m++) alfa = alfa + p[m]*q[m];
        alfa = ((double)alfa) / rho;
        rs.V = alfa;
        rs.synchronize();
        alfa = ((double)1) / rs.V;
        for(m = 0; m <= dim-1; m++) z[m] = z[m] + alfa * p[m];
        rhoi = rho;
        for(m = 0; m <= dim-1; m++) r[m] = r[m] - alfa * q[m];
        rho = 0;
        for(int m = 0; m <= lines-1; m++) rho += System.Math.Pow(r[m],2);
        rs.V = rho;
        rs.synchronize();
        rho = rs.V;
        beta = ((double)rho)/rhoi;
        for(int m = 0; m <= lines-1; m++) p[m] = r[m] + beta * p[m];
        residue = 0;
        for(int m = 0; m <= lines-1; m++) residue += System.Math.Pow(r[m],2);
        rs.V = residue;
        rs.synchronize();
        residue =  System.Math.Pow(rs.V,((double)1)/2);
    }
}

public void compute()
{
    init();
    cg();
}
```

Figura 5.13: Tradução de ConjugateGradient

$$
\begin{array}{l}
| \; N : \mathbb{N} \\
N > 0 \\
\textbf{channel } c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\\
\textbf{process } reduce \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\quad \textbf{state } State \mathrel{\widehat=} [k : \mathbb{R}] \\
\quad updState \mathrel{\widehat=} \textbf{val } aux : \mathbb{R} \bullet [\Delta State, aux? : \mathbb{R} \mid k' = k + aux?] \\
\quad reduceSum \mathrel{\widehat=} \textbf{var } aux : \mathbb{R} \bullet \\
\qquad (i = 0 \;\&\; |||\, j : 1..N-1 \bullet c.j.0?aux \rightarrow updState(aux); \\
\qquad |||\, j : 1..N-1 \bullet c.0.j!k) \\
\qquad \Box \\
\qquad (i \neq 0 \;\&\; c.i.0!k \rightarrow c.0.i?aux \rightarrow updState(aux)) \\
\quad \bullet \; reduceSum \\
\\
\textbf{end} \\
\\
\textbf{process } ReduceSum \mathrel{\widehat=} \Big|\Big|_{c} \; i : \{0..N-1\} \bullet reduce(i)
\end{array}
$$

Figura 5.14: Contrato Circus do Componente ReduceSum

5.3.4 Verificação

A tradução dos contratos e especificações concretas dos componentes ReduceSum, AllGather, MatVecProduct e ConjugateGradient para a linguagem Circus são apresentados nas figuras 5.14, 5.15, 5.16, 5.17, 5.18, 5.19, 5.20 e 5.21, respectivamente.

$$
\begin{array}{l}
| \; N : \mathbb{N} \\
N > 0 \\
\textbf{channel } c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\textbf{process } gather \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\\
\quad \textbf{state } State \mathrel{\widehat=} [v : Array_1(\mathbb{R}), beg : \mathbb{N}, end : \mathbb{N}, dim : \mathbb{N}] \\
\\
\quad allGather \mathrel{\widehat=} \mathop{\fatsemi}_{j : 0..beg-1} c.j.i?v[j]; \\
\qquad\qquad \mathop{\fatsemi}_{j : beg..end} \bullet \\
\qquad\qquad \mathop{\fatsemi}_{k : 0..i-1} c.i.k!v[j]; \; \mathop{\fatsemi}_{k : i+1..N-1} c.i.k!v[j]; \\
\qquad\qquad \mathop{\fatsemi}_{j : end+1..dim-1} c.j.i?v[j]; \\
\\
\quad \bullet \; allGather \\
\\
\textbf{end} \\
\\
\textbf{process } AllGather \mathrel{\widehat=} \Big|\Big|_{c} \; i : \{0..N-1\} \bullet gather(i)
\end{array}
$$

Figura 5.15: Contrato Circus do Componente AllGather

5.4 Considerações Finais

Nesta seção, foi apresentada a aplicação de todo o processo de derivação aos contratos IntegerSort e ConjugateGradient. Tais contratos foram desenvolvidos baseados nas

$$
\begin{array}{l}
\mid N : \mathbb{N} \\
N > 0 \\
\textbf{process}\ product \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat=} [lines : \mathbb{N}, dim : \mathbb{N}, A : Array_2(\mathbb{R}), v : Array_1(\mathbb{R}), x : Array_1(\mathbb{R})] \\
\quad computeProduct \mathrel{\widehat=} x : [\forall j : 0..lines - 1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k]] \\
\quad \bullet\ computeProduct \\
\textbf{end} \\
\textbf{process}\ MatVecProduct \mathrel{\widehat=} \Big|\Big|_{c}\ i : \{0..N-1\} \bullet product(i)
\end{array}
$$

Figura 5.16: Contrato Circus do Componente MatVecProduct

descrições presentes em [19]. Todas as etapas do processo de derivação dos componentes IntegerSort e ConjugateGradient foram feitas de forma manual.

Aplicar o processo na derivação e certificação de componentes, de forma manual, ou seja, sem o auxílio de ferramentas, é um trabalho árduo e bastante susceptível a erros. Na etapa de refinamento, a aplicação correta das leis, e as demonstrações das obrigações de prova são tarefas que exigem bastante atenção, e, pelo fato de ser bastante extensa, os erros que acontecem nessa etapa são mais difíceis de se detectar. As etapas de tradução para a linguagem alvo e tradução para a linguagem Circus são menos críticas, pelo fato de se tratar apenas de transformações sintáticas e, pelo fato de que possíveis erros são facilmente detectados, sobretudo na tradução para a linguagem alvo. Por esses motivos, o uso de ferramentas é importante, pois elas abreviam e facilitam o processo de derivação dos componentes como um todo. Na etapa de refinamento, podem ser usados refinadores para que as leis sejam aplicadas corretamente e, quando possível, as obrigações de prova são automaticamente demonstradas ou refutadas. Nas etapas de tradução, tradutores podem ser desenvolvidos para fazer as transformações necessárias para a linguagem alvo ou para o Circus. Mesmo que tais ferramentas não sejam totalmente automáticas, requerendo a intervenção humana algumas vezes, elas são imprescindíveis para o desenvolvimento de componentes livres de erros e que obedeçam aos seus contratos.

Após a tradução para a linguagem Circus, para se fazer verificações a respeito do contrato ou das especificações concretas derivadas, é necessário que tanto o contrato quanto as especificações concretas sejam expressos de acordo com as suas semânticas, que são baseadas na TUP e definidas através da lógica de primeira ordem. A semântica de um processo é definida através da semântica de seus parágrafos de ações, que, por sua vez, são definidos através da semântica de cada ação presente no parágrafo. A semântica de cada ação pode ser encontrada em [96, 129]. Tendo-se a fórmula que representa a semântica do processo, qualquer verificação de propriedades pode ser feita. Seja P a propriedade que se deseja demonstrar e S a fórmula que representa a semântica do processo, para saber se a propriedade P vale, basta provar que $S \Rightarrow P$; para o caso específico onde se

$$
\begin{array}{l}
| \; N : \mathbb{N} \\
N > 0 \\
\textbf{channel}\ rs\_c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\textbf{channel}\ a\_c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\\
\textbf{process}\ cg \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\\
\quad \textbf{state}\ State \mathrel{\widehat=} [residue : \mathbb{R}, dim : \mathbb{N}, lines : \mathbb{N}, z : Array_1(\mathbb{R}), x : Array_1(\mathbb{R}), A : Array_2(\mathbb{R}), r : Array_1(\mathbb{R}), \\
\qquad\qquad mv::lines : \mathbb{N}, mv::dim : \mathbb{N}, mv::A : Array_2(\mathbb{R}), mv::v : Array_1(\mathbb{R}), \\
\qquad\qquad mv::x : Array_1(\mathbb{R}), a::v : Array_1(\mathbb{R}), a::beg : \mathbb{N}, a::end : \mathbb{N}, a::dim : \mathbb{N}, rs::k : \mathbb{R}] \\
\quad mv\_computeProduct \mathrel{\widehat=} x : [\forall\, j : 0..mv::lines - 1 \bullet mv::x[j]' = \textstyle\sum_{k=0}^{mv::dim-1} mv::A[j,k] \times mv::v[k]] \\
\quad mv \mathrel{\widehat=} mv\_computeProduct \\
\\
\quad a\_allGather \mathrel{\widehat=} \mathop{\fatsemi}_{9} j : 0..a::beg - 1 \bullet a\_c.j.i?a::v[j]; \\
\qquad\qquad \mathop{\fatsemi}_{9} j : a::beg..a::end \bullet \\
\qquad\qquad \mathop{\fatsemi}_{9} k : 0..i - 1 \bullet a\_c.i.k!a::v[j]; \ \mathop{\fatsemi}_{9} k : i + 1..N - 1 \bullet a\_c.i.k!a::v[j]; \\
\qquad\qquad \mathop{\fatsemi}_{9} j : a::end + 1..a::dim - 1 \bullet a\_c.j.i?a::v[j]; \\
\\
\quad a\_MainAction \mathrel{\widehat=} a\_allGather \\
\quad rs\_updState \mathrel{\widehat=} \textbf{val}\ aux : \mathbb{R} \bullet [\Delta State, aux? : \mathbb{R} \mid rs::k' = rs::k + aux?] \\
\quad rs\_reduceSum \mathrel{\widehat=} \textbf{var}\ aux : \mathbb{R} \bullet \\
\\
\qquad (i = 0 \ \&\ |||\, j : 1..N - 1 \bullet rs\_c.j.0?aux \rightarrow rs\_updState(aux); \\
\qquad |||\, j : 1..N - 1 \bullet rs\_c.0.j!rs::k) \\
\qquad \Box \\
\qquad (i \neq 0 \ \&\ rs\_c.i.0!rs::k \rightarrow rs\_c.0.i?aux \rightarrow rs\_updState(aux)) \\
\\
\quad rs\_MainAction \mathrel{\widehat=} rs\_reduceSum \\
\\
\quad init \mathrel{\widehat=} l : \left[\; (i = N - 1 \wedge lines' = dim - (N-1) \times \lfloor \tfrac{dim}{N} \rfloor) \vee (i \neq N - 1 \wedge lines' = \lfloor \tfrac{dim}{N} \rfloor) \; \right]; \\
\qquad z : [\forall\, j : 0..lines - 1 \bullet z[j]' = 0]; \ r : [\forall\, j : 0..lines - 1 \bullet r[j]' = x[j]]; \\
\qquad mv::A : [\forall\, j : 0..lines - 1 \bullet \forall\, k : 0..dim - 1 \bullet mv::A[j,k]' = A[j,k]]; \\
\qquad mv::dim : [mv::dim' = dim]; \ /; \ a::dim : [a::dim' = dim] \\
\\
\quad cg \mathrel{\widehat=} \textbf{var}\ p : Array_1(\mathbb{R}); \ q : Array_1(\mathbb{R}); \ rho : \mathbb{R}; \ rhoi : \mathbb{R}; \ alfa : \mathbb{R}; \ beta : \mathbb{R} \bullet \\
\\
\qquad rho : [rho' = \textstyle\sum_{j=0}^{lines-1} r[j]^2]; \ rs::k := rho; \ rs\_MainAction; \ rho := rs::k; \\
\qquad p : [\forall\, j : 0..lines - 1 \bullet p[j]' = r[j]]; \\
\qquad \mathop{\fatsemi}_{9} j : 1..25 \bullet \\
\qquad\qquad a:: \left[\begin{array}{c} (i \neq N - 1 \wedge a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge \\ \forall\, k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \vee \\ (i = N - 1 \wedge a::beg' = i \times \lfloor \tfrac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\ \forall\, k : (i \times \lfloor \tfrac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i]) \end{array} \right]; \\
\qquad\qquad mv::v : [\forall\, k : 0..dim - 1 \bullet mv::v[k]' = a::v[k]]; \ mv\_MainAction; \\
\qquad\qquad q : [\forall\, k : 0..lines - 1 \bullet q[k]' = mv::x[k]]; \\
\qquad\qquad alfa : [alfa' = \tfrac{\sum_{k=0}^{lines-1} p[k] \times q[k]}{rho}]; \ \ rs::k := alfa; \ rs\_MainAction; \\
\qquad\qquad alfa := \tfrac{1}{rsk}; \ z : [\forall\, k : 0..lines - 1 \bullet z[k]' = z[k] + alfa \times p[k]]; \\
\qquad\qquad rhoi := rho; \ \ r : [\forall\, k : 0..lines - 1 \bullet r[k]' = r[k] - alfa \times q[k]]; \\
