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

Coordenação e Reconfiguração Dinâmica de Componentes em uma Plataforma de Computação Paralela

Juliano Efson Norberto Sales

Fortaleza – Ceará
Novembro 2012

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

Coordenação e Reconfiguração Dinâmica de Componentes em uma Plataforma de Computação Paralela

Autor
Juliano Efson Norberto Sales

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
Novembro 2012

Não haveria mestrado, se antes não houvesse alfabetização. Não haveria mestrado, se antes não houvesse formação cidadã...

Titia Terezinha exerceu papel fundamental para que estes dois requisitos pudessem ser formados e maturados em mim. Não bastassem tais motivos, por todo cuidado, toda atenção, todo carinho e todo amor que me destes ao longo da vida, dedico este trabalho a ti, minha tia.

Agradecimentos

Agradeço ao professor Heron, pessoa sem a qual esta jornada certamente seria mais difícil. Um pesquisador nato, que com uma impressionante dedicação ao trabalho, conseguiu transmitir a mim ensinamentos científicos e também morais.

Agradeço aos meus amigos e familiares que compreenderam as minhas ausências e me fortaleceram nos clássicos momentos de angústias a que fui acometido ao longo de curso. Sou grato também pelos momentos de fuga e diversão que puderam me proporcionar com suas excelentes companhias.

Agradeço aos meus amigos do MDCC pelas horas de estudos e pelo fortalecimento e encorajamento mútuo. Até os vossos exemplos de vida foram motivadores para mim.

Agradeço a Deus por ter-me concedido a chance de vivenciar esta experiência única de aprendizagem.

Por fim, agradeço ao Banco do Nordeste por me conceder um período de dedicação integral a esta dissertação. Tais semanas foram de fundamental importância para a maturação deste trabalho.

Resumo

Nos domínios da Computação de Alto Desempenho (CAD), são comuns aplicações com tempo de execução de longa duração. Durante a execução de uma aplicação dessa natureza, podem ser identificadas melhorias ou correções nos algoritmos em execução que não invalidam o processamento previamente realizado. Neste cenário, a capacidade de realizar modificações em tempo de execução se mostra de grande utilidade. A esta técnica chamamos *reconfiguração dinâmica*, a qual, dentre outros meios discutidos ao longo do trabalho, pode ser implementada a partir do uso de linguagens de propósito específico como as linguagens de descrição de arquitetura (ADL).

Uma ADL permite a especificação de um sistema de *software* a partir da construção de *conectores exógenos* com a função de combinar e definir os formatos de dados e protocolos nas interações de *componentes*.

Este trabalho de pesquisa tem como objeto o projeto de uma ADL e a implementação de um ambiente de interpretação de configuração para a plataforma de componentes paralelos HPE (*Hash Programming Environment*). Essa linguagem tem como principal propósito oferecer a capacidade de especificar conectores exógenos e suporte à reconfiguração dinâmica.

Estudos de caso avaliam o desempenho da interpretação dos componentes desenvolvidos pela ADL, como também validam as operações de reconfiguração dinâmica. Os resultados mostram sobrecarga considerada aceitável no processo de interpretação, para instâncias realísticas de problemas, de forma que, quando utilizado com prudência, os conectores podem ser utilizados até mesmo em cenários de produção. Em alguns casos, o peso da interpretação chega a ser desprezível. Os ensaios de reconfiguração também se mostram satisfatórios para os requisitos apresentados, sendo um dos principais diferencias da solução, a simplicidade de uso do mecanismo.

Abstract

Long running applications are very common in High Performance Computing (HPC) domains. During the execution of this kind of application, some improvements or corrections can be identified and applied without making invalid the data that has been processed. In these cases, the ability to make changes in a parallel program during execution is considered useful. *Dynamic reconfiguration* is the term used to describe this technique, which can be implemented using different alternatives, like architecture description languages (ADL).

An ADL allows a the specification of a software based on *exogenous conectors* to combine and define data types and protocols for orchestrating the interaction between components.

This research has the goal of designing an ADL and implementing a configuration interpretation environment for the HPE component-based parallel computing platform. The main purpose of this language is to provide the ability to specify exogenous connectors and support dynamic reconfiguration.

Case studies evaluate the performance of the component interpretation developed by the ADL, as well as validate the actions of dynamic reconfiguration. The results are an evidence that the overhead in the interpretation process for realistic problem instances is acceptable, in such a way that, when used wisely, the connectors can be used even in production scenarios. In some cases, the interpretation weight can be disregarded. The reconfiguration experiments are also deemed satisfactory, making the simplicity of the mechanism the major draw of the solution.

Sumário

Capítulo 1

Introdução

O trabalho de pesquisa cujos resultados são apresentados neste documento tem como objeto o projeto de uma linguagem de descrição de arquitetura e a implementação de um ambiente de interpretação de configuração para a plataforma de componentes paralelos HPE (*Hash Programming Environment*) [20]. A plataforma HPE é aderente à especificação CCA [7], além de implementar o modelo de componentes HASH [17]. Essa linguagem tem como principal propósito oferecer a capacidade de especificar conectores exógenos como componentes do modelo HASH e oferecer suporte à reconfiguração dinâmica desses componentes durante a sua execução em plataformas de computação paralela.

1.1 Motivações

À medida que cresce a complexidade das aplicações dentro de um determinado domínio de interesse, a descrição arquitetural do sistema torna-se mais significativa frente a descrição dos algoritmos e estruturas de dados usados em sua implementação. Desta forma, move-se de uma perspectiva de desenvolvimento em pequena escala (*programming-in-the-small*) para uma perspectiva de desenvolvimento de grande escala (*programming-in-the-large*). Sob essa perspectiva, um *software* passa a ser visto como uma composição de elementos arquiteturais de maior granularidade e suas interações, somados às restrições que governam tais interações e aos padrões com que esses elementos podem ser combinados entre si [51, 63].

Uma *linguagem de descrição de arquitetura* (ADL) permite a especificação de um sistema de *software* a partir de partes que isolam interesses, ou conjuntos de interesses inter-relacionados. Essas partes são chamadas de *componentes*, os

quais podem ser ainda reutilizáveis entre *softwares* distintos e possuir ciclos de vida independentes entre si e em relação aos *softwares* nos quais são utilizados. Para que componentes possam colaborar, utilizam-se conectores, os quais tem o papel de definir quais são os formatos de dados e protocolos permitidos nas interações, sendo estes os que de fato governam o sistema.

Diz-se que um conector é *endógeno* quando atua de forma passiva, ou seja, o padrão de interação entre os componentes está definida na lógica de implementação dos seus próprios propósitos [39]. Além dos seus próprios interesses, os componentes absorvem os fluxos de execução. Esse tipo de conector transforma-se em um meio de acesso à outros componentes, sendo inclusive capaz de absorver parte das regras de comunicação, porém incapazes de encapsular todo o controle da interação entre os componentes. Por outro lado, quando esse controle é realizado pelo próprio conector, eliminando a necessidade de dispersar as lógicas de interação entre os que realizam a computação, este é chamado de *exógeno* [39]. O conector exógeno é, portanto, responsável por reger, ativa e espontaneamente, a interação entre os componentes, ou seja, a comunicação, coordenação, conversão, facilitação ou suas combinações [18].

Algumas ADL restringem-se a descrever um sistema estruturalmente, apenas declarando o conjunto de componentes que compõem o sistema e seus conectores. Essas linguagens não oferecem suporte para a especificação do comportamento do sistema através da orquestração explícita de seus componentes e conectores, sob uma perspectiva de programação em grande escala. O comportamento do sistema encontra-se descrito internamente aos componentes e conectores, utilizando linguagens de programação usuais, sob uma perspectiva de programação em pequena escala.

Um outro grupo de ADL permite a especificação do comportamento de sistemas, tornando possível orquestrar a execução dos componentes e conectores. Linguagens pertencentes a esse grupo estão capacitadas a descrever sistemas, bem como outros componentes, por meio de conectores exógenos, sendo esse o motivo pelo qual esse trabalho de pesquisa interessa-se primordialmente por esse tipo de ADL.

A descrição de conectores exógenos a partir de linguagens de descrição de arquitetura é portanto fundamental para os objetivos desse trabalho de pesquisa, devido às suas propriedades benéficas à descrição de sistemas, destacadas a seguir:

- conectores exógenos permitem um maior grau de independência funcional entre os componentes ao restringir as dependências do sistema para o sentido conector-componente, ao invés de componente-componente, e, portanto,

oferecem um maior suporte a reusabilidade, tanto de componentes quanto de conectores; e

- sistemas de *software* definidos por conectores exógenos permitem a criação de meios mais simples de intervenção na execução, através da reconfiguração dinâmica de sua estrutura e do seu comportamento.