\qquad\qquad rho : [rho' = \textstyle\sum_{k=0}^{lines-1} r[k]^2]; \ \ rs::k := rho; \ /; \ rs\_MainAction; \ rho := rs::k; \\
\qquad\qquad beta : [beta' = \tfrac{rho}{rhoi}]; \ \ p : [\forall\, k : 0..lines - 1 \bullet p[k]' = r[k] + beta \times p[k]]; \\
\qquad\qquad residue : [residue' = \textstyle\sum_{k=0}^{lines-1} r[k]^2]; \ \ rs::k := residue; \ rs\_MainAction; \\
\qquad\qquad residue : [residue' = \sqrt{rs::k}] \\
\\
\quad \bullet\ init; \ cg \\
\\
\textbf{end} \\
\\
\textbf{process}\ ConjugateGradient \mathrel{\widehat=} \Big|\Big|_{c} \; i : \{0..N - 1\} \bullet cg(i)
\end{array}
$$

Figura 5.17: Contrato Circus do Componente ConjugateGradient

$$
\begin{array}{l}
| \; N : \mathbb{N} \\
N > 0 \\
\textbf{channel } c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\\
\textbf{process } reduce \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\quad \textbf{state } State \mathrel{\widehat=} [k : \mathbb{R}] \\
\quad updState \mathrel{\widehat=} \textbf{val } aux : \mathbb{R} \bullet k := k + aux \\
\\
\quad reduceSum \mathrel{\widehat=} \textbf{var } aux : \mathbb{R} \bullet (i = 0 \;\&\; |||\, j : 1..N-1 \bullet c.j.0?aux \rightarrow updState(aux); \\
\qquad\qquad\qquad\qquad |||\, j : 1..N-1 \bullet c.0.j!k) \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad (i \neq 0 \;\&\; c.i.0!k \rightarrow c.0.i?aux \rightarrow updState(aux)) \\
\\
\quad \bullet \; reduceSum \\
\\
\textbf{end} \\
\\
\textbf{process } ReduceSum \mathrel{\widehat=} \Big|\Big|_{c} \; i : \{0..N-1\} \bullet reduce(i)
\end{array}
$$

Figura 5.18: Especificação Concreta Circus do Componente ReduceSum

deseja mostrar que uma especificação concreta de semântica S satisfaz um contrato de semântica C, basta provar que: $S \Rightarrow C$. Para auxiliar nesta tarefa, pode ser utilizado um provador de teoremas, que consiste em uma ferramentas usada para demonstrar ou refutar proposições lógicas. As fórmulas derivadas dos contratos e especificações concretas seriam devidamente codificadas em um provador e a ferramenta se encarregaria de fazer o trabalho de demonstrar ou refutar as implicações. O provador de teoremas ProofPower-Z pode ser utilizado na demonstração dos teoremas, onde, fazendo-se uso da semântica já definida da notação Z, poderia-se codificar uma especificação Circus através de parágrafos Z, assim como mostrado em [131], para se fazer a demonstração de propriedades.

$$
\begin{array}{l}
| \; N : \mathbb{N} \\
N > 0 \\
\textbf{channel } c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\textbf{process } gather \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\quad \textbf{state } State \mathrel{\widehat=} [v : Array_1(\mathbb{R}), beg : \mathbb{N}, end : \mathbb{N}, dim : \mathbb{N}] \\
\\
\quad allGather \mathrel{\widehat=} \mathop{\fatsemi}_{9} j : 0..beg-1 \bullet c.j.i?v[j]; \\
\qquad\qquad\qquad \mathop{\fatsemi}_{9} j : beg..end \bullet \\
\qquad\qquad\qquad \mathop{\fatsemi}_{9} k : 0..i-1 \bullet c.i.k!v[j]; \; \mathop{\fatsemi}_{9} k : i+1..N-1 \bullet c.i.k!v[j]; \\
\qquad\qquad\qquad \mathop{\fatsemi}_{9} j : end+1..dim-1 \bullet c.j.i?v[j]; \\
\\
\quad \bullet \; allGather \\
\\
\textbf{end} \\
\\
\textbf{process } AllGather \mathrel{\widehat=} \Big|\Big|_{c} \; i : \{0..N-1\} \bullet gather(i)
\end{array}
$$

Figura 5.19: Especificação Concreta Circus do Componente AllGather

$$
\begin{array}{l}
| \; N : \mathbb{N} \\
N > 0 \\
\textbf{process}\ product \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat=} [lines : \mathbb{N}, dim : \mathbb{N}, A : Array_2(\mathbb{R}), v : Array_1(\mathbb{R}), x : Array_1(\mathbb{R})] \\
\quad computeProduct \mathrel{\widehat=} \mathop{\fatsemi}_{9} m : 0..lines{-}1 \bullet (x[m] := 0; \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathop{\fatsemi}_{9} l : 0..dim - 1 \bullet x[m] := x[m] + A[m, l] \times v[l]) \\
\quad \bullet\ computeProduct \\
\textbf{end} \\
\textbf{process}\ MatVecProduct \mathrel{\widehat=} \Big\|_{c}\ i : \{0..N - 1\} \bullet product(i)
\end{array}
$$

Figura 5.20: Especificação Concreta Circus do Componente MatVecProduct

$$
\begin{array}{l}
| \; N : \mathbb{N} \\
N > 0 \\
\textbf{channel}\ rs\_c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\textbf{channel}\ a\_c : \mathbb{N} \times \mathbb{N} \times \mathbb{R} \\
\\
\textbf{process}\ cg \mathrel{\widehat=} i : \mathbb{N} \bullet \\
\textbf{begin} \\
\\
\quad \textbf{state}\ State \mathrel{\widehat=} [residue : \mathbb{R}, dim : \mathbb{N}, lines : \mathbb{N}, z : Array_1(\mathbb{R}), x : Array_1(\mathbb{R}), A : Array_2(\mathbb{R}), r : Array_1(\mathbb{R}), \\
\qquad\qquad mv::lines : \mathbb{N}, mv::dim : \mathbb{N}, mv::A : Array_2(\mathbb{R}), mv::v : Array_1(\mathbb{R}), \\
\qquad\qquad mv::x : Array_1(\mathbb{R}), a::v : Array_1(\mathbb{R}), a::beg : \mathbb{N}, a::end : \mathbb{N}, a::dim : \mathbb{N}, rs::k : \mathbb{R}] \\
\\
\quad mv\_computeProduct \mathrel{\widehat=} x : [\forall\, j : 0..mv::lines - 1 \bullet mv::x[j]' = \textstyle\sum_{k=0}^{mv::dim-1} mv::A[j,k] \times mv::v[k]] \\
\\
\quad mv \mathrel{\widehat=} mv\_computeProduct \\
\\
\quad a\_allGather \mathrel{\widehat=} \mathop{⨾}_{9} j : 0..a::beg - 1 \bullet a\_c.j.i?a::v[j]; \\
\qquad\qquad\qquad \mathop{⨾}_{9} j : a::beg..a::end \bullet \\
\qquad\qquad\qquad \mathop{⨾}_{9} k : 0..i - 1 \bullet a\_c.i.k!a::v[j]; \mathop{⨾}_{9} k : i + 1..N - 1 \bullet a\_c.i.k!a::v[j]; \\
\qquad\qquad\qquad \mathop{⨾}_{9} j : a::end + 1..a::dim - 1 \bullet a\_c.j.i?a::v[j]; \\
\\
\quad a\_MainAction \mathrel{\widehat=} a\_allGather \\
\\
\quad rs\_updState \mathrel{\widehat=} \textbf{val}\ aux : \mathbb{R} \bullet [\Delta State, aux? : \mathbb{R} \mid rs::k' = rs::k + aux?] \\
\\
\quad rs\_reduceSum \mathrel{\widehat=} \textbf{var}\ aux : \mathbb{R} \bullet (i = 0\ \&\ |||\, j : 1..N - 1 \bullet rs\_c.j.0?aux \rightarrow rs\_updState(aux); \\
\qquad\qquad\qquad\qquad\qquad\qquad |||\, j : 1..N - 1 \bullet rs\_c.0.j!rs::k) \\
\qquad\qquad\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad\qquad\qquad (i \neq 0\ \&\ rs\_c.i.0!rs::k \rightarrow rs\_c.0.i?aux \rightarrow rs\_updState(aux)) \\
\\
\quad rs\_MainAction \mathrel{\widehat=} rs\_reduceSum \\
\\
\quad init \mathrel{\widehat=} \textbf{if}\ i = N - 1 \rightarrow lines := dim - (N - 1) \times \lfloor dim/N \rfloor \\
\qquad\quad [\!]\, i \neq N - 1 \rightarrow lines := \lfloor dim/N \rfloor \\
\qquad\quad \textbf{fi}; \\
\\
\quad \mathop{⨾}_{9} m : 0..lines - 1 \bullet z[m] := 0; \mathop{⨾}_{9} m : 0..lines - 1 \bullet r[m] := x[m]; \\
\quad \mathop{⨾}_{9} m : 0..lines - 1 \bullet \mathop{⨾}_{9} l : 0..dim - 1 \bullet mv::A[m,l] := A[m,l]; \\
\quad mv::dim := dim;\ a::dim := dim \\
\\
\quad cg \mathrel{\widehat=} \textbf{var}\ p : Array_1(\mathbb{R});\ q : Array_1(\mathbb{R});\ rho : \mathbb{R};\ rhoi : \mathbb{R};\ alfa : \mathbb{R};\ beta : \mathbb{R} \bullet \\
\\
\qquad rho := 0; \mathop{⨾}_{9} m : 0..lines - 1 \bullet rho := rho + r[m]^2;\ rs::k := rho; \\
\qquad rs::k := rho;\ \ rs\_MainAction!;\ \ rho := rs::k; \\
\qquad \mathop{⨾}_{9} m : 0..lines - 1 \bullet p[m] := r[m]; \\
\qquad \mathop{⨾}_{9} j : 1..25 \bullet \\
\qquad\qquad \textbf{if}\ i = N - 1 \rightarrow a::end := dim - 1;\ a::beg := i \times \lfloor \tfrac{dim}{N} \rfloor; \\
\qquad\qquad\qquad\qquad \mathop{⨾}_{9} m : (i \times \lfloor \tfrac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[m] := p[m - lines \times i] \\
\qquad\qquad [\!]\, i \neq N - 1 \rightarrow a::end := lines \times (i + 1) - 1;\ a::beg := lines \times i; \\
\qquad\qquad\qquad\qquad \mathop{⨾}_{9} m : (lines \times i)..(lines \times (i + 1) - 1) \bullet a::v[m] := p[m - lines \times i] \\
\qquad\qquad \textbf{fi}; \\
\qquad\qquad \mathop{⨾}_{9} m : 0..dim - 1 \bullet mv::v[m] := a::v[m]; \\
\qquad\qquad mv!; \mathop{⨾}_{9} m : 0..lines - 1 \bullet q[m] := mv::x[m]; \\
\qquad\qquad alfa := 0; (\mathop{⨾}_{9} m : 0..lines - 1 \bullet alfa := alfa + p[m] \times q[m]); \\
\qquad\qquad alfa := alfa/rho;\ rs::k := alfa;\ rs\_MainAction; \\
\qquad\qquad alfa := \tfrac{1}{rs::k}; \\
\qquad\qquad \mathop{⨾}_{9} m : 0..lines - 1 \bullet z[m] := z[m] + alfa \times p[m]; \\
\qquad\qquad rhoi := rho; \mathop{⨾}_{9} m : 0..lines - 1 \bullet r[m] := r[m] - alfa \times q[m]; \\
\qquad\qquad rho := 0; \mathop{⨾}_{9} m : 0..lines - 1 \bullet rho := rho + r[m]^2; \\
\qquad\qquad rs::k := rho;\ /;\ rs\_MainAction;\ /;\ rho := rs::k; \\
\qquad\qquad beta := rho/rhoi; \\
\qquad\qquad \mathop{⨾}_{9} m : 0..lines - 1 \bullet p[m] := r[m] + beta \times p[m]; \\
\qquad\qquad residue := 0; \mathop{⨾}_{9} m : 0..lines - 1 \bullet residue := residue + rs[m]^2;\ r::k := residue;\ rs\_MainAction; \\
\qquad\qquad residue := \sqrt{rs::k} \\
\\
\quad \bullet\ init;\ cg \\
\\
\textbf{end} \\
\\
\textbf{process}\ ConjugateGradient \mathrel{\widehat=} \Big|\Big|_{c}\ i : \{0..N - 1\} \bullet cg(i)
\end{array}
$$

Figura 5.21: Especificação Concreta Circus do Componente ConjugateGradient

Capítulo 6

Conclusões e Propostas de Trabalhos Futuros

A certificação de componentes na nuvem computacional de serviços de computação de alto desempenho (CAD), a qual constitui o pano de fundo a partir do qual são motivados os objetivos desta dissertação, tem o papel importante de garantir, antes da execução de uma aplicação, de importância crítica e/ou de longa duração, se é seguro usar um certo componente no contexto de utilização no qual ele está sendo ligado a aplicação. Para isso, utiliza-se o conceito de contrato, o qual define propriedades funcionais de um componente cuja implementação mais otimizada pode variar entre diferentes arquiteturas de plataformas de computação paralela, aproveitando-se de características intrínsecas de uma dada arquitetura. Esse é um requisito importante no domínio de aplicações de CAD, raramente presente no domínio de aplicações corporativas e de negócios, onde a tecnologia de componentes foi desenvolvida e tem sido utilizada com relativo êxito.

Esta dissertação investigou o uso de certos formalismos e técnicas para lidar com as questões envolvidas na busca por soluções a esse problema, tomando como base na experiência anterior dos projetos do grupo de pesquisa envolvido, notadamente relacionados ao Haskell$_{\#}$ [33] e ao HPE (*Hash Programming Environment*) [32].

Foi proposto um processo para derivação formal de componentes cujo ponto de partida é o contrato para o qual se deseja construir um componente concreto que o implemente. Isso é conseguido através de etapas de refinamento do contrato até ser construída uma especificação concreta do componente, a qual é traduzida para a linguagem alvo. Os passos de refinamento e tradução devem preservar as propriedades do contrato. Para a certificação do componente derivado, utiliza-se a especificação concreta do componente, que foi gerada na etapa de refinamento, para fazer as verificações de implementação de contratos.

A linguagem de especificação Circus foi utilizada para materialização do processo proposto, por dois motivos importantes:

- Circus é uma extensão de CSP com parágrafos Z para definir ações básicas de programa, estabelecendo um hierarquia de processos, onde CSP é usada para coordenação concorrente de ações com significado funcional rigorosamente especificado. Essa hierarquia de processos tem relação com a linguagem Haskell$_{\#}$, a qual tem como premissas a separação entre a coordenação de processos funcionais da descrição da computação realizada por esses processos, descritos na linguagem funcional Haskell. Com isso, esperava-se isolar as preocupações de raciocínio sobre programas em dois níveis. No nível de coordenação, cuja semântica é baseada em CSP, foram utilizadas redes de Petri, cuja tradução da semântica de CSP havia sido investigada em trabalhos anteriores do grupo, enquanto no nível de computação, Haskell possuía propriedades que facilitavam o raciocínio sobre programas por ser uma linguagem funcional pura e não-estrita. Entretanto, com o HPE, Haskell passou a ser apenas uma alternativa para implementação da parte computacional dos componentes, levando a idéia de usar como linguagem de especificação alguma combinação de CSP com Z, linguagem de especificação de uso difundido para descrever funcionalidade de programas e garantir a capacidade de raciocínio em nível de computação. Circus atende esse requisito, além de outras combinações de Z e CSP também propostas;

- Circus possui um cálculo de refinamentos [97], o qual é essencial para o processo de derivação de componentes proposto nesta dissertação, inclusive com ferramentas já desenvolvidas para tradução de especificações Circus para programas em linguagens de programação de uso disseminado, como Java [95].

Com o processo de derivação proposto, tem-se um meio de certificação de componentes voltado para a nuvem de componentes, que consistia no objetivo principal desta dissertação.

Foi realizada ainda uma validação básica, porém representativa, do uso do método para derivação e verificação de programas reais, extraídos dos kernels do NAS Parallel Benchmarks (NPB) [18, 19]. Uma limitação importante observada no processo proposto, através dessa validação, é a complexidade da aplicação do processo proposto sobre programas considerados simples, como o IS e CG, considerado uma atividade exaustiva e intelectualmente exigente para ser usada por programadores convencionais. Espera-se que isso possa ser amenizado com a construção de ambientes de auxílio e automatização de alguns passos do processo, o que seria esperado no ambiente de nuvem de components.

6.1 Limitações e Dificuldades Encontradas

Há alguns problemas específicos relevantes que não foram tratados nesta dissertação, embora tenham sido enunciados nas propostas que lhe deram origem, descritos a seguir:

- Escolher uma técnica mais apropriada para embutir o contrato de um componente como parte do seu código objeto, de forma que o contrato não possa ser alterado ou violado. Espera-se que seja possível aplicar técnicas de *proof-carrying code*, que permitem a um programa carregar consigo informações sobre a especificação a partir do qual foi derivado, ou mesmo invariantes sobre o seu estado de execução, que podem ser usados para garantir a segurança de sua execução, bem como dos programas com os quais interage;

- Uso de esquemas de tradução específicos de acordo com a arquitetura de uma plataforma de computação paralela, tentando tomar vantagem de suas características para obter um melhor desempenho. Foi definido um esquema de tradução geral, que se aplicaria a um *cluster* convencional, sem suposições a respeito da existência de aceleradores computacionais ou peculiaridades de sua hierarquia de memória;

- Formalização de uma extensão para o sistema de tipos do HPE, o HTS, para um sistema de resolução de contratos, que permitiria a resolução dinâmica e automática de componentes baseado nas propriedades de seu contrato.

Cada um desses problemas define propostas de trabalhos futuros, que serão discutidas na Seção 6.3.

Uma das principais dificuldades encontradas foi a procura e aplicação de leis de refinamento visando uma implementação livre de chamadas recursivas de funções, na adaptação da semântica de comunicação por canais, característica da linguagem Circus, à semântica de comunicação ponto-a-ponto do MPI e na forma para tradução de uma especificação Circus/HCL para uma especificação Circus, pelo fato de que, em uma especificação Circus não é possível que um processo acesse diretamente o estado de outro, característica presente em especificações Circus/HCL, onde um componente pode acessar diretamente o estado de seus componentes aninhados.

Devemos ainda ressaltar o fato de não termos conseguido a tempo de ser apresentado nesta dissertação traduzir os contratos e especificações concretas discutidos nas Seções 5.2.4 e 5.3.4 para uma ferramenta de prova automática de teoremas, no caso o ProofPower-Z. Embora seja uma tarefa mecânica, exigindo a tradução sintática das especificações apresentadas para a linguagem dessa ferramenta, é bastante trabalhosa e susceptível a erros, razão pela qual não foi possível sua apresentação. Entretanto, entendemos não ser esse um fator que diminua o valor do trabalho apresentado, frente aos objetivos traçados,

uma vez que mostramos os passos sistemáticos de refinamento até o código C# de um componente a partir do seu contrato.

6.2 Contribuições

Entendemos como as principais contribuições desta dissertação, em ordem de importância:

- proposta de processo de derivação e verificação de componentes aplicados a uma nuvem de componentes para serviços de CAD;
- análise dos requisitos para certificação formal de componentes na plataforma da nuvem de componentes, de forma a garantir que componentes atendam as funcionalidades esperadas pelas aplicações que fazem uso deles;
- levantamento do estado-da-arte do uso de métodos formais e componentes no contexto de CAD;
- análise de limitações de linguagens de especificação, como Circus, para os requisitos de aplicações científicas, apresentando propostas de extensões, como abstrações linguísticas e açúcares sintáticos para lidar com vetores multidimensionais e números complexos e permitir uma tradução mais adequada do ponto de vista de desempenho;
- estudo de caso de derivação de programas que representam *kernels* típicos computação científica, com requisitos de CAD, utilizando Circus, originalmente desenvolvida para atender os requisitos de aplicações fora do contexto de CAD;
- Grande números de possibilidades de trabalhos futuros para serem investigados em outras dissertações, os quais estão descritos na seção a seguir.

6.3 Propostas de Trabalhos Futuros

Em termos de trabalhos futuros, do nosso ponto de vista há vários pontos que merecem ser investigados, delineados a seguir:

- Extensão dos estudos de caso, envolvendo aplicações simuladas do próprio NPB, ou mesmo programas reais, os quais podem envolver uma extensa hierarquia de componentes, permitindo estudar as propriedades composicionais do modelo proposto com Circus, algo que foi timidamente explorado nos estudos de caso desta dissertação. Para isso, poderíamos aplicar a experiência com a refatoração em componentes HPE das aplicações simuladas do NPB (SP, BT, LU e FT) [108];

- Refinamento do modelo proposto para incorporar a possibilidade de múltiplos alvos de tradução, representados em esquemas múltiplos que poderiam ser suportados por um ambiente de programação, cada qual aplicado a uma certa arquitetura de computação paralela, levando em conta suas peculiaridades, inclusive a presença de aceleradores computacionais e diferentes hierarquias de memória;

- Projeto e implementação de um ambiente para derivação e verificação de componentes baseado no modelo proposto, que possa ser disponibilizado através da plataforma idealizada de nuvem de componentes, o que envolve o estudo de técnicas e abstrações para reduzir a complexidade dessas tarefas. Para isso, é importantíssimo os resultados obtidos com os estudos de caso propostos;

- Proposta de técnica adequada para que um componente possa carregar sua especificação concreta de forma segura, ou seja que possa garantir, em certo grau conhecido, que o código objeto do componente foi derivado daquela especificação concreta e seja possível verificar automaticamente sua compatibilidade com o contrato requerido pela aplicação;

- Extensão do sistema de tipos HTS (*Hash Type System*), proposto para o HPE, com o uso de contratos de componentes ao invés de componentes abstratos, incorporando propriedades semânticas do componentes às propriedades sintáticas atualmente suportadas pelo HTS para ligação automática e dinâmica de componentes.

Referências Bibliográficas

[1] FDR2. Disponível em: `http://www.fsel.com/`. Acessado em 08 de Agosto de 2012.

[2] Top500. Disponível em: `http://www.top500.org/`. Acessado em 13 de Junho de 2012.

[3] GridCOMP - Grids Programming with Components. An Advanced Component Platform for an Effective Invisible Grid. Disponível em: `http://gridcomp.ercim.eu/`. Acessado em 06 de Fevereiro de 2012., 2009.