1.1.1 Importância do Tema na Computação de Alto Desempenho (CAD)

Nos domínios da Computação de Alto Desempenho (CAD), são comuns aplicações com tempo de execução de longa duração, em magnitudes variadas, medidas em horas, dias ou até meses. Durante a execução de uma aplicação dessa natureza, podem ser identificadas melhorias ou correções nos algoritmos em execução que não invalidam os processamentos previamente realizados. Além disso, novos algoritmos podem ser inseridos com diferentes propósitos, como acelerar a convergência de um método numérico para obter um resultado de forma mais rápida ou mesmo aumentar a precisão e confiabilidade da solução. A reinicialização da computação de um sistema dessa natureza a fim de introduzir alterações, independentemente do motivo, implica em desperdiçar a computação já realizada e, potencialmente, desperdiçar também recursos financeiros [31]. Assim, a menos que os resultados parciais obtidos até então estejam incorretos, é prudente fazer uso de métodos que efetuem os devidos ajustes de forma a maximizar o tempo e minimizar o custo do alcance da informação final.

A *reconfiguração dinâmica* é a capacidade de realizar modificações em um sistema que encontra-se em execução. Consideremos um exemplo concreto para ilustrá-la.

Em problemas numéricos, onde é comum lidar com a solução de sistemas lineares, existem algoritmos otimizados de acordo com o tipo da matriz, a qual pode ser densa ou esparsa. Matrizes esparsas podem ainda ser classificadas de acordo com o padrão de ocorrência dos elementos não-nulos, podendo ser triangulares, diagonais, tridiagonais, pentadiagonais, bloco-tridiagonais, etc. A utilização de um algoritmo genérico, capaz de calcular com precisão a solução independentemente do tipo da matriz, certamente seria eficaz para atender aos requisitos funcionais do problema em questão, mas provavelmente não seria eficiente, frente aos requisitos não-funcionais relacionados ao tempo necessário para chegar à solução. Implementações especializadas para um tipo específico de matriz podem obter ganhos significativos em termos de tempo de execução, necessidade de espaço de memória durante o processamento e precisão dos resultados. Nesse caso,

interessam alternativas que permitam identificar as características da instância do problema dinamicamente e reconfigurar o sistema durante a sua execução [31], onde a reconfiguração dinâmica torna-se uma importante ferramenta [35].

Normalmente, aplicações com forte relação com matrizes ou outras estruturas de dados de armazenamento de grande quantidade de informação estão relacionadas à problemas que envolvem simulações, pesquisas ou otimizações, procedimentos estes que usualmente estão associadas a grande esforço computacional. Como estes, os exemplos aqui expostos que ajudaram a compreender parte dos conceitos a respeito da reconfiguração dinâmica, não por acaso remetem à aplicações de computação intensiva, campo nato do estudo da área de computação de alto desempenho.

Conforme veremos ao longo desta dissertação, os principais modelos de componentes especificamente desenvolvidos para aplicações de alto desempenho, já oferecem suporte a mudanças dinâmicas. Outros estudos, por sua vez envolvendo aplicações paralelas tradicionais, sem o uso de modelos de componentes, também vem, em anos recentes, investigando formas de incluir a reconfiguração dinâmica nas suas capacidades [35] [32].

Ao longo dos últimos anos, a plataforma de componentes HPE (*Hash Programming Enviroment*) tem sido prototipada por membros do grupo de pesquisa ParGO (*Paralelismo, Grafos e Otimização*) do MDCC/UFC, para implantação e execução de componentes paralelos que seguem o modelo HASH. Esse modelo é designado para arquiteturas de *cluster computing*, a qual é largamente utilizada e corresponde a 82% das entradas do *ranking* Top 500[1] de novembro de 2012, que classifica os 500 computadores paralelos de maior desempenho da atualidade.

Embora importante, pelos argumentos expostos, a reconfiguração dinâmica de componentes paralelos não tem sido tratada na implementação do HPE. Esta dissertação tem o propósito de preencher essa lacuna, oferecendo o projeto e prototipação de uma linguagem de descrição arquitetural que permita a configuração e reconfiguração dinâmica de conectores exógenos na forma de componentes regulares da plataforma. Essa nova linguagem é uma extensão da linguagem HCL (*Hash Configuration Language*), a qual atualmente se restringe aos aspectos estruturais da descrição arquitetural, ignorando porém os aspectos comportamentais dos componentes paralelos.

[1] http://www.top500.org/statistics/list/

1.1.2 Contribuições

Pesquisadores e cientistas das áreas de ciências naturais e engenharias correspondem a um significativo grupo de usuários com interesse em aplicações de CAD. Naturalmente, cada um deles possui habilidades, conhecimento e experiência profissional em suas áreas de formação, as quais normalmente são divergentes das ciências computacionais. Apesar do aumento de oferta de disciplinas associadas a programação nos cursos universitários dessas áreas, o nível de formação ainda torna-se incipiente para a plena utilização de técnicas e tecnologias mais apuradas.

Nesses termos, este trabalho desenvolve um protótipo de ambiente que seria disponibilizado com suporte a programação em larga escala, visto como um tipo de programação que favorece os usuários sem massivos conhecimentos em programação de alto desempenho no trato das aplicações na forma de grandes elementos arquiteturais. Aplicações construídas com essa forma de composição podem ser mais facilmente manipuladas, ao compará-las com as aplicações desenvolvidas segundo os princípios da programação em pequena escala.

Assim, acreditamos que contribuímos para aproximar as tecnologias de computação de alto desempenho, dos usuários alheios, a sólidos conhecimentos nos domínios das ciências computacionais e engenharias, uma vez que estamos convictos que o uso de um ambiente que suporte a programação e execução de aplicações, na forma de conectores exógenos desenvolvidas através de linguagens de descrição de arquitetura, traz significativa redução de sua complexidade.

Aliado a esse benefício, a possibilidade de realizar reconfigurações dinâmicas facilitam a experimentação e prototipação de aplicações de CAD. A simplicidade da ferramenta disponibilizada, juntamente com a definição de processos de uso intuitivos, contribuem, mais uma vez, para alcançarmos usuários com formação diferentes da nossa, cientistas da computação.

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho de pesquisa tem como objetivo a proposta e implementação de uma linguagem de descrição de arquitetura que forneça suporte à configuração e à reconfiguração dinâmica de conectores exógenos para orquestração de componentes paralelos sobre a plataforma HPE.

1.2.2 Objetivos Específicos

- Avaliar qualitativamente os mecanismos que permitem a inclusão de reconfiguração dinâmica em modelos de componentes em geral;
- Identificar e descrever os principais modos de reconfiguração estática e dinâmica de componentes paralelos na plataforma HPE;
- Disponibilizar uma infraestrutura de execução de aplicações baseadas em conectores exógenos, desenvolvidas com a ADL proposta.

1.3 Organização da Proposta

Seguido a este capítulo introdutório, esta dissertação possui outros cinco capítulos, os quais descrevemos a seguir.

No Capítulo 2, apresentamos uma visão geral sobre a avaliação de mudanças de *software*, seguido por temas focados em orientação a componentes. Os conceitos de componentes, conectores e linguagens de descrição de arquitetura são motivados e explicados em detalhe. O capítulo finaliza avaliando diversas formas de implementação de reconfiguração dinâmica e outras considerações importantes a seu respeito.

No Capítulo 3, são apresentados modelos de componentes para a computação de alto desempenho. Em seguida são avaliadas suas ADL e os mecanismos de reconfiguração dinâmica presentes nesses modelos e nas plataformas que os implementam.

O Capítulo 4 apresenta o trabalho realizado, descrevendo as extensões da HCL para a especificação de conectores exógenos, as estratégias e operações de reconfiguração que são suportadas pelo HPE e uma explanação do protótipo de interpretação e reconfiguração desenvolvido para suportar a ADL proposta.

O Capítulo 5 apresenta resultados experimentais, a fim de avaliar a solução proposta. O capítulo inicia com a apresentação das aplicações do *NAS Parallel Benchmarks (NPB)* que foram utilizadas durante a fase de testes sobre o HPE. Em seguida, são detalhados os experimentos realizados, juntamente com a metodologia. O capítulo se encerra com a apresentação dos resultados e discussões a seu respeito.

No Capítulo 6, são expostas as conclusões sobre o trabalho desenvolvido, confrontando os resultados alcançados e discutidos com os objetivos propostos, relatando suas contribuições, e delineando sugestões de trabalhos futuros com base na análise crítica das limitações do trabalho desenvolvido.

Capítulo 2

Contexto e Referencial Teórico

A necessidade de automatizar processos mais complexos, aliada com a disponibilidade de maior capacidade computacional, em termos de armazenamento e processamento de dados, motiva o desenvolvimento de *softwares* cada vez mais sofisticados. Os atuais problemas computacionais das pessoas e corporações não podem mais ser resolvidos apenas com pequenos utilitários. Atualmente, são necessárias milhares ou até milhões de linhas de código para se desenvolver *software* de larga escala. [68].