[4] MxN Research @ Indiana University. Disponível em: `http://www.cs.indiana.edu/~febertra/mxn/`. Acessado em 08 de Julho de 2012., 2009.

[5] npb-for-hpe - Implementation of the NAS Parallel Benchmarks for Evaluating Performance of HPE - Google Project Hosting. http://npb-for-hpe.googlecode.com/, Aug. 2011.

[6] Fractal - Home Page, July 2012.

[7] Allan, B. A., Armstrong, R. C., Wolfe, A. P., Ray, J., Bernholdt, D. E., and Kohl, J. A. The CCA Core Specification in a Distributed Memory SPMD Framework. *Concurrency and Computation: Practice and Experience 14*, 5 (2002), 323–345.

[8] Allen, R., and Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology 6*, 3 (Jul 1997), 213–249.

[9] Allen, R. J. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1997. AAI9813815.

[10] Amedro, B., Baude, F., Caromel D., Delbe, C., Filali, I., Huet, F., Mathias, E., and Smirnov O. *An Efficient Framework for Running Applications on Clusters, Grids and Clouds*. Springer, 2010, chapter 10, pp. 163–178.

[11] Amedro, B., Caromel, D., Huet, F., Bodnartchouk, V., Delbé, C., and Taboada, G. HPC in Java: Experiences in Implementing the NAS Parallel Benchmarks. In *Proceedings of the 10th WSEAS International Conference on APPLIED INFORMATICS AND COMMUNICATIONS (AIC '10)* (Aug. 2010).

[12] Antonopoulos, N., and Gillam, L. *Cloud Computing: Princiles, Systems and Applications*. Computer Commmunications and Networks. Springer, 2011.

[13] Arbab, F. Reo: A Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science 14*, 3 (2004), 329–366.

[14] Armstrong, R., Kumfert, G., Mcinnes, L. C., Parker, S., Allan, B., Sottile, M., Epperly, T., and Dahlgren, T. The CCA Component Model for High-Performance Scientific Computing. *Concurrency and Computation: Practice and Experience 18*, 2 (2006), 215–229.

[15] Arthan, R. ProofPower. Disponível em: `http://www.lemma-one.com/ProofPower/index/index.html`. Acessado em 15 de Agosto de 2012.

[16] Back, R. J. R. Procedural Abstraction in the Refinement Calculus. Tech. Rep. 55, Åbo Akademi, Åbo, Finland, 1987.

[17] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications 5*, 3 (1991), 66–73.

[18] Bailey, D. H., and et al. The NAS Parallel Benchmarks. *International Journal of Supercomputing Applications 5*, 3 (1991), 63–73.

[19] Bailey, D. H., Harris, T., Shapir, W., van der Wijngaart, R., Woo, A., and Yarrow, M. The NAS Parallel Benchmarks 2.0. Tech. Rep. NAS-95-020, NASA Ames Research Center, Dec. 1995.

[20] Barbosa, L. S. *Components as Coalgebras*. PhD thesis, Universidade do Minho, Departamento de Informática, 2001.

[21] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., and Pérez, C. GCM: a Grid Extension to Fractal for Autonomous Distributed Components. *Annals of Telecommunications 64* (2009), 5–24.

[22] Baude, F., Caromel, D., and Morel, M. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications* (2003), Springer-Verlag, pp. 1226–1242.

[23] Becker, D. J., Sterling, T., Savarese, D., Dorban, J. E., Ranawak, U. A., and Packer, C. V. Bewoulf: A Parallel Workstation for Scientific Computation. In *1995 International Conference on Parallel Processing* (1995).

[24] Berthomieu, B., Ribet, P. O., and Vernadat, F. The Tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research 42* (July 15, 2004), 2741–2756(16).

[25] Bertran, F., Bramley, R., Sussman, A., Bernholdt, D. E., Kohl, J. A., Larson, J. W., and Damevski, K. B. Data Redistribution and Remote Method Invocation in Parallel Component Architectures. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Apr. 2005), IEEE.

[26] Bertrand, F., and Bramley, R. DCA: A Distributed CCA Framework Based on MPI. In *Proceedings of the HIPS2004 - 9th International Workshop on Highlevel Parallel Programming Models and Supportive Environments* (2004).

[27] Bode, A., Dongarra, J., Ludwig, T., and Sunderam, V. S., Eds. *Parallel Virtual Machine - EuroPVM'96, Third European PVM Conference, München, Germany, October 7-9, 1996, Proceedings* (1996), vol. 1156 of *Lecture Notes in Computer Science*, Springer.

[28] Bruneton, E., Coupaye, T., and Stefani, J. Recursive and Dynamic Software Composition with Sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02)* (2002), European Conference on Object-Oriented Programming.

[29] Carvalho Junior, F. H. *Programação Paralela Eficiente e de Alto Nível sobre Arquiteturas Distribuídas.* PhD thesis, Centro de Informática, Universidade Federal de Pernambuco, 2003.

[30] Carvalho Junior, F. H., and Correa, R. C. The Design of a CCA Framework with Distribution, Parallelism, and Recursive Composition. In *Proceedings of the 2010 Workshop on Component-Based High Performance Computing (CBHPC'10)* (2010).

[31] Carvalho Junior, F. H., Lima, R. M. F., and Lins, R. D. Coordinating Functional Processes with Haskell#. In *Proceedings of the 2002 ACM symposium on Applied computing* (New York, NY, USA, 2002), SAC '02, ACM, pp. 393–400.

[32] Carvalho Junior, F. H., Lins, R., Correa, R. C., and Araújo, G. A. Towards an Architecture for Component-Oriented Parallel Programming. *Concurrency and Computation: Practice and Experience 19*, 5 (2007), 697–719.

[33] Carvalho Junior, F. H., and Lins, R. D. Haskell$_{\#}$: Parallel Programming Made Simple and Efficient. *Journal of Universal Computer Science 9*, 8 (Aug. 2003), 776–794.

[34] Carvalho Junior, F. H., and Lins, R. D. On the Implementation of SPMD Applications using Haskell$_{\#}$. In *15th Brazilian Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2003)* (Nov. 2003), IEEE Press.

[35] Carvalho Junior, F. H., and Lins, R. D. Separation of Concerns for Improving Practice of Parallel Programming. *INFORMATION, An International Journal 8*, 5 (Sept. 2005).

[36] Carvalho Junior, F. H., and Lins, R. D. An Institutional Theory for #-Components. *Electronic Notes in Theoretical Computer Science 195* (Jan. 2008), 113–132.

[37] Carvalho Junior, F. H., and Lins, R. D. Compositional Specification of Parallel Components Using Circus. *Electronic Notes in Theoretical Computer Science 260* (2010), 47 – 72. Proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008).

[38] Carvalho Junior, F. H., Lins, R. D., Correa, R. C., and Araújo, G. A. Towards an Architecture for Component-Oriented Parallel Programming. *Concurrency and Computation: Practice and Experience 19*, 5 (2007), 697–719.

[39] Carvalho Junior, F. H., Lins, R. D., and Lima, R. M. F. Parallelising MCP-Haskell$_{\#}$ for Evaluating *Haskell$_{\#}$* Parallel Programming Environment. In *13th Brazilian Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD 2001)* (Sept. 2001), UnB, Ed.

[40] Carvalho Junior, F. H., Lins, R. D., and Lima, R. M. F. Translating Haskell$_{\#}$ Programs into Petri Nets. *Lecture Notes in Computer Science (VECPAR'2002) 2565* (2002), 635–649.

[41] Carvalho Junior, F. H., Lins, R. D., and Lima, R. M. F. Translating Haskell# Programs into Petri Nets. In *Proceedings of the 5th international conference on High performance computing for computational science* (Berlin, Heidelberg, 2003), VECPAR'02, Springer-Verlag, pp. 635–649.

[42] Cavalcanti, A. L. C. *A Refinement Calculus for Z*. PhD thesis, Oxford University Computing Laboratory, Oxford - UK, 1997.

[43] Cavalcanti, A. L. C., Sampaio, A. C., and Woodcock, J. C. P. Refinement of Actions in Circus. *Electronic Notes in Theoretical Computer Science 70*, 3 (2002), 132 – 162. REFINE 2002, The BCS FACS Refinement Workshop (Satellite Eventof FLoC 2002).

[44] Cirne, W., Paranhos, D., Costa, L., Santos Neto, E., Brasileiro, F., Sauv, J., Silva, F., Barros C., and Silveira, C. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proceedings of the ICCP'2003 - International Conference on Parallel Processing* (2003).

[45] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst. 8* (April 1986), 244–263.

[46] Cole, M. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing 30*, 3 (2004), 389–406.

[47] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[48] Dijkstra, E. W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM 18* (August 1975), 453–457.

[49] Division, N. A. S. Nas parallel benchmarks. Disponível em: `http://www.nas.nasa.gov/publications/npb.html`. Acessado em 01 de Maio de 2012.

[50] Dongarra, J. Trends in High Performance Computing. *The Computer Journal 47*, 4 (2004), 399–403.

[51] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., and White, A. *Sourcebook of Parallel Computing*. Morgan Kauffman Publishers, 2003, ch. 20-21.

[52] Dongarra, J., Otto, S. W., Snir, M., and Walker, D. A Message Passing Standard for MPP and Workstation. *Communications of ACM 39*, 7 (1996), 84–90.

[53] Ekanayake, J., and Fox, G. High Performance Parallel Computing with Clouds and Cloud Technologies. In *CloudComp* (2009), vol. 34 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer, pp. 20–38.

[54] Elgar, T. Intel Many Integrated Core (MIC) Architecture. In *2nd UK GPU Computing Conference* (December 2010).

[55] Fan, Z., Qiu, F., Kaufman, A., and Yoakum Stover, S. GPU Cluster for High Performance Computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC'04)* (2004), IEEE Computer Society, pp. 47–47.

[56] Feng, X., Ni, Z., Shao, Z., and Guo, Y. An Open Framework for Foundational Proof-Carrying Code. In *Proceedings the 2007 ACM Sigplan International Workshop on Types in Language Design and Implementation (TLDI '07* (2007), ACM Press, pp. 67–78.

[57] Fiadeiro, J. L. *Categories for Software Engineering.* Springer, 2005.

[58] Fischer, C. Combining CSP and Z. Tech. Rep. TRCF-97-1, University of Oldenburg, 1997.

[59] Franck Cappello and Thomas Hérault and Jack Dongarra, Ed. *Verifying Parallel Programs with MPI-Spin* (Paris, France, 2007), vol. 4757, Springer.

[60] Freitas, A., and Cavalcanti, A. L. C. Automatic Translation from Circus to Java. In *Lecture Notes in Computer Science: FM 2006: Formal Methods* (2006), Springer, pp. 115–130.

[61] Gischer, J. Shuffle Languages, Petri Nets, and Context-Sensitive Grammars. *Communications of the ACM 24*, 9 (Sept. 1981), 597–605.

[62] Gopalakrishnan, G. L., and Kirby, R. M. Top Ten Ways to Make Formal Methods for HPC Practical. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (New York, NY, USA, 2010), FoSER '10, ACM, pp. 137–142.

[63] Grahlmann, B., and Best, E. PEP - More Than a Petri Net Tool. In *Lecture Notes in Computer Science (Tools and Algorithms for the Construction and Analysis of Systems, Second Int. Workshop, TACAS'96, Passau, Germany)* (Mar. 1996), vol. 1055, Springer Verlag, pp. 397–401.

[64] Grama, A., Gupta, A., Karypis, J., and Kumar, V. *Introduction to Parallel Computing.* Addison-Wesley, 1976.

[65] Green, C. Application of Theorem Proving to Problem Solving. In *Proceedings of the 1st international joint conference on Artificial intelligence* (San Francisco, CA, USA, 1969), Morgan Kaufmann Publishers Inc., pp. 219–239.

[66] Herbordt, M. C., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., and DiSabello, D. Achieving High Performance with FPGA-Based Computing. *Computer 40* (March 2007), 50–57.

[67] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[68] Hoare, C. A. R., and Jifeng, H. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[69] Holzmann, G. J. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.

[70] IBM. IBM Java Development Kit, May 2012.

[71] IBM. Jikes RVM - Home, May 2012.

[72] Inmos. Occam Programming Manual. *Prentice-Hall, C.A.R. Hoare Series Editor* (1984).

[73] Ito, T., and Nishitani, Y. On Universality of Concurrent Expressions with Synchronization Primitives. *Theoretical Computer Science 19* (1982), 105–115.

[74] Jacky, J. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, New York, NY, USA, 1996.

[75] Jacob, B., Larson, J., and Ong, E. M×N Communication and Parallel Interpolation in Community Climate System Model Version 3 Using the Model Coupling Toolkit. *The International Journal of High Performance Computing Applications 19*, 3 (2005), 293–307.

[76] Jensen, K., Kristensen, L., and Wells, L. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT) 9* (2007), 213–254. 10.1007/s10009-007-0038-x.

[77] Krishnan, S., and Gannon, D. XCAT3: A Framework for CCA Components as OGSA Services. In *Proceedings of the HIPS2004 - 9th International Workshop on Highlevel Parallel Programming Models and Supportive Environments* (2004).

[78] Lamport, L., Matthews, J., Tuttle, M., and Yu, Y. Specifying and verifying systems with tla+. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (New York, NY, USA, 2002), EW 10, ACM, pp. 45–48.

[79] Lau, K., Elizondo, P. V., and Wang, Z. Exogenous Connectors for Software Components. *Lecture Notes in Computer Science (Proceedings of 2005 International SIGSOFT Symposium on Component-Based Software Engineering - CBSE'2005) 3489* (2005), 90–108.

[80] Laudon, J., and Spracklen, L. The Coming Wave of Multithreaded Chip Multiprocessors. *International Journal of Parallel Programming 35* (2007), 299–330.

[81] Lemma 1 Ltd. ProofPower-Z. Disponível em: `http://www.lemma-one.com/ProofPower/doc/usr030.pdf`. Acessado em 15 de Agosto de 2012.

[82] Lima, R. M. F., Carvalho Junior, F. H., and Lins, R. D. *Haskell$_{\#}$*: A Message Passing Extension to Haskell. In *3rd Latin American Conference on Functional Programming* (Mar. 1999), pp. 93–108.

[83] Lima, R. M. F., and Lins, R. D. Translating HCL Programs into Petri Nets. In *Proceedings of the 14th Brazilian Symposium on Software Engineering* (2000).

[84] Mahmood, N. *Productivity with Performance: Property/Behavior-Based Automated Composition of Parallel Programs From Self-Describing Components.* PhD thesis, University of Texas at Austin, Austin, TX, USA, 2007.

[85] Mahony, B., and Dong, J. S. Blending Object-Z and Timed CSP: an Introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering* (1998), ICSE '98, IEEE Computer Society, pp. 95–104.

[86] Malawski, M., Meizner, J., Bubak, M., and Gepner, P. Component Approach to Computational Applications on Clouds. *Procedia Computer Science 4* (may 2011), 432–441.

[87] Mateescu, G., Gentzsch, W., and Ribbens, C. J. Hybrid Computing - Where HPC meets grid and Cloud Computing. *Future Generation Computer Systems 27*, 5 (may 2011), 440–453.

[88] Meng, S., and Barbosa, L. S. On Refinement of Generic Software Components. In *In 10th International Conference Algebraic Methodology and Software Technology (AMAST)* (2003), Springer, pp. 506–520.

[89] Monin, J. F. *Understanding Formal Methods.* Springer, Secaucus, NJ, USA, 2001.

[90] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., and Vanderheyden, P. *Eclipse Development Using the Graphical Editing Framework and Eclipse Modelling Framework.* IBM International Technical Support Organization, Feb. 2004. http://www.ibm.com/redbooks.

[91] Morgan, C. The Specification Statement. *ACM Trans. Program. Lang. Syst. 10*, 3 (July 1988), 403–419.

[92] NAS - Advanced Supercomputing Division. NPB for HPE Project. Disponível em: `http://npb-for-hpe.googlecode.com/`. Acessado em 10 de Março de 2012., 2011.

[93] Necula, G. C. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1997), POPL '97, ACM, pp. 106–119.

[94] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'2009)* (May 2009), pp. 124–131.

[95] Oliveira, M. *Formal Derivation of State-Rich Reactive Programs Using Circus.* PhD thesis, Department of Computer Science, University of York, 2005.

[96] Oliveira, M., Cavalcanti, A. L. C., and Woodcock, J. C. P. A UTP semantics for Circus. *Formal Aspects of Computing 21*, 1-2 (feb 2009), 3–32.

[97] Oliveira, M. V. M., Gurgel, A. C., and Castro, C. G. CRefine: Support for the Circus Refinement Calculus. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 281–290.

[98] OpenMP Architecture Review Board. OpenMP: Simple, Portable, Scalable SMP Programming. www.openmp.org, 1997.

[99] Oracle. OpenJDK, May 2012.

[100] Oracle. Oracle Java Development Kit, May 2012.

[101] Paulson, L. C. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning 5* (1989).

[102] Pervez, S., Gopalakrishnan, G. L., Kirby, R. M., Thakur, R., and Gropp, W. Formal Methods Applied to High-performance Computing Software Design: A Case Study of MPI One-Sided Communication-Based Locking. *Software: Practice and Experience* (2010), 23–43.

[103] Peterson, J. L. *Petri Net Theory and The Modeling of Systems.* Foundations of Philosophy Series. Prentice-Hall, 1981.

[104] Petri, C. A. Kommunikation Mit Automaten. *Technical Report RADC-TR-65-377, Griffiths Air Force Base, New York 1*, 1 (1966).

[105] Peyton Jones, S. L., Hall, C., Hammond, K., and Partain, W. The Glasgow Haskell Compiler: a Technical Overview. *Joint Framework for Information Technology Technical Conference* (1993), 249–257.

[106] Plasil, F., and Visnovsky, S. Behavior Protocols for Software Components . *IEEE Transactions on Software Engineering 28*, 11 (2002), 1056–1076.

[107] Post, D. E., and Votta, L. G. Computational Science Demands a New Paradigm. *Physics Today 58*, 1 (2005), 35–41.

[108] Rezende, C. A., and Carvalho Junior, F. H. d. Component-Based Refactoring of Parallel Numerical Simulation Programs: A Case Study on Component-Based Parallel Programming. In *Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2011)* (Oct. 2011), Vitória, Brazil, IEEE Computer Society.

[109] Roch, S., and Starke, P. *Manual: Integrated Net Analyzer Version 2.2*, 1999.

[110] Rumbaugh, J., Jacobson, I., and Booch, G. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[111] Rutten, J. J. M. M. Universal Coalgebra: A Theory of Systems. *Theoretical Computer Science 249*, 1 (Oct. 2000), 3–80.

[112] Sampaio, A. C. A., Woodcock, J. C. P., and Cavalcanti, A. L. C. Refinement in Circus. *Lecture Notes in Computer Science 2391* (2002), 451–470. Formal Methods Europe (FME) 2002.

[113] Shaw, M. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *International Workshop on Studies of Software Design* (1994), vol. 1078 of *Lecture Notes in Computer Science*, Springer-Verlag.

[114] Siegel, S. F. Model Checking Nonblocking MPI Programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)* (2007), Springer, pp. 44–58.

[115] Skjellum, A., Bangalore, P., Gray, J., and Bryant B. Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software. In *International Workshop on Software Engineering for High Performance Computing System Applications* (May 2004), ACM, pp. 59–63. Edinburgh.

[116] Snir, M., Otto, S., Huss Lederman, S., Walker, D., and Dongarra, J. *MPI: The Complete Reference*. MIT Press Cambridge, 1995.

[117] Stephen, B., and John, N. Z Base Standard, Verson 1.0. Tech. Rep. 107, Oxford University, 1992.

[118] Stickel, M. E., Waldinger, R. J., Lowry, M. R., Pressburger, T., and Underwood, I. Deductive Composition of Astronomical Software from Subroutine Libraries. In *Proceedings of the 12th International Conference on Automated Deduction* (London, UK, 1994), CADE-12, Springer-Verlag, pp. 341–355.

[119] Taboada, G. L., Ramos, S., Expósito, R. R., Tourino, J., and Doallo, R. Java in the High Performance Computing arena: Research, practice and experience. *Science of Computer Programming ?*, ? (2011), ?

[120] Takizawa, H., and Kobayashi, H. Hierarchical Parallel Processing of Large Scale Data Clustering on a PC Cluster with GPU Co-Processing. *Journal of Supercomputing 36* (June 2006), 219–234.

[121] Thompson, S. *Haskell, The Craft of Functional Programming*. Addison-Wesley Publishers Ltd., 1996.

[122] Vakkalanka, S. S., Sharma, S., Gopalakrishnan, G. L., and Kirby, R. M. ISP: a Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 285–286.

[123] van der Steen, A. J. Issues in Computational Frameworks. *Concurrency and Computation: Practice and Experience 18*, 2 (2006), 141–150.

[124] Vecchiola, C., Pandey, S., and Buyya, R. High-Performance Cloud Computing: A View of Scientific Applications. In *ISPAN* (2009), IEEE Computer Society, pp. 4–16.

[125] Vecchiola, C., Pandey, S., and Buyya, R. High-Performance Cloud Computing: A View of Scientific Applications. In *Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN'09)* (2009), IEEE Computer Society, pp. 4–16.

[126] Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G. L., Kirby, R. M., and Thakur, R. Formal Verification of Practical MPI Programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 261–270.

[127] Wang, A. J. A., and Qian, K. *Component-Oriented Programming.* Wiley-Interscience, 2005.

[128] Woodcock, J. C. P., and Cavalcanti, A. L. C. A Concurrent Language for Refinement. In *5th Irish Workshop on Formal Methods* (2001), Butterfield, Andrew and Strong, Glenn and Pahl, Claus, Ed., Workshops in Computing.

[129] Woodcock, J. C. P., and Cavalcanti, A. L. C. The Semantics of *Circus*. In *ZB 2002: Formal Specification and Development in Z and B* (2002), D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, Eds., vol. 2272 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 184—203.

[130] Woodcock, J. C. P., and Cavalcanti, A. L. C. Circus Website. Disponível em: `http://www.cs.york.ac.uk/circus/`. Acessado em 03 de Março de 2012., 2008.

[131] Zeyda, F., and Cavalcanti, A. L. C. Encoding circus programs in ProofPower-Z. In *Proceedings of the 2nd international conference on Unifying theories of programming* (Berlin, Heidelberg, 2010), UTP'08, Springer-Verlag, pp. 218–237.

[132] Zhang, K., Damevski, K., Venkatachalapathy, V., and Parker, S. SCIRun2: A CCA Framework for High Performance Computing. In *Proceedings of the HIPS2004 - 9th International Workshop on Highlevel Parallel Programming Models and Supportive Environments* (2004).

[133] Zhang, L., and Parashar, M. Enabling Efficient and Flexible Coupling of Parallel Scientific Applications. In *20th International of Parallel and Distributed Processing Symposium (IPDPS'2006)* (2006), IEEE Computer Society, pp. 117–127.

Apêndice A

Obrigações de Prova

Neste apêndice são demonstradas todas as obrigações de prova desta dissertação.