Gerir todo esse volume de artefatos de forma a alcançar a almejada produtividade no desenvolvimento de *software*, bem como a sua qualidade, não é uma uma tarefa trivial. Para alcançar esses objetivos com o menor esforço possível, é essencial conhecer as formas de mudanças a que um *software* está sujeito e utilizar técnicas adequadas para cada tipo de mudança.

Este capítulo apresenta a evolução do *software* no contexto das arquiteturas baseadas em componentes e com foco nos conectores exógenos, fazendo uso de linguagens de descrição de arquitetura para definir as relações entre conectores e componentes, bem como guiar sua evolução dinâmica. Inicialmente, o texto trata de uma taxonomia para mudanças de *software*, em um contexto geral. Em seguida, são motivados e introduzidos os conceitos de componentes e de conectores, bem como suas diversas formas de relacionamento. Por fim, a descrição arquitetural de sistemas e sua relação com as linguagens de descrição de arquitetura são discutidas, mostrando como a modificação dinâmica de *software* pode ser abordada nesse contexto.

2.1 Evolução de *Software*

Softwares estão em constante evolução em consonância com as mudanças do mundo real e é natural imaginar que *softwares* maiores potencialmente necessitem de mais esforço para realização de manutenções, tanto no desenvolvimento de novas funcionalidade, como na alteração de outras já existentes.

No processo de evolução de sistemas, mudanças são fatos certos de ocorrer [59]. Considerando a dinâmica dos domínios dentro dos quais os aplicativos são desenvolvidos, a evolução deve ser vista como uma preocupação imperativa no ciclo de vida do *software*.

Software[1] *é (1) um conjunto de instruções (programas de computador) que quando executados proveem a funcionalidade e execução desejadas, (2) estruturas de dados que permitem que programas manipulem adequadamente informações, e (3) documentos que descrevem as operações e uso dos programas* [59].

Por essa definição, assumimos que os *softwares* são bem mais do que os simples arquivos finais executáveis. Um *software* é um conjunto de diversos *artefatos* inter-relacionados. Um artefato é uma especificação de uma parte física de informação que é usada ou produzida por um processo de desenvolvimento de *software*, por implantação ou operação de um sistema [55]. Estão inclusos na lista de artefatos, não limitados a estes, os modelos, os códigos fonte e os *scripts* de banco de dados.

Por estes termos, evolução de sistemas é qualquer criação ou alteração realizada em um ou mais artefatos que compõe o código fonte ou sua especificação, seja por inclusão ou remoção de fragmentos de informação.

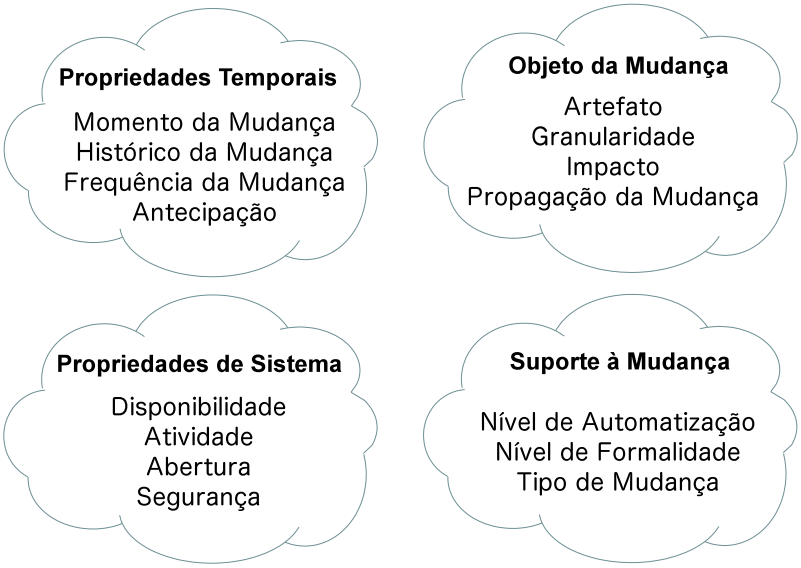
2.1.1 Taxonomia para Mudanças de *Software*

Cada aplicação tem particularidades que podem exigir formas específicas de evolução. Conhecer essas particularidades aumenta as chances de se aplicar soluções mais adequadas ao problema. Para uma melhor compreensão do tema, Buckley e seus colegas [16] apresentaram uma taxonomia que classifica a *mudança de software* considerando as técnicas e as ferramentas empregadas. O trabalho toma por base os seguintes temas lógicos: *propriedades temporais*, *objeto da mudança*, *propriedades do sistema* e *suporte da mudança*, os quais estão focados respectivamente em *quando*

[1] Neste trabalho, trataremos os termos *software*, *sistema*, *aplicativo* e *aplicação* como sinônimos.

a mudança ocorre, *onde* a mudança ocorre, *o que* está sendo mudado e *como* essa mudança é realizada. A Figura 2.1 apresenta as dimensões de avaliação, agrupados por tema.

Figura 2.1: Temas e dimensões da mudança de software



Fonte: [16]

Mudanças Temporais

As dimensões pertencentes ao tema temporal descrevem propriedades relacionadas ao momento e à frequência com que as mudanças ocorrem. Correspondem à questão "*quando*".

Momento da Mudança: dimensão que classifica as mudanças quanto ao momento exato em que o *software* sofre alteração, considerando os estados do ciclo de vida de execução: *estático*, *tempo de carregamento* e *dinâmico*. A evolução estática é aquela na qual o desenvolvedor encerra a execução da aplicação, altera o código-fonte, compila o novo programa (a depender da tecnologia) e submete para novo carregamento e execução. Essa é a abordagem mais tradicional de evolução de *software*. Algumas mudanças podem ser planejadas para ocorrer na aplicação a depender de critérios que podem ser determinados no momento da carga da aplicação. Podemos considerar como exemplo o `ClassLoader`, mecanismo da linguagem Java que permite a carga de classes no momento do carregamento da aplicação. No outro extremo, está a mudança dinâmica. Mudanças dessa natureza realizam alterações na aplicação sem finalizar a sua execução.

Histórico da Mudança: de acordo com essa dimensão, mudanças podem ocorrer *sequencial* ou *paralelamente*. Diz-se que uma mudança é sequencial quando sucessivas mudanças exigem que a anterior tenha sido completamente finalizada para início da próxima. Assim, tem-se sempre uma única versão da aplicação. Por outro lado, mudanças paralelas permitem que dois ou mais desenvolvedores realizem a alteração ao mesmo tempo.

Frequência da Mudança: dimensão que trata da frequência com que mudanças ocorrem. Por essa classificação, elas podem ocorrer *continua* ou *periodicamente*, ou ainda em intervalos *arbitrários*. Mudanças são consideradas contínuas se ocorrem várias vezes, a todo momento e sem nenhuma periodicidade claramente definida. Já as periódicas respeitam determinados prazos para ocorrer. Por exemplo, muitas empresas de desenvolvimento determinam calendário (mensal, bimestral, anual) para a geração de novas versões com a incorporação de mudanças. Por fim, as mudanças arbitrárias, como o nome já sugere, não seguem nenhum padrão temporal.

Antecipação: dimensão que classifica as mudanças como *previsíveis* ou *não-previsíveis*. A definição do quão podemos premeditar a ocorrência de uma mudança possibilita preparar a aplicação desde seu projeto para sua ocorrência. Esse é o caso das mudanças previsíveis. As mudanças não-previsíveis não são do conhecimento da equipe de desenvolvimento antes de seu surgimento.

Objeto da Mudança

As dimensões participantes desse tema descrevem a localização no sistema *onde* as mudanças ocorrem, além dos mecanismos de suporte a elas.

Artefato: dependendo do processo de desenvolvimento a ser utilizado, os envolvidos na construção da aplicação vão elaborar diversos tipos de artefatos de vários níveis de abstração. Documentos de requisitos, modelos de análise, artefatos de projeto e descrições de arquitetura são exemplos de artefatos que se juntam aos códigos-fonte durante as etapas de criação do *software*. Essa dimensão busca avaliar quais artefatos serão alterados para determinar o mecanismo adequado que irá realizar a mudança.

Granularidade: a granularidade está ligada com a escala do artefato que sofrerá a mudança. Pode ser toda a aplicação, um subsistema, uma subrotina, ou até mesmo uma simples variável. A classificação é amparada pelos conjuntos de *mudanças de*

fina granularidade, quando a alteração é restrita a apenas um arquivo, ou *grossa granularidade* para alterações acima disso, porém outras classificações com níveis intermediários também podem ser utilizadas.

Impacto e Propagação da Mudança: mudanças precisam investigar o seu impacto no sistema. Por exemplo, a alteração da assinatura de um método pode gerar apenas uma mudança local, caso a sua chamada seja restrita a sua classe, ou um impacto a nível de sistema, se sua chamada está distribuída por todo código [2].

Propriedades de Sistema

Este tema diz respeito às caraterísticas dos *softwares* alvo de mudanças e seus atributos. Responde a questão *o quê*.

Disponibilidade: algumas aplicações precisam manter-se permanentemente em execução em função de sua criticidade. Já para outras, é aceitável que possam ser paradas por algum tempo de forma a permitir a adição ou alteração de novas funcionalidades. De imediato, podemos perceber que alterações dinâmicas certamente são uma exigência para aplicações de disponibilidade permanente.

Atividade: para *softwares* que realizam evoluções dinâmicas, podemos classificá-los como *reativos* e *proativos*. Sistemas reativos necessitam de algum evento ou alguma intervenção externa para realizar a mudança, enquanto que sistemas ativos têm a capacidade de identificar essas situações para proceder a alteração.

Abertura: o nível de abertura de um *software* é determinado pela sua capacidade de receber extensões. Sistemas *abertos* permitem que novas funcionalidades sejam incorporadas à aplicação (estática ou dinamicamente), enquanto que os sistemas *fechados* tem todas as suas funcionalidades definidas em seu próprio código. O suporte a *Plug-ins* é um bom exemplo de abertura de *software*.

Segurança: o termo *segurança* pode ser utilizado para relatar várias preocupações distintas, como proteção a vírus, restrição de acesso a funcionalidades do sistema ou ainda *comportamento seguro*. Aqui iremos ressaltar a terceira opção, a qual busca garantir que o sistema não entre em um estado inválido, ou seja, que não apresente erros de execução, inclusive após a realização de mudanças. Em alguns sistemas, como os de alta criticidade, é desejável que haja garantia de qualidade dessa propriedade do sistema.

[2] O trabalho original [16] trata *impacto* e *propagação* como dimensões diferentes. Em função de sua leve distinção, optamos por unificá-los neste resumo.

Suporte à Mudança

Este tema descreve dimensões que influenciam os mecanismos de suporte a mudança, além de oferecer critérios para classificação desses mecanismos.

Nível de Automatização: o nível de automatização busca definir até que ponto o sistema é capaz de realizar mudanças em si mesmo sem instruções externas. Enquanto a dimensão *atividade* determina o nível de habilidade que o sistema tem de identificar quando deve mudar, essa dimensão avalia qual a competência (conhecimento) que o sistema tem do que fazer. Sistemas *automatizados* podem realizar todas as mudanças sem intervenção externa. Sistemas *manuais* estão na outra extremidade, tendo total dependência de instruções para realização da mudança. Uma classificação intermediária são os sistemas *semi-automatizados* os quais intercalam as alterações entre partes automatizadas e instruções externas.

Nível de Formalidade: dimensão que determina o nível de formalismo matemático a que o mecanismo de mudança está baseado. Abordagens baseadas em grafo, em álgebras de processos ou em lógica formal são alguns formalismos frequentemente utilizados [49].

Tipo de Mudança: Mudanças são classificadas como *estruturais* e *semânticas*. De modo simplista, mudanças estritamente estruturais não causam alterações no comportamento do sistema, enquanto que as semânticas o fazem. Entretanto, esta avaliação depende do escopo avaliado. É possível que em uma mudança haja alteração de comportamento no nível de método, porém avaliando o componente inteiro, como uma caixa preta, o comportamento permanece inalterado. Além disso, os tipos não são mutualmente exclusivos.

2.2 Componentes de *Software*

Para facilitar o processo de criação (e evolução) de sistemas, muitas técnicas e tecnologias tem sido propostas ao longo das décadas, como, por exemplo, a migração das linguagens de máquina para linguagens de alto nível e a evolução dos vários paradigmas de programação com diferentes capacidades de abstração.

Na direção dos paradigmas, com o uso da programação estruturada, os programas tornaram-se mais modulares, deixando de ser estritamente monolíticos e passando a possuir divisões em subrotinas. Assim, com um maior grau de modularidade, é possível desfrutar de um maior nível de reuso e melhor controle de impacto de alteração ao estabelecer adequadamente a divisão do *software* em partes.

Surge em seguida a programação orientada a objetos, trazendo consigo a organização do código por classes. Os conceitos de encapsulamento e herança permitem um melhor grau de modularização, de modo a potencializar o reuso de código, e trazer ainda mais facilidade para a manutenção. Dessa vez, o uso de classes permite que porções maiores de funcionalidades sejam reutilizadas.

Apesar dos avanços, os níveis de manutenibilidade e reusabilidade providos por essas técnicas ainda são muito incipientes frente a complexidade dos *softwares* atuais, conforme apresentado na Tabela 2.2.

Tabela 2.1: Comparação entre paradigmas. (PE - programação estruturada; POO - programação orientada a objetos; POC - programação orientada a componentes)

Características	**PE**	**POO**	**POC**
Divisão e Conquista	Sim	Sim	Sim
Gerencia a complexidade;			
Divide um problema grande em partes menores.			
Unificação de Dado e Função		Sim	Sim
Uma entidade de software combina dados e as funções processam esses dados;			
Melhora a coesão.			
Encapsulamento		Sim	Sim
O cliente da entidade de software é separado da forma como esta entidade de software armazena seus dados ou como suas funções são implementadas;			
Reduz o acoplamento.			
Identidade		Sim	Sim
Cada entidade de software tem um identificador único.			
Interface			Sim
Representa a especificação das dependências;			
Divide a especificação do componente em interfaces;			
Restringe a dependência inter componentes.			
Implantação			Sim
A unidade de abstração pode ser implantada de forma independente.			

Fonte: [68]

Diante do cenário apresentado, o desenvolvimento de *software* busca inspiração na indústria, onde máquinas e equipamentos complexos são construídos a partir da composição de partes menores, utilizando encaixes e conexões previamente acordadas. Assim, o desenvolvimento de *software* passa a se organizar também

em partes, agora chamadas de *componentes*.

Podemos definir um componente de *software* da seguinte forma [68]:

Um componente de software é uma parte de um programa autocontida e independetemente implantável, com funcionalidades bem definidas e que pode ser montado a partir de outros componentes através de suas interfaces.

Um componente se comunica com o mundo externo através de suas interfaces de entrada ou saída. As interfaces de entrada proveem dados ou serviços, enquanto que as de saída estabelecem um mecanismo para consumir dados ou serviços de outros componentes. As interfaces definem contratos que especificam formalmente:

- como as aplicações e os componentes usuários podem acessar o componente servidor (assinaturas de suas operações válidas);
- qual o comportamento do componente em sua execução, quando o comportamento afeta o ambiente no qual está executando;
- o(s) resultado(s) fornecido(s) devido a sua execução, em função das operações realizadas.

Dessa forma, os componentes podem ser enxergados como caixas-pretas de modo que o conhecimento sobre o seu funcionamento interior não é necessário para o funcionamento de seus componentes usuários, desde que o contrato de sua interface seja obedecido.

Com essa organização, a programação orientada a componentes possui três grandes objetivos: administrar a complexidade, gerenciar mudanças e reutilizar componentes previamente escritos.

Conforme apresentado anteriormente, a complexidade é inerente ao *software* atual e tende a crescer cada vez mais. A abordagem orientada a componentes provê um mecanismo intrínseco para lidar com isso.

Utilizando componentes, a realização de mudanças passa a ter um escopo localizado. Devido ao estabelecimento de interfaces formais, alterações no interior de um componente, por exemplo, não possuem consequências em nenhum dos outros, considerando os aspectos funcionais, desde que sejam mantidas as interfaces e o significado dos valores retornados por tais funcionalidades.

Reusabilidade é um aspecto fundamental para que se possa minimizar o tempo e o custo de desenvolvimento. Por definição, um componente possui independência

suficiente para ser reusado em outros cenários, característica viabilizada novamente pela definição de interfaces.

Wang e Qian [68] estruturam esses objetivos dentro de 5 princípios inter-relacionados que fazem apologia ao uso da componentização no desenvolvimento de *software*:

- **Princípio 1: Componentes representam decomposição e abtração**
  Uma estratégia básica e eficiente para tratar grandes problemas computacionais é a "divisão e conquista" e componentes representam bem essa ideia - porções de *software* com claras divisões que permitem mais facilmente o desenvolvimento por diferentes pessoas. Do ponto de vista da abstração, uma modelagem adequada de componentes permite uma visão macro das partes revelantes do sistemas e de suas interações.