OP.1 Obrigação de Prova OP.1

$r::k = \sum_{i=0}^{x-1} v::v(i) \times u::v(i) \wedge inv$
$\Rightarrow r::k + v::v(x) \times u::v(x) = \sum_{i=0}^{x} v::v(i) \times u::v(i) \wedge inv$

Como premissa, temos $r::k = \sum_{i=0}^{x-1} v::v(i) \times u::v(i)$. Dessa forma, temos que:

$r::k + v::v(x) \times u::v(x) =$
$\sum_{i=0}^{x-1}(v::v(i) \times u::v(i)) + v::v(x) \times u::v(x) =$
$\sum_{i=0}^{x} v::v(i) \times u::v(i)$
$\square$

OP.2 Obrigação de Prova OP.2

$p_2[\_/'] \wedge p_3[\_/'] \wedge \forall\, l : 0..(j-1) \bullet ord[l] = 0$
$\Rightarrow (p_2 \wedge p_3 \wedge (\forall\, l : 0..j \bullet ord[l]' = 0))[0/ord[j]'][\_/']$

Partindo da premissa dada, tem-se que:

$p_2[\_/'] \wedge p_3[\_/'] \wedge \forall\, l : 0..(j-1) \bullet ord[l] = 0$
$\Rightarrow p_2[\_/'] \wedge p_3[\_/'] \wedge (\forall\, l : 0..(j-1) \bullet ord[l] = 0) \wedge (0 = 0)$
$\Rightarrow$(dado que $ord[j]'$ não aparece nas proposições $p_2[\_/'] \cup p_3[\_/']$, dado que só há variáveis não decoradas em tais proposições)
$(p_2[\_/'] \wedge p_3[\_/'] \wedge (\forall\, l : 0..(j-1) \bullet ord[l] = 0) \wedge ord[j]' = 0)[0/ord[j]']$
$\Rightarrow (p_2 \wedge p_3 \wedge (\forall\, l : 0..(j-1) \bullet ord[l]' = 0) \wedge ord[j]' = 0)[0/ord[j]'][\_/']$

$\Rightarrow (p_2 \wedge p_3 \wedge (\forall\, l : 0..j \bullet ord[l]' = 0))[0/ord[j]'][\_/']$

$\square$

OP.3 Obrigação de Prova OP.3

$\forall\, j : b_0..m-1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux$
$\Rightarrow (\forall\, j : b_0..m \bullet rank[j]' = (\sum_{l=b_0}^{j-1} ord[l]) + aux)$
$[rank[m-1] + ord[m-1]/rank[m]'][\_/']$

Partindo da premissa dada, tem-se que:

$\forall\, j : b_0..m-1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux$
$\Rightarrow (\forall\, j : b_0..m-1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux) \wedge$
$rank[m-1] + ord[m-1] = rank[m-1] + ord[m-1]$
(dado que $rank[m]'$ não aparece em nenhuma das proposições da fórmula)
$\Rightarrow ((\forall\, j : b_0..m-1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux) \wedge$
$rank[m]' = rank[m-1] + ord[m-1])[rank[m-1] + ord[m-1]/rank[m]']$
$\Rightarrow ((\forall\, j : b_0..m-1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux) \wedge$
$rank[m]' = ((\sum_{l=b_0}^{m-2} ord[l]) + aux) + ord[m-1])[rank[m-1] + ord[m-1]/rank[m]']$
$\Rightarrow ((\forall\, j : b_0..m-1 \bullet rank[j] = (\sum_{l=b_0}^{j-1} ord[l]) + aux) \wedge$
$rank[m]' = ((\sum_{l=b_0}^{m-1} ord[l]) + aux))[rank[m-1] + ord[m-1]/rank[m]']$
$\Rightarrow ((\forall\, j : b_0..m-1 \bullet rank[j]' = (\sum_{l=b_0}^{j-1} ord[l]) + aux) \wedge$
$rank[m]' = ((\sum_{l=b_0}^{m-1} ord[l]) + aux))[rank[m-1] + ord[m-1]/rank[m]'][\_/']$
$\Rightarrow (\forall\, j : b_0..m \bullet rank[j]' = (\sum_{l=b_0}^{j-1} ord[l]) + aux)$
$[rank[m-1] + ord[m-1]/rank[m]'][\_/']$

$\square$

OP.4 Obrigação de Prova OP.4

$\forall\, j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k]$
$\Rightarrow (\forall\, j : 0..m-1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m]' = 0)[0/x[m]'][\_/']$

Partindo da premissa, tem-se:

$\forall\, j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k]$
$\Rightarrow \forall\, j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge 0 = 0$
Dado que $x[m]'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow (\forall\, j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m]' = 0)[0/x[m]']$
$\Rightarrow (\forall\, j : 0..m-1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k] \wedge x[m]' = 0)[0/x[m]'][\_/']$

$\square$

OP.5 Obrigação de Prova OP.5

$$(\forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k]) \wedge x[m] = \sum_{k=0}^{l-1} A[m,k] \times v[k]$$
$$\Rightarrow ((\forall j : 0..m-1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k]) \wedge$$
$$x[m]' = \sum_{k=0}^{l} A[m,k] \times v[k])[x[m] + A[m,l] \times v[l]/x[m]'][\_/']$$

Partindo da premissa dada, tem-se que:

$$(\forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k]) \wedge x[m] = \sum_{k=0}^{l-1} A[m,k] \times v[k]$$
$$\Rightarrow (\forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k]) \wedge x[m] + A[m,l] \times v[l] =$$
$$(\sum_{k=0}^{l-1} A[m,k] \times v[k]) + A[m,l] \times v[l]$$

Dado que $x[m]'$ não aparece em nenhuma das proposições da fórmula

$$\Rightarrow ((\forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k]) \wedge$$
$$x[m]' = (\sum_{k=0}^{l-1} A[m,k] \times v[k]) + A[m,l] \times v[l])[x[m] + A[m,l] \times v[l]/x[m]']$$
$$\Rightarrow ((\forall j : 0..m-1 \bullet x[j] = \sum_{k=0}^{dim-1} A[j,k] \times v[k]) \wedge$$
$$x[m]' = (\sum_{k=0}^{l} A[m,k] \times v[k]))[x[m] + A[m,l] \times v[l]/x[m]']$$
$$\Rightarrow ((\forall j : 0..m-1 \bullet x[j]' = \sum_{k=0}^{dim-1} A[j,k] \times v[k]) \wedge$$
$$x[m]' = (\sum_{k=0}^{l} A[m,k] \times v[k]))[x[m] + A[m,l] \times v[l]/x[m]'][\_/']$$

□

OP.6 Obrigação de Prova OP.6

$$true \Rightarrow (i = N-1) \vee (i \neq N-1)$$

Para provar a implicação, basta provar $(i = N-1) \vee (i \neq N-1)$. A proposição $(i = N-1) \vee (i \neq N-1)$ é verdadeira pela lei do terceiro excluído.

□

OP.7 Obrigação de Prova OP.7

$$i = N-1$$
$$\Rightarrow ((i = N-1 \wedge lines' = dim - (N-1) \times \lfloor \tfrac{dim}{N} \rfloor)$$
$$\vee (i \neq N-1 \wedge lines' = \lfloor \tfrac{dim}{N} \rfloor))[dim - (N-1) \times \lfloor dim/N \rfloor / lines'][\_/']$$

Partindo da premissa, tem-se que:

$$i = N-1$$
$$\Rightarrow i = N-1 \wedge dim - (N-1) \times \lfloor \tfrac{dim}{N} \rfloor = dim - (N-1) \times \lfloor \tfrac{dim}{N} \rfloor$$
$$\Rightarrow (i = N-1 \wedge dim - (N-1) \times \lfloor \tfrac{dim}{N} \rfloor = dim - (N-1) \times \lfloor \tfrac{dim}{N} \rfloor) \vee$$
$$(i \neq N-1 \wedge dim - (N-1) \times \lfloor \tfrac{dim}{N} \rfloor = \lfloor \tfrac{dim}{N} \rfloor)$$

Dado que $lines'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((i = N-1 \wedge lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor)$
$\quad \vee (i \neq N-1 \wedge lines' = \lfloor \frac{dim}{N} \rfloor))[dim - (N-1) \times \lfloor dim/N \rfloor / lines']$
$\Rightarrow ((i = N-1 \wedge lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \vee$
$\quad (i \neq N-1 \wedge lines' = \lfloor \frac{dim}{N} \rfloor))[dim - (N-1) \times \lfloor dim/N \rfloor / lines'][\_/']$
$\square$

OP.8 Obrigação de Prova OP.8

$i \neq N-1$
$\Rightarrow ((i \neq N-1 \wedge lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \vee$
$\quad (i \neq N-1 \wedge lines' = \lfloor \frac{dim}{N} \rfloor))[\lfloor dim/N \rfloor \rfloor / lines'][\_/']$

Partindo da premissa, tem-se que:

$i \neq N-1$
$\Rightarrow i \neq N-1 \wedge \lfloor \frac{dim}{N} \rfloor = \lfloor \frac{dim}{N} \rfloor$
$\Rightarrow (i = N-1 \wedge \lfloor \frac{dim}{N} \rfloor = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \vee$
$\quad (i \neq N-1 \wedge dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor = \lfloor \frac{dim}{N} \rfloor)$
Dado que $lines'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((i = N-1 \wedge lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \vee$
$\quad (i \neq N-1 \wedge lines' = \lfloor \frac{dim}{N} \rfloor))[\lfloor \frac{dim}{N} \rfloor / lines']$
$\Rightarrow ((i = N-1 \wedge lines' = dim - (N-1) \times \lfloor \frac{dim}{N} \rfloor) \vee$
$\quad (i \neq N-1 \wedge lines' = \lfloor \frac{dim}{N} \rfloor))[\lfloor \frac{dim}{N} \rfloor / lines'][\_/']$
$\square$

OP.9 Obrigação de Prova OP.9

$\forall j : 0..m-1 \bullet z[j] = 0 \Rightarrow (\forall j : 0..m \bullet z[j]' = 0)[0/z[m]'][\_/']$

Partindo da premissa, tem-se que:

$\forall j : 0..m-1 \bullet z[j] = 0 \Rightarrow (\forall j : 0..m-1 \bullet z[j] = 0) \wedge (0 = 0)$
Dado que $z[m]'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((\forall j : 0..m-1 \bullet z[j] = 0) \wedge (z[m]' = 0))[0/z[m]']$
$\quad ((\forall j : 0..m-1 \bullet z[j]' = 0) \wedge (z[m]' = 0))[0/z[m]'][\_/']$
$\Rightarrow (\forall j : 0..m \bullet z[j]' = 0)[0/z[m]'][\_/']$
$\square$

OP.10 Obrigação de Prova OP.10

$$\forall j : 0..m-1 \bullet r[j] = x[j] \Rightarrow (\forall j : 0..m \bullet r[j]' = x[j])[x[m]/r[m]'][\_/']$$

Partindo da premissa, tem-se que:

$\forall j : 0..m-1 \bullet r[j] = x[j] \Rightarrow (\forall j : 0..m-1 \bullet r[j] = x[j]) \wedge (x[m] = x[m])$
Dado que $r[m]'$ não aparece em nenhuma das proposições da fórmula
$((\forall j : 0..m-1 \bullet r[j] = x[j]) \wedge (r[m]' = x[m]))[x[m]/r[m]']$
$\Rightarrow ((\forall j : 0..m-1 \bullet r[j]' = x[j]) \wedge (r[m]' = x[m]))[x[m]/r[m]'][\_/']$
$\Rightarrow (\forall j : 0..m \bullet r[j]' = x[j])[x[m]/r[m]'][\_/']$
□

OP.11 Obrigação de Prova OP.11

$$\begin{aligned}
&((\forall j : 0..m-1 \bullet \forall k : 0..dim-1 \bullet mv::A[j,k] = A[j,k]) \wedge \\