- **Princípio 2: Reusabilidade pode ser alcançada em vários níveis**
  Em processos de desenvolvimento que utilizam documentações de requisitos, análise e projeto, a reutilização pode ocorrer a partir do nível documentação. Além disso, em testes formais, considerando similaridade de objetivos de vários componentes, casos de testes podem ser reutilizados para garantir o pleno funcionamento.

- **Princípio 3: Desenvolvimento orientado a componentes aumenta a confiança do *software***
  A componentização inerentemente facilita a validação de requisitos críticos e verificação de segurança. Além disso, de forma coerente ao item anterior, componentes já existentes tendem a ser mais testados, e portanto mais seguros.

- **Princípio 4: Desenvolvimento orientado a componentes pode aumentar a produtividade do *software***
  Na maioria dos casos, criar uma aplicação a partir do reuso de componentes é mais rápido que o desenvolvimento a partir do zero. Um outro aspecto consiste no fato de que a componentização provê facilidades para a divisão de trabalho entre diversas equipes.

- **Princípio 5: Desenvolvimento orientado a componentes promove a padronização de *software***
  A necessidade de se especificar com maior rigor e cuidado as interfaces dos

componentes, induz a criação de padrões corporativos, chegando até a níveis interempresarial de mercado.

Fica evidente que a programação orientada a componentes vem como uma etapa complementar às outras técnicas de programação, tornando-se ortogonal aos paradigmas de programação. Porém, assim como a programação orientada a objetos comumente faz uso de programação estruturada e imperativa, o desenvolvimento de componentes faz uso comum da modelagem baseada em classes e objetos. Entretanto, devemos ressaltar que o desenvolvimento de componentes não necessariamente deve ser feito sob o paradigma de orientação a objetos, podendo ser usado com linguagens imperativas ou funcionais, por exemplo.

2.2.1 Modelo e Plataforma de Componentes

Construir uma aplicação fazendo mero uso das técnicas de desenvolvimento orientado a componentes não garante total operabilidade entre os componentes produzidos. Por exemplo, poderá não ser possível compor um componente escrito em uma linguagem de programação x, em uma aplicação orientada a componentes produzida por uma outra linguagem de programação y, mesmo que o desenvolvimento de ambos tenha respeitado todos os princípios da orientação a componentes. A linguagem é apenas um exemplo de limitação que componentes podem encontrar para serem ligados. Mesmo dentro de uma mesma linguagem de programação, componentes podem ser inoperáveis se suas interfaces não puderem ser conectadas.

Para que sejam interoperáveis, é necessário estabelecer uma base onde são definidas a *semântica* dos componentes (o que são componentes), a *sintaxe* dos componentes (como eles são definidos, construídos e representados) e a *composição* dos componentes (como eles são compostos ou montados) [40]. A este conjunto de regras básicas chamamos *modelo de componente*.

> "*Um modelo de componente define padrões para (i) propriedades que componentes individuais devem satisfazer, e (ii) métodos para composição de componentes.*" [21]

Amparado neste novo conceito, podemos apresentar uma definição mais precisa de componente de *software*:

> "*Um componente é um elemento de software que obedece um modelo de componente e que pode ser independentemente implantado e composto*

sem alteração de acordo com o padrão de composição." [29]

Apesar de diferente, a definição acima não invalida, mas sim reforça, todos as características e definições já apresentadas. O conceito de modelo de componente apenas inclui algumas restrições para que componentes possam de fato interagir entre si.

Por exemplo, a definição e características referente à interface de componentes que fora apresentadas anteriormente continuam válidas. A interface corresponde a uma das *propriedade* que *componentes individuais devem satisfazer*, além de está contida no conjunto de *métodos para composição*. Obviamente, as propriedades e os métodos não estão limitadas a interface. Este grupo ainda inclui outras propriedades funcionais, as propriedades não-funcionais, além de regras que as conexões devem atender.

Além disso, apenas conceber um modelo, formalizando a *semântica*, *sintaxe* e *composição* de componentes não é suficiente para torná-lo prático. Ferramentas de suporte são necessárias para que o modelo possa ser operacional. A esse conjunto básico de ferramentas chamamos *plataforma de componentes* [21].

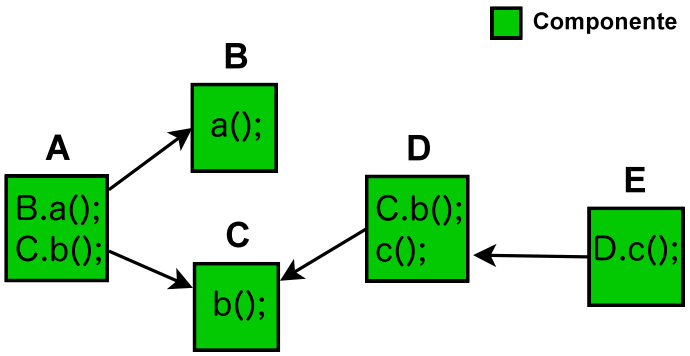
"*Plataforma de componentes é a materialização do modelo através da implementação de ferramentas que permitam a implantação, integração e execução de componentes.*"

2.3 Conectores

Sob a pespectiva do desenvolvimento baseado em componentes, passamos a observar um *software* como um conjunto de componentes que cooperam a fim de implementar o interesse da aplicação a qual se destina. Entretanto, para realizar essa cooperação, é fundamental que possam ser estabelecidos mecanismos de interações entre os componentes, os quais estão relacionados à comunicação, à coordenação, à conversão ou à facilitação envolvendo componentes, ou ainda uma combinação de mais de uma dessas interações [52].

Inicialmente, podemos propor que os componentes façam chamadas diretas uns aos outros para que ocorra a computação. Assim, a título de exemplo, se um componente A deseja fazer uso da funcionalidade implementada por um componente B, o primeiro executa uma chamada direta ao método do segundo conforme apresentamos na Figura 2.3.

Figura 2.2: Chamada direta entre os componentes A e B.



Fonte: [39]

A simplicidade do exemplo mostra que esse tipo de abordagem acarreta uma série de problemas no contexto arquitetural. O primeiro deles é o acoplamento. A associação direta entra A e B dificulta a reusabilidade de A, uma vez que os dois componentes estão interligados por código fixo, ou seja, a ligação está incluída dentro da definição do componente.

Um outro problema é a inclusão da lógica de comunicação dentro do componente. Em outras situações, para atender a interface de outro componente, os dados podem necessitar de reorganização. Essa lógica, também definida como protocolo, irá ficar sob a responsabilidade do componente chamador, adicionando novas responsabilidades a ele e desviando o coesão do seu objetivo. Além disso, todas as informações sobre interações passam a ficar dispersas entre os vários componentes, o que inviabiliza o reuso desses protocolos [39].

Assim, foram definidos artifícios mais elaborados para a composição dos componentes de forma a resolver, ou ao menos mitigar, esses problemas. No geral, ao invés de se realizar chamadas diretas uns aos outros, são definidos novos elementos que irão intermediar a interação entre os componentes, chamados *conectores*.

Da mesma forma que definimos componentes com a intenção de encapsular um comportamento potencialmente reutilizável, faremos o mesmo para os conectores, porém com ênfase no encapsulamento da interação entre componentes.

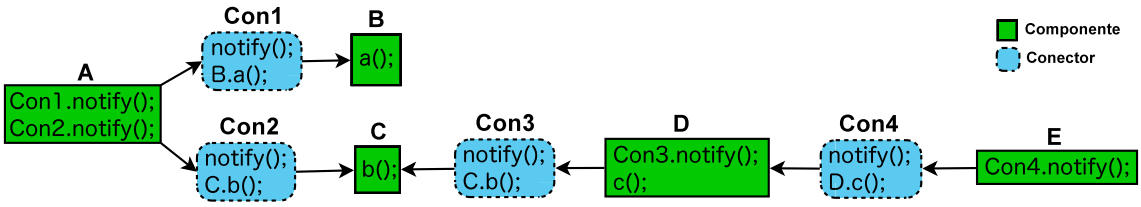
Podemos ter interações extremamente simples, como chamadas de métodos, passando por outras bastante complexas onde seja necessário executar transformações de dados complexas. Independentemente de sua complexidade, um conector passa a ser o responsável pela interação. Isso porque, conforme já motivamos anteriormente, além do reuso da interação, buscamos também um maior

desacoplamento dos componentes, prevalencendo a sua independência funcional.

Os conectores são definidos como elementos que possuem uma interface de forma a permitir que os componentes possam se conectar. À primeira vista, essa afirmação pode parecer estranha ao considerar que caímos no mesmo problema de conexão ao desejar unir componentes e conectores. Entretanto, esse tipo de conexão é simplificada e padronizada de forma que a linguagem que especifica as interações (definida adiante) já possui mecanismos nativos para a conexão desses elementos de primeira classe.

Assim, em termos práticos, utilizando o mesmo exemplo anteriormente apresentado, se um componente A deseja consumir aquela mesma computação definida no componente B, um conector será definido e inserido entre eles, de forma que A não faça chamadas diretamente a B. Num primeiro momento, utilizaremos o conector apenas como um intermediador da chamada, ilustrado na Figura 2.3.

Figura 2.3: Uso de conectores entre os componentes A e B.



Fonte: [39]

Essa nova abordagem diminui o acoplamento, de forma que os componentes não mais dependem diretamente uns dos outros. A dependência existente agora é entre os componentes e os conectores. Em muitos casos, conseguimos embutir no conector as responsabilidades de comunicação. Porém, o controle do fluxo de execução ainda continua distribuído entre os componentes.

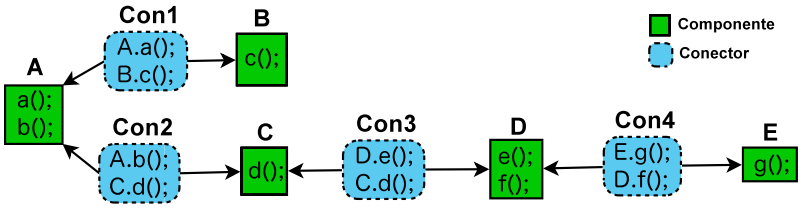
Para permitir um grau ainda maior de desacoplamento, surge um novo conceito de conectores, os chamados *conectores exógenos*.

Em um cenário ideal, os componentes devem especificar e computar apenas as suas responsabilidades e suas interfaces devem expor as dependências que sejam necessárias para a realização de sua tarefa. Isso quer dizer que os componentes não devem ter conhecimento sobre os conectores, mas sim o contrário.

Conectores exógenos são definidos de forma tal que circundam os componentes envolvidos. O conector passa a ter a responsabilidade de efetuar as chamadas aos componentes e gerenciar a comunicação e o fluxo de execução. Dessa

forma, componentes não mais iniciam computações, as quais serão disparadas por conectores, conforme apresentado na Figura 2.3.

Figura 2.4: Uso de conectores exógenos entre os componentes A e B.



Fonte: [39]

Além dos já mencionados benefícios relacionados ao desacoplamento e reuso, essa abordagem facilita a verificação de propriedades de componentes, sendo também mais adequada para a composição hierárquica destes [39].

Apesar da distinção de responsabilidades, componentes e conectores possuem muitas semelhanças. Possuem interface de comunicação, são construídos por porções de códigos executáveis e encapsulam um interesse. Assim, apenas o tipo de interesse é que de fato distingue um componente de um conector [33]. Por essa perspectiva, definimos conectores como componentes que encapsulam interesses de iteração entre componentes de computações, ou seja, a comunicação, coordenação, conversão, facilitação ou suas combinações [18].

2.4 Descrição Arquitetural

As linguagens de propósito geral foram concebidas para definir estruturas de dados e operar algoritmos sobre elas. Utilizá-las para definir a arquitetura de sistemas de *software* nos traz dois potenciais problemas: [62]

- Expressar interações não suportadas diretamente pela linguagem de programação irá requerer a codificação de sequência de chamadas de procedimentos, mesmo quando conceitualmente essa iteração esteja relacionada a uma ação atômica;
- Desviar insatisfatoriamente o interesse central das linguagens de programação ao incluir novos construtores para suportar necessidades de modelagem arquitetural;
- Redução da capacidade de raciocínio sobre a arquitetura do *software*, incluindo

os seus aspectos comportamentais, ao utilizar uma linguagem com capacidade expressiva muito além das necessidades para essa finalidade.

Nos anos 1970, os pesquisadores passaram a dar maior atenção ao projeto de *software* tendo em vista problemas enfrentados no desenvolvimento de sistemas na década anterior. Esses trabalhos levaram ao amadurecimento de técnicas, ferramentas e linguagens nos anos 1980 que permitiram grande avanço na concepção e análise de sistemas, a exemplo do uso de técnicas de descrição formal e da presença de sistemas de tipos de dados mais sofisticados.

Porém, os avanços obtidos com as pesquisas focadas em projeto de *software*, ainda eram voltados a um nível de abstração muito baixo. Os estudos resultaram principalmente em técnicas e notações que foram incorporados em linguagens de programação de propósito geral, notoriamente no tocante à modularização do sistema: porções fragmentadas de código que permitem a compilação em separado, possibilidade de compartilhamento de bibliotecas e ainda a definição de interfaces de módulos [56].

É nesse contexto que Shaw e Garlan realizaram um estudo mais abrangente sobre as possíveis técnicas e métodos disponíveis até então para se avaliar a pertinência de utilizá-las para descrever uma arquitetura [63].

Assumimos a seguinte definição de arquitetura de *software* [64]:

"Uma arquitetura de software corresponde a descrição dos elementos que compõe o sistema, a interação entre eles, os padrões que guiam suas composições e as restrições desses padrões."

Diagramas informais, modularização suportada por linguagens de programação, linguagens de interconexão de módulos, linguagens que suportam formas alternativas de interação e notações para estilos arquiteturais especializados, segundo o trabalho, são inadequadas para descrever uma arquitetura de *software* de forma precisa, por falhar no atendimento de pelo menos uma das seis classes de propriedades fundamentais para a descrição de uma arquitetura. São elas [63]:

- Composição: deve ser possível descrever um sistema como uma composição de conectores e componentes independentes;
- Abstração: deve ser possível estabelecer os papéis que os componentes e suas interações exercem na arquitetura de *software* de forma clara e explícita;

- Reusabilidade: deve ser possível o reuso de componentes, conectores e padrões arquiteturais em diferentes descrições de arquitetura, mesmo que eles tenham sido desenvolvidos fora do contexto do sistema arquitetural;
- Configuração: descrições arquiteturais devem concentrar-se na descrição da estrutura do sistema, independente dos elementos estarem estruturados. Deve ainda suportar a reconfiguração dinâmica;
- Heterogeneidade: deve ser possível combinar múltiplas e heterogêneas descrições arquiteturais;
- Análise: deve ser possível realizar ricas e variadas análises de descrições arquiteturais.

Na verdade, diante dos vários trabalhos publicados sobre o assunto, observa-se que não há total consenso sobre o que uma arquitetura deve descrever, nem tampouco o nível de detalhe desta descrição [51]. Porém, os estudos, embora divergêntes, possuem vários pontos em comum, de forma que podemos aceitar os objetivos listados acima como válidos.

Dentro dessa concepção, surge a necessidade de uma nova classe de linguagens que possam atender a esses anseios: as *linguagens de descrição de arquitetura* - ADL (do inglês *architecture description language*).

2.4.1 Linguagem de Descrição de Arquitetura (ADL)

Uma ADL deve ser capaz de expressar os requisitos da especificação de uma aplicação. O conceito de arquitetura de *software* surge como uma analogia à construção civil. Lá, espera-se que a arquitetura possa ser expressa de várias formas diferentes: mestres-de-obra desejam conhecer detalhes da construção, eletricistas desejam observar um mapa elétrico, já os responsáveis pelas tubulações sanitárias desejam um mapa hidráulico, enquanto que os engenheiros estruturais irão se ater à fundação e estruturas de sustentação. Apesar de ser o mesmo projeto, temos visões diferentes sobre ele. Assim, os vários papéis envolvidos com a construção de *software* também desejam inclinar mais atenção em uma visão específica do projeto, o que motiva a existência das diferentes visões [56]. A linguagem pode apresentar visões hierárquicas da aplicação, além de visões por interfaces gráficas, de forma a facilitar a compreensão da arquitetura [51].

Apesar do interesse na modelagem da aplicação, a ADL não se resume a fornecer uma notação formal para o modelo de relacionamento entre componentes e

Figura 2.5: Arcabouço de classificação e comparação de ADL.

1. Características de Modelagem de Arquitetura
- **1.1. Componentes**
  - 1.1.1. Interface
  - 1.1.2. Tipos
  - 1.1.3. Semântica
  - 1.1.4. Restrições
  - 1.1.5. Evolução
  - 1.1.6. Propriedades Não-funcionais
- **1.2. Conectores**
  - 1.2.1. Interface
  - 1.2.2. Tipos
  - 1.2.3. Semântica
  - 1.2.4. Restrições
  - 1.2.5. Evolução
  - 1.2.6. Propriedades Não-funcionais
- **1.3. Configurações Arquiteturais**
  - 1.3.1. Compreensibilidade
  - 1.3.2. Composicionalidade
  - 1.3.3. Refinamento e rastreabilidade
  - 1.3.4. heterogeneidade
  - 1.3.5. Escalabilidade
  - 1.3.6. Evolução
  - 1.3.7. Dinamismo
  - 1.3.8. Restrições
  - 1.3.9. Propriedades Não-funcionais

2. Suporte de Ferramentas
- 2.1. Especificação Ativa
- 2.2. Visões Múltiplas
- 2.3. Análise
- 2.4. Refinamento
- 2.5. Geração de Implementação
- 2.6. Dinamismo

Fonte: [51]

conectores. Espera-se que essa especificação possa ser executada, relacionando-se a nível de implementação com os artefatos, de forma a estabelecer a devida instanciação e uso destes.