&(\forall j : 0..m \bullet \forall k : 0..l-1 \bullet mv::A[j,k] = A[j,k])) \\
&\Rightarrow ((\forall j : 0..m-1 \bullet \forall k : 0..dim-1 \bullet mv::A[j,k]' = A[j,k]) \wedge \\
&\quad (\forall j : 0..m \bullet \forall k : 0..l \bullet mv::A[j,k]' = A[j,k]))[A[m,l]/mv::A[m,l]'][_/']
\end{aligned}$$

Partindo da premissa, tem-se que:

$$\begin{aligned}
&((\forall j : 0..m-1 \bullet \forall k : 0..dim-1 \bullet mv::A[j,k] = A[j,k]) \wedge \\
&(\forall j : 0..m \bullet \forall k : 0..l-1 \bullet mv::A[j,k] = A[j,k])) \\
&\Rightarrow ((\forall j : 0..m-1 \bullet \forall k : 0..dim-1 \bullet mv::A[j,k] = A[j,k]) \wedge \\
&\quad (\forall j : 0..m \bullet \forall k : 0..l-1 \bullet mv::A[j,k] = A[j,k]) \wedge (A[m,l] = A[m,l]))
\end{aligned}$$

Dado que $mv::A[m,l]'$ não aparece em nenhuma das proposições da fórmula

$$\begin{aligned}
&\Rightarrow ((\forall j : 0..m-1 \bullet \forall k : 0..dim-1 \bullet mv::A[j,k] = A[j,k]) \wedge \\
&\quad (\forall j : 0..m \bullet \forall k : 0..l-1 \bullet mv::A[j,k] = A[j,k]) \wedge \\
&\quad (mv::A[m,l]' = A[m,l]))[A[m,l]/mv::A[m,l]'] \\
&\Rightarrow ((\forall j : 0..m-1 \bullet \forall k : 0..dim-1 \bullet mv::A[j,k]' = A[j,k]) \wedge \\
&\quad (\forall j : 0..m \bullet \forall k : 0..l-1 \bullet mv::A[j,k]' = A[j,k]) \wedge \\
&\quad (mv::A[m,l]' = A[m,l]))[A[m,l]/mv::A[m,l]'][_/'] \\
&\Rightarrow ((\forall j : 0..m-1 \bullet \forall k : 0..dim-1 \bullet mv::A[j,k]' = A[j,k]) \wedge \\
&\quad (\forall j : 0..m \bullet \forall k : 0..l \bullet mv::A[j,k]' = A[j,k]))[A[m,l]/mv::A[m,l]'][_/']
\end{aligned}$$

□

OP.12 Obrigação de Prova OP.12

$ro = \sum_{j=0}^{m-1} r[j]^2 \Rightarrow (ro' = \sum_{j=0}^{m} r[j]^2)[rho + r[m]^2/rho'][\_/']$

Partindo da premissa, tem-se que:

$rho = \sum_{j=0}^{m-1} r[j]^2 \Rightarrow (rho = \sum_{j=0}^{m-1} r[j]^2) \wedge (rho + r[m]^2 = rho + r[m]^2)$
Dado que rho' não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((rho = \sum_{j=0}^{m-1} r[j]^2) \wedge (rho' = rho + r[m]^2))[rho + r[m]^2/rho']$
$\Rightarrow (rho' = \sum_{j=0}^{m} r[j]^2)[rho + r[m]^2/rho']$
$\Rightarrow (rho' = \sum_{j=0}^{m} r[j]^2)[rho + r[m]^2/rho'][\_/']$
□

OP.13 Obrigação de Prova OP.13

$\forall j : 0..m-1 \bullet p[j] = r[j] \Rightarrow (\forall j : 0..m \bullet p[j]' = r[j])[r[m]/p[m]'][\_/']$

Partindo da premissa, tem-se que:

$\forall j : 0..m-1 \bullet p[j] = r[j] \Rightarrow (\forall j : 0..m-1 \bullet p[j] = r[j]) \wedge (r[m] = r[m])$
Dado que $p[m]'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((\forall j : 0..m-1 \bullet p[j] = r[j]) \wedge (p[m]' = r[m]))[r[m]/p[m]']$
$\Rightarrow ((\forall j : 0..m-1 \bullet p[j]' = r[j]) \wedge (p[m]' = r[m]))[r[m]/p[m]'][\_/']$
$\Rightarrow (\forall j : 0..m \bullet p[j]' = r[j])[r[m]/p[m]'][\_/']$
□

OP.14 Obrigação de Prova OP.14

$true \Rightarrow (i = N-1) \vee (i \neq N-1)$

Para provar a implicação, basta provar $(i = N-1) \vee (i \neq N-1)$. A proposição $(i = N-1) \vee (i \neq N-1)$ é verdadeira pela lei do terceiro excluído.

OP.15 Obrigação de Prova OP.15

$(i = N-1) \wedge (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge$
$\forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim-1) \bullet a::v[k]' = p[k - lines \times i])$
$\Rightarrow (i \neq N-1 \wedge a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge$
$\quad \forall k : (lines \times i)..(lines \times (i+1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \vee$
$\quad (i = N-1 \wedge a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge$
$\quad \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim-1) \bullet a::v[k]' = p[k - lines \times i])$

Partindo da premissa, tem-se que:

$$
\begin{array}{l}
(i = N - 1) \wedge (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\
\forall\, k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i]) \\
\Rightarrow ((i = N - 1) \wedge (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\
\quad \forall\, k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i])) \vee \\
\quad (i \neq N - 1 \wedge a::beg' = lines \times i \wedge a::end' = lines \times (i + 1) - 1 \wedge \\
\quad \forall\, k : (lines \times i)..(lines \times (i + 1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \\
\Box
\end{array}
$$

OP.16 Obrigação de Prova OP.16

$$
\begin{array}{l}
(i \neq N - 1) \wedge (a::beg' = lines \times i \wedge a::end' = lines \times (i + 1) - 1 \wedge \\
\forall\, k : (lines \times i)..(lines \times (i + 1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \\
\Rightarrow (i \neq N - 1 \wedge a::beg' = lines \times i \wedge a::end' = lines \times (i + 1) - 1 \wedge \\
\quad \forall\, k : (lines \times i)..(lines \times (i + 1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \vee \\
\quad (i = N - 1 \wedge a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\
\quad \forall\, k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i]) \\
\Box
\end{array}
$$

Partindo da premissa, tem-se que:

$$
\begin{array}{l}
(i \neq N - 1) \wedge (a::beg' = lines \times i \wedge a::end' = lines \times (i + 1) - 1 \wedge \\
\forall\, k : (lines \times i)..(lines \times (i + 1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \\
\Rightarrow (i \neq N - 1 \wedge a::beg' = lines \times i \wedge a::end' = lines \times (i + 1) - 1 \wedge \\
\quad \forall\, k : (lines \times i)..(lines \times (i + 1) - 1) \bullet a::v[k]' = p[k - lines \times i]) \vee \\
\quad (i = N - 1 \wedge a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge \\
\quad \forall\, k : (i \times \lfloor \frac{dim}{N} \rfloor)..(dim - 1) \bullet a::v[k]' = p[k - lines \times i]) \\
\Box
\end{array}
$$

OP.17 Obrigação de Prova OP.17

$$
\begin{array}{l}
a::end = dim - 1 \\
\Rightarrow (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1)[i \times \lfloor \frac{dim}{N} \rfloor / a::beg'][\_/']
\end{array}
$$

Partindo da premissa, tem-se que:

$$
\begin{array}{l}
a::end = dim - 1 \\
\Rightarrow (i \times \lfloor \frac{dim}{N} \rfloor = i \times \lfloor \frac{dim}{N} \rfloor) \wedge (a::end = dim - 1)
\end{array}
$$

Dado que $a::beg'$ não aparece em nenhuma das proposições da fórmula

$\Rightarrow ((a::beg' = i \times \lfloor \frac{dim}{N} \rfloor) \wedge (a::end = dim - 1))[i \times \lfloor \frac{dim}{N} \rfloor / a::beg']$
$\Rightarrow ((a::beg' = i \times \lfloor \frac{dim}{N} \rfloor) \wedge (a::end' = dim - 1))[i \times \lfloor \frac{dim}{N} \rfloor / a::beg'][\_/']$
$\square$

OP.18 Obrigação de Prova OP.18

$a::beg = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end = dim - 1 \wedge$
$\forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m-1) \bullet a::v[k] = p[k - lines \times i]$
$\Rightarrow (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge$
$\quad \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m) \bullet$
$\qquad a::v[k]' = p[k - lines \times i])[p[m - lines \times i]/a::v[m]'][\_/']$

Partindo da premissa, tem-se que:

$a::beg = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end = dim - 1 \wedge$
$\forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m-1) \bullet a::v[k] = p[k - lines \times i]$
$\Rightarrow a::beg = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end = dim - 1 \wedge$
$\quad \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m-1) \bullet a::v[k] = p[k - lines \times i] \wedge$
$\quad p[m - lines \times i] = p[m - lines \times i]$
Dado que $a::v[m]'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow (a::beg = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end = dim - 1 \wedge$
$\quad \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m-1) \bullet a::v[k] = p[k - lines \times i] \wedge$
$\quad a::v[m]' = p[m - lines \times i])[p[m - lines \times i]/a::v[m]']$
$\Rightarrow (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge$
$\quad \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m-1) \bullet a::v[k]' = p[k - lines \times i] \wedge$
$\quad a::v[m]' = p[m - lines \times i])[p[m - lines \times i]/a::v[m]'][\_/']$
$\Rightarrow (a::beg' = i \times \lfloor \frac{dim}{N} \rfloor \wedge a::end' = dim - 1 \wedge$
$\quad \forall k : (i \times \lfloor \frac{dim}{N} \rfloor)..(m) \bullet$
$\qquad a::v[k]' = p[k - lines \times i])[p[m - lines \times i]/a::v[m]'][\_/']$
$\square$

OP.19 Obrigação de Prova OP.19

$a::end = lines \times (i+1) - 1$
$\Rightarrow (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1)[lines \times i / a::beg'][\_/']$

Partindo da premissa, tem-se que:

$a::end = lines \times (i+1) - 1$
$\Rightarrow (a::end = lines \times (i+1) - 1) \wedge (lines \times i = lines \times i)$

Dado que $a::beg'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((a::end = lines \times (i+1) - 1) \wedge (a::beg' = lines \times i))[lines \times i / a::beg']$