No princípio, muito se discutia sobre os requisitos e objetivos que as ADL deveriam atender. Um dos trabalhos mais concisos [51] propõe inclusive uma classificação dessas linguagens. Os critérios utilizados nessa classificação são apresentados na Figura 2.5.

Observando os critérios utilizados, notamos que essa avaliação leva em consideração apenas questões técnicas, buscando oferecer respostas sempre à luz dos princípios da computação.

Outro trabalho desses mesmos autores [50] mostra o quanto é limitada a concepção e avaliação de ADL apenas por critérios técnicos. Além das questões tecnológicas, o domínio (área do conhecimento) onde a linguagem é utilizada e aspectos de negócios também devem ser considerados, definindo assim os três *pilares de interesses*.

- **Tecnologia:** focado nos desafios técnicos relacionados principalmente com a engenharia de *software*. Tem o objetivo de estabelecer os meios pelos quais a arquitetura será representada, identificando as abstrações e os conceitos que serão utilizados na modelagem das aplicações. Também se estende quanto ao suporte ao desenvolvimento, no sentido de fornecer ambientes e ferramentas;

- **Domínio:** com ênfase na inclusão de conhecimento específico da área de atuação da aplicação nas abstrações da linguagem. Espera-se, por exemplo, que entidades e relacionamentos advindos do domínio do problema alvo possam

ser expressos e modelados por elementos de primeira classe da linguagem, de forma a representar melhor as relações no mundo real;

- **Negócio:** busca trazer questões de negócio para a notação a ser desenvolvida com o objetivo de permitir a organização das visões por níveis de abstração do projeto em consonância com a segurança das informações e agregar características que reflitam peculiaridades do processo de desenvolvimento do domínio da aplicação, contribuindo para o trabalho colaborativo.

Esse entendimento está baseado em uma nova forma de julgar as linguagens de descrição de arquitetura. Ao invés de avaliar aspectos objetivos ligados às suas características (elementos de primeira classe existentes, formas de modelar os relacionamentos, entre outros), o novo trabalho sugere que as notações sejam avaliadas quanto a sua pertinência em modelar e abstrair os conceitos de seu domínio, principalmente considerando o quão adequadas são sob a ótica de todos os envolvidos em seu ambiente de uso (*stakeholders*).

O grande foco dado à área técnica, até então é um reflexo da composição das equipes de projeto que propõe e especificam tais linguagens, normalmente dominados por pessoal da área de ciência da computação. Porém, a presença e incentivo de pessoal das áreas interessadas é essencial para o desenvolvimento de novas linguagens de descrição de arquitetura.

Passa a ser mais razoável admitir uma definição mais flexível para ADL:

Uma linguagem de descrição de arquitetura é uma notação capaz de expressar um modelo de arquitetura, a qual, por sua vez, é um artefato ou documento que captura algumas ou todas as decisões de projeto que forma a arquitetura de um sistema [50].

Vale ressaltar que essa nova visão sobre o papel e objetivos da ADL não invalidam os trabalhos anteriores. Mostram apenas que a maioria dos trabalhos realizados até então foram desenvolvidos focados apenas nos requisitos de engenharia de *software* e para avaliar esses critérios, o conhecimento produzido continua válido.

2.4.2 Darwin e Rapide

A título de ilustração, iremos apresentar duas linguagens de descrição de arquitetura bastante difundidas na década de 1990.

Darwin

Darwin [45] é uma linguagem declarativa que permite a especificação de programas distribuídos a partir de uma estrutura hierárquica. Os componentes provêem e demandam serviços, chamados respectivamente de *provides* e *requires*, a partir dos quais é possível especificar a estrutura de uma aplicação mediante a ligação dos componentes.

Uma vez que os componentes seguem uma estrutura hierárquica, a composição de um conjunto de componentes pode ser vista como um *componente composto* que pode ser também combinado através de seus serviços *provides* e *requires* que ainda não estejam conectados.

Por se tratar de uma linguagem extritamente declarativa, ao invés de imperativa, não há operações para a desconexão dos serviços [46]. Por este mesmo motivo, todo o comportamento está encapsulado nos componentes, não sendo possível decrever fluxos de execução.

Rapide

Rapide [43] é uma linguagem de definição de arquitetura que une a capacidade de representação estrutural com características de linguagem de simulação baseada em eventos. Para Rapide, uma arquitetura é composta por *interfaces*, *conectores* e *restrições*, que têm a função de especificar, respectivamente, o comportamento dos componentes, as formas de comunicação entre eles e as restrições sobre esses comportamentos e comunicações.

A linguagem foi projetada com o propósito de permitir uma fácil modelagem de protótipos que permitam o estudo do comportamento de equipamentos, antes de sua construção, como também, servir de guia para a construção de sistemas de *software*.

Rapide utiliza as *interfaces* como abstração de componentes. Similarmente a Darwin, as interfaces definem portas para prover (*provided*) e requerer (*required*) eventos de outras interfaces. O seu comportamente é definido através da orientação a eventos, mediante a observação e geração destes.

A execução de uma arquitetura descrita em Rapide, considerando o seu propósito de experimentação, resulta na definição de um conjunto de eventos em ordem parcial, onde são apresentadas as suas dependências. Este tipo de linguagem descreve portanto a simulação da estrutura e comportamento de um *software* ou *hardware* a ser projetado.

2.5 Reconfiguração Dinâmica

Uma vez estabelecida a composição inicial dos componentes para formar a aplicação, ou seja, a sua configuração arquitetural, vários podem ser os motivos para que essa aplicação tenha necessidade de alteração. Algumas dessas motivações podem demandar que a alteração ocorra em tempo de execução. Substituição de componentes em sistemas que demandam garantia de alta disponibilidade [11], sistemas nos quais a criticidade impede que evoluções na aplicação ocorram apenas quando ela seja desligada [10] ou mesmo sistemas cuja natureza intrínseca exige alto potencial de mudanças [67], são alguns exemplos.

Reconfiguração dinâmica é um mecanismo que permite que um *software* tenha a sua composição arquitetural alterada em tempo de execução, a qual pode se utilizar de diversas estratégias para chegar a esse fim.

O propósito da reconfiguração dinâmica é permitir que uma aplicação evolua incrementalmente a partir de uma configuração para outra em tempo de execução, sem a necessidade de descarregá-la ou reiniciá-la e introduzindo o mínimo impacto na execução do sistema [69].

Para que a reconfiguração possa ocorrer, o mecanismo deve suspender parcialmente a execução, realizar a mudança e, após concluída, dar continuidade a partir do estado suspenso. Entretanto, antes da efetivação do processo de reconfiguração, a consistência da aplicação deverá ser certificada. Para ser considerada consistente, uma aplicação deverá atender a três critérios [37] [70] [69]:

- Satisfazer os requisitos de *integridade estrutural*;
- Garantir que os elementos que compõe a aplicação estão em *estados mútuos de consistência*;
- Garantir a manutenção das *invariantes do estado da aplicação*.

Requisitos de *integridade estrutural* definem e restringem a forma como os elementos da aplicação se relacionam. Por exemplo, suponhamos a substituição do componente A pelo componente B em uma aplicação qualquer. A definição de interface do componente B deve ser compatível com a interface do componente A, de forma a permitir os relacionamentos antes existentes. Além disso, todos os componentes cliente de A devem ter atualizadas as suas referências para B [69].

Estados mútuos de consistência devem ser atendidos para todo par de elementos que se relacionam na aplicação. Componentes estão em estados mútuos de

consistência se ao ser completadas as suas interações, estas são consideradas bem-definidas e cada componente está em um estado consistente. Em outras palavras, dois componentes são considerados em estados mútuos de consistência se as suas suposições sobre o resultado da interação são compatíveis [69].

Invariantes do estado da aplicação são predicados lógicos descritos em termos de um conjunto de estados dos elementos da aplicação. A garantia da manutenção da execução (*liveness*) depende dessas invariantes. Por exemplo, suponhamos que uma aplicação possua um componente X que tenha a responsabilidade de gerar identificadores únicos. Assim, essa aplicação tem uma invariante que expressa a unicidade dos identificadores enquanto a aplicação estiver em execução. Se substituirmos o componente X por outro X' que possua a mesma interface e não esteja em uso durante a operação, iremos atender aos dois critérios anteriores. Porém, se não informamos ao componente o último identificar gerado (supondo que o gerador conceda os identificadores de forma sequencial), a operação irá ferir uma invariante do sistema e, provavelmente, o fará ter um comportamento indesejado, resultado em erro de aplicação ou de resultado [69].

Uma vez que esses critérios sejam atendidos, concentraremos a atenção nas tecnologias empregadas para efetivar as reconfigurações. Apresentamos a seguir três estratégias para implementação de reconfiguração dinâmica. Duas delas utilizam artifícios para atribuir a ADL a responsabilidade pela realização da reconfiguração: utilização de linguagens interpretadas e aplicação dos conceitos de aspectos, enquanto que a terceira estratégia se utiliza de um mecanismo de alteração do código semi-compilado para máquinas virtual.

2.5.1 Suporte por Linguagens Interpretadas

Linguagens interpretadas possuem a característica de não necessitar que os códigos gerados sejam submetidos a um processo de compilação para serem colocadas em execução. Dessa forma, trechos do código podem ser alterados enquanto a aplicação funciona. Se uma ADL é especificada como uma linguagem interpretada, os programas configurados a partir dela gozam dessa mesma propriedade e reconfigurar uma aplicação não exige que ela venha a ser desligada. Um dos trabalhos nessa área [11] chega a essas conclusões ao especificar uma ADL de *script* que utiliza um processo de instanciação dinâmica de componentes.

Essa abordagem permite que as reconfigurações possam ser automáticas, ou seja, realizadas pela própria aplicação a partir de processos internos de melhoria de

desempenho ou *ad-hoc*, com a intervenção do usuário a partir do uso de ferramentas do modelo de componentes.

Do ponto de vista prático, o uso dessa técnica não apresenta grandes dificuldades, inclusive, a definição de uma ADL como interpretada pode simplificar o processo de execução. Entretanto, quando considerada no contexto de Computação de Alto Desempenho (CAD), a interpretação, num primeiro momento, aparenta ser inadequada. É sabido que essa forma de execução apresenta significativa sobrecarga de desempenho frente aos processos compilados, porém vislubrar um processo interpretado apenas como um orquestrador de componentes compilados, passa a ter outra conotação. Isso é possível ao considerar que os componentes de CAD têm grossa granularidade, ou seja, realizam computação intensiva de longa duração. Assim, a sobrecarga de desempenho nos trechos interpretados pode ser considerada insignificante frente ao tempo total de processamento. De toda forma, esse fator deve ser mensurado para verificar se o seu custo é aceitável.

2.5.2 Suporte por Uso do Conceitos de Aspectos

Definido de maneira simples, *programação orientada a aspectos* é o paradigma no qual se deseja separar interesses que são transversais aos interesses usados na decomposição canônica da aplicação.

No desenvolvimento de aplicações, objetiva-se modularizar os interesses em classes (no caso da orientação a objetos) e componentes. Alguns dos interesses estão distribuídos transversalmente pela aplicação de forma a permear vários módulos. Distribuir essas codificações pelos vários módulos não é uma boa solução. Para encapsular esses interesses, é definido o conceito de *aspecto* [34].

A implementação de um aspecto é chamado de *adendo* (*advice code*). Os adendos são inseridos na aplicação em *pontos de junção* (*join points*). Pontos de junção são geralmente definidos por expressões regulares com a finalidade de realizar o casamento com classes, métodos, atributos, exceções ou outros tipos possíveis de ações sobre o código orientado a objetos.

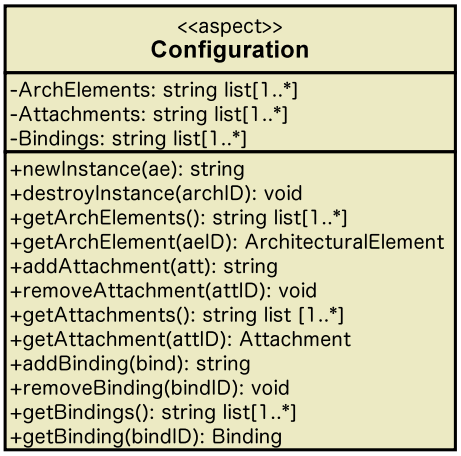
Alguns trabalhos, discutidos adiante, utilizam-se dos conceitos defendidos por esse paradigma para inserir reconfiguração dinâmica numa aplicação. Identificamos 3 abordagens diferentes:

- Aspectos na configuração;
- Aspectos em componentes.

- Aspectos em conectores;

PRISMA [67] apresenta o conceito de aspectos relacionados com a própria configuração da aplicação. Nesse modelo, além dos componentes e conectores, uma nova entidade, chamada `aspecto`, também é definida como elemento de primeira classe. Essa entidade é definida por uma classe, conforme a Figura 2.6, a qual armazena as referências das instâncias dos componentes e dos conectores que fazem parte do sistema. Essa classe também armazena o relacionamento entre eles. Assim, considerando que os metadados da aplicação são determinados por variáveis, a reconfiguração ocorre a partir da alteração do estado do sistema.

Figura 2.6: Definição do Elemento *Aspect*



Fonte: [67]

Considerando os benefícios da orientação a componentes e da orientação a aspectos, Pessemier e seus colegas [57] propõem o uso dos dois paradigmas dentro do mesmo modelo de componentes, fazendo surgir assim o *componente aspectual*.

Um componente aspectual encapsula um comportamento que transpassa a aplicação. O objetivo é que haja reuso desses comportamentos a partir do compartilhamento desse artefato.

Um comportamento aspectual é dado pela inclusão de uma interface *aspectual* no componente. Assim, os componentes são genuinamente aspectuais quando não há interfaces regulares. Este é um dos pontos de destaque da solução, uma vez que propõe um tratamento homogêneo para os vários tipos de componentes.

Os componentes podem se associar de quatro formas, definidas por todas as combinações entre componentes regulares e componentes aspectuais. Em essência, o que irá definir se o componente é regular ou aspectual em determinada interação é a interface que está sendo utilizada.

Encontramos ainda uma abordagem que atribui ao conector o comportamento dos aspectos. Em [10] é definida uma Linguagem de Descrição de Arquitetura Orientada a Aspectos que provê maior responsabilidade aos conectores. Os conectores definem o protocolo da reconfiguração, enquanto que os componentes (também chamados `aspectuais`) responsabilizam-se pelo comportamento a ser executado, o qual é definido por uma cláusula de *ação*. Por analogia, podemos dizer que essas cláusulas assemelham-se aos adendos. Enquanto que os protocolos definidos nos conectores são como pontos de junção. A rigor, conforme mencionado pelos autores, tanto o conector quanto o componente passam a ser aspectuais.

Tendo o foco em aplicações de CAD, a abordagem de inclusão de reconfiguração dinâmica através de aspectos apresenta vantagens e desvantagens potenciais.

O uso de aspectos provê mais uma forma de modularização da aplicação, oferecendo mais ferramentas para a divisão dos interesses. Porém, vale ressaltar que a inclusão dos aspectos representa também um aumento na complexidade do modelo, que necessita de inserção de mais um tipo de abstração. Esse aumento de complexidade pode prejudicar o seu entendimento conceitual, como também a compreensão e legibilidade das aplicações sob ele desenvolvidos.

Outro fator que deve ser avaliado é a forma de implementação desse recurso. Algumas implementações de aspectos identificam os pontos de junção e realizam a interceptação do fluxo de execução em tempo de execução, causando sobrecarga. Em princípio, uma implementação que realize a ligação dos aspectos aos seus pontos de junção em tempo de compilação aparenta ser mais adequado para modelos de CAD.

2.5.3 Suporte por Alteração de Código em Máquinas Virtuais

Máquinas virtuais têm se tornado uma importante ferramenta em sistemas computacionais para criação de aplicativos portáveis, tornando as restrições físicas transponíveis [66]. No tocante ao suporte à linguagem de programação, o uso dessas máquinas nos permite criar ambientes de execução similares aos interpretados, porém com avanços em processos de compilação parciais. Baseados nessas tecnologias, identificamos duas abordagens para lidar com reconfiguração dinâmica de aplicações.

Baseado em tecnologia Java e executando sobre uma máquina virtual (JVM[3]), foi criada uma obordagem [35] que se utiliza das ferramentas nativas da tecnologia

[3] *Java Virtual Machine*

para realizar alterações no código executável da aplicação.

Com o objetivo de dar suporte a processos complexos qde depuração, foi incluída na JVM uma API chamada *HotSwap* a qual, entre outras funcionalidades, é capaz de substituir classes em tempo de execução.

Existem uma série de regras que limitam as possíveis alterações, porém o trabalho cria artifícios que conseguem resolver os empecilhos iniciais e viabilizar a substituição de componentes em aplicações de CAD.

Uma outra abordagem diz respeito ao uso de MetaJava