$\Rightarrow (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1)[lines \times i / a::beg'][\_/']$
$\square$

OP.20 Obrigação de Prova OP.20

$a::beg = lines \times i \wedge a::end = lines \times (i+1) - 1 \wedge$
$\forall\, k : (lines \times i)..(m-1) \bullet a::v[k] = p[k - lines \times i]$
$\Rightarrow (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge$
$\quad \forall\, k : (lines \times i)..m \bullet a::v[k]' = p[k - lines \times i])[p[m - lines \times i] / a::v[m]'][\_/']$

Partindo da premissa, tem-se que:

$a::beg = lines \times i \wedge a::end = lines \times (i+1) - 1 \wedge$
$\forall\, k : (lines \times i)..(m-1) \bullet a::v[k] = p[k - lines \times i]$
$\Rightarrow a::beg = lines \times i \wedge a::end = lines \times (i+1) - 1 \wedge$
$\quad \forall\, k : (lines \times i)..(m-1) \bullet a::v[k] = p[k - lines \times i] \wedge$
$\quad p[m - lines \times i] = p[m - lines \times i]$
Dado que $a::v[m]'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow (a::beg = lines \times i \wedge a::end = lines \times (i+1) - 1 \wedge$
$\quad \forall\, k : (lines \times i)..(m-1) \bullet a::v[k] = p[k - lines \times i] \wedge$
$\quad a::v[m]' = p[m - lines \times i])[p[m - lines \times i] / a::v[m]']$
$\Rightarrow (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge$
$\quad \forall\, k : (lines \times i)..(m-1) \bullet a::v[k]' = p[k - lines \times i] \wedge$
$\quad a::v[m]' = p[m - lines \times i])[p[m - lines \times i] / a::v[m]'][\_/']$
$\Rightarrow (a::beg' = lines \times i \wedge a::end' = lines \times (i+1) - 1 \wedge$
$\quad \forall\, k : (lines \times i)..m \bullet a::v[k]' = p[k - lines \times i])[p[m - lines \times i] / a::v[m]'][\_/']$
$\square$

OP.21 Obrigação de Prova OP.21

$\forall\, k : 0..m-1 \bullet mv::v[k] = a::v[k]$
$\Rightarrow (\forall\, k : 0..m \bullet mv::v[k]' = a::v[k])[a::v[m] / mv::v[m]'][\_/']$

Partindo da premissa, tem-se que:

$\forall\, k : 0..m-1 \bullet mv::v[k] = a::v[k]$
$\Rightarrow (\forall\, k : 0..m-1 \bullet mv::v[k] = a::v[k]) \wedge (a::v[m] = a::v[m])$
Dado que $mv::v[m]'$ não aparece em nenhuma das proposições da fórmula

$\Rightarrow ((\forall\, k : 0..m-1 \bullet mv::v[k] = a::v[k]) \wedge$
$(mv::v[m]' = a::v[m]))[a::v[m]/mv::v[m]']$
$\Rightarrow ((\forall\, k : 0..m-1 \bullet mv::v[k]' = a::v[k]) \wedge$
$(mv::v[m]' = a::v[m]))[a::v[m]/mv::v[m]'][\_/']$
$\Rightarrow (\forall\, k : 0..m \bullet mv::v[k]' = a::v[k])[a::v[m]/mv::v[m]'][\_/']$
$\square$

OP.22 Obrigação de Prova OP.22

$\forall\, k : 0..m-1 \bullet q[k] = mv::x[k]$
$\Rightarrow (\forall\, k : 0..m \bullet q[k]' = mv::x[k])[mv::x[m]/q[m]'][\_/']$

Partindo da premissa, tem-se que:

$\forall\, k : 0..m-1 \bullet q[k] = mv::x[k]$
$\Rightarrow (\forall\, k : 0..m-1 \bullet q[k] = mv::x[k]) \wedge (mv::x[m] = mv::x[m])$
Dado que $q[m]'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((\forall\, k : 0..m-1 \bullet q[k] = mv::x[k]) \wedge (q[m]' = mv::x[m]))[mv::x[m]/q[m]']$
$\Rightarrow ((\forall\, k : 0..m-1 \bullet q[k]' = mv::x[k]) \wedge$
$(q[m]' = mv::x[m]))[mv::x[m]/q[m]'][\_/']$
$\Rightarrow (\forall\, k : 0..m \bullet q[k]' = mv::x[k])[mv::x[m]/q[m]'][\_/']$
$\square$

OP.23 Obrigação de Prova OP.23

$alfa = \sum_{k=0}^{m-1} p[k] \times q[k]$
$\Rightarrow (alfa' = \sum_{k=0}^{m} p[k] \times q[k])[alfa + p[m] \times q[m]/alfa'][\_/']$

Partindo da premissa, tem-se que:

$alfa = \sum_{k=0}^{m-1} p[k] \times q[k]$
$\Rightarrow (alfa = \sum_{k=0}^{m-1} p[k] \times q[k]) \wedge (alfa + p[m] \times q[m] = alfa + p[m] \times q[m])$
Dado que $alfa'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((alfa = \sum_{k=0}^{m-1} p[k] \times q[k]) \wedge$
$(alfa' = alfa + p[m] \times q[m]))[alfa + p[m] \times q[m]/alfa']$
$\Rightarrow ((alfa' = \sum_{k=0}^{m-1} p[k] \times q[k]) \wedge$
$(alfa' = alfa + p[m] \times q[m]))[alfa + p[m] \times q[m]/alfa'][\_/']$
$\Rightarrow (alfa' = \sum_{k=0}^{m} p[k] \times q[k])[alfa + p[m] \times q[m]/alfa'][\_/']$
$\square$

OP.24 Obrigação de Prova OP.24

$$alfa = \sum_{k=0}^{m} p[k] \times q[k] \Rightarrow (alfa' = \frac{\sum_{k=0}^{m} p[k] \times q[k]}{rho})[alfa/rho/alfa'][\_/']$$

Partindo da premissa, tem-se que:

$alfa = \sum_{k=0}^{m} p[k] \times q[k] \Rightarrow (alfa = \sum_{k=0}^{m} p[k] \times q[k]) \wedge (alfa/rho = alfa/rho)$
Dado que $alfa'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((alfa = \sum_{k=0}^{m} p[k] \times q[k]) \wedge (alfa' = alfa/rho))[alfa/rho/alfa']$
$\Rightarrow (alfa' = \frac{\sum_{k=0}^{m} p[k] \times q[k]}{rho})[alfa/rho/alfa']$
$\Rightarrow (alfa' = \frac{\sum_{k=0}^{m} p[k] \times q[k]}{rho})[alfa/rho/alfa'][\_/']$
□

OP.25 Obrigação de Prova OP.25

$\forall\, k : 0..m-1 \bullet z[k] = z[k] + alfa \times p[k]$
$\Rightarrow (\forall\, k : 0..m \bullet z[k]' = z[k] + alfa \times p[k])[z[m] + alfa \times p[m]/z[m]'][\_/']$

Partindo da premissa, tem-se que:

$\forall\, k : 0..m-1 \bullet z[k] = z[k] + alfa \times p[k]$
$\Rightarrow (\forall\, k : 0..m-1 \bullet z[k] = z[k] + alfa \times p[k]) \wedge$
$(z[m] + alfa \times p[m] = z[m] + alfa \times p[m])$
Dado que $z[m]'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((\forall\, k : 0..m-1 \bullet z[k] = z[k] + alfa \times p[k]) \wedge$
$(z[m]' = z[m] + alfa \times p[m]))[z[m] + alfa \times p[m]/z[m]']$
$\Rightarrow ((\forall\, k : 0..m-1 \bullet z[k]' = z[k] + alfa \times p[k]) \wedge$
$(z[m]' = z[m] + alfa \times p[m]))[z[m] + alfa \times p[m]/z[m]'][\_/']$
$\Rightarrow (\forall\, k : 0..m \bullet z[k]' = z[k] + alfa \times p[k])[z[m] + alfa \times p[m]/z[m]'][\_/']$
□

OP.26 Obrigação de Prova OP.26

$\forall\, k : 0..m-1 \bullet r[k] = r[k] - alfa \times q[k]$
$\Rightarrow (\forall\, k : 0..m \bullet r[k]' = r[k] - alfa \times q[k])[r[m] - alfa \times q[m]/r[m]'][\_/']$

Partindo da premissa, tem-se que:

$\forall\, k : 0..m-1 \bullet r[k] = r[k] - alfa \times q[k]$
$\Rightarrow (\forall\, k : 0..m-1 \bullet r[k] = r[k] - alfa \times q[k]) \wedge (r[m] - alfa \times q[m] =$

$r[m] - alfa \times q[m])$

Dado que $r[m]'$ não aparece em nenhuma das proposições da fórmula

$\Rightarrow ((\forall\, k : 0..m-1 \bullet r[k] = r[k] - alfa \times q[k]) \wedge$
$(r[m]' = r[m] - alfa \times q[m]))[r[m] - alfa \times q[m]/r[m]']$

$\Rightarrow ((\forall\, k : 0..m-1 \bullet r[k]' = r[k] - alfa \times q[k]) \wedge$
$(r[m]' = r[m] - alfa \times q[m]))[r[m] - alfa \times q[m]/r[m]'][\_/']$

$\Rightarrow (\forall\, k : 0..m \bullet r[k]' = r[k] - alfa \times q[k])[r[m] - alfa \times q[m]/r[m]'][\_/']$

□

OP.27 Obrigação de Prova OP.27

$rho = \sum_{k=0}^{m-1} r[k]^2 \Rightarrow (rho' = \sum_{k=0}^{m} r[k]^2)[rho + r[m]^2/rho'][\_/']$

Partindo da premissa, tem-se que:

$rho = \sum_{k=0}^{m-1} r[k]^2 \Rightarrow (rho = \sum_{k=0}^{m-1} r[k]^2) \wedge (rho + r[m]^2 = rho + r[m]^2)$

Dado que rho' não aparece em nenhuma das proposições da fórmula

$\Rightarrow ((rho = \sum_{k=0}^{m-1} r[k]^2) \wedge (rho' = rho + r[m]^2))[rho + r[m]^2/rho']$

$\Rightarrow (rho' = \sum_{k=0}^{m} r[k]^2)[rho + r[m]^2/rho']$

$\Rightarrow (rho' = \sum_{k=0}^{m} r[k]^2)[rho + r[m]^2/rho'][\_/']$

□

OP.28 Obrigação de Prova OP.28

$\forall\, k : 0..m-1 \bullet p[k] = r[k] + beta \times p[k]$

$\Rightarrow (\forall\, k : 0..m \bullet p[k]' = r[k] + beta \times p[k])[r[m] + beta \times p[m]/p[m]'][\_/']$

Partindo da premissa, tem-se que:

$\forall\, k : 0..m-1 \bullet p[k] = r[k] + beta \times p[k]$

$\Rightarrow (\forall\, k : 0..m-1 \bullet p[k] = r[k] + beta \times p[k]) \wedge$
$(r[m] + beta \times p[m] = r[m] + beta \times p[m])$

Dado que $p[m]'$ não aparece em nenhuma das proposições da fórmula

$\Rightarrow ((\forall\, k : 0..m-1 \bullet p[k] = r[k] + beta \times p[k]) \wedge$
$(p[m]' = r[m] + beta \times p[m]))[r[m] + beta \times p[m]/p[m]']$

$\Rightarrow ((\forall\, k : 0..m-1 \bullet p[k]' = r[k] + beta \times p[k]) \wedge$
$(p[m]' = r[m] + beta \times p[m]))[r[m] + beta \times p[m]/p[m]'][\_/']$

$\Rightarrow (\forall\, k : 0..m \bullet p[k]' = r[k] + beta \times p[k])[r[m] + beta \times p[m]/p[m]'][\_/']$

□

OP.29 Obrigação de Prova OP.29

$residue = \sum_{k=0}^{m-1} r[k]^2$
$\Rightarrow (residue' = \sum_{k=0}^{m} r[k]^2)[residue + r[m]^2/residue'][\_/']$

Partindo da premissa, tem-se que:

$residue = \sum_{k=0}^{m-1} r[k]^2$
$\Rightarrow (residue = \sum_{k=0}^{m-1} r[k]^2) \wedge (residue + r[m]^2 = residue + r[m]^2)$
Dado que $residue'$ não aparece em nenhuma das proposições da fórmula
$\Rightarrow ((residue = \sum_{k=0}^{m-1} r[k]^2) \wedge$
$(residue' = residue + r[m]^2))[residue + r[m]^2/residue']$
$\Rightarrow (residue' = \sum_{k=0}^{m} r[k]^2)[residue + r[m]^2/residue']$
$\Rightarrow (residue' = \sum_{k=0}^{m} r[k]^2)[residue + r[m]^2/residue'][\_/']$
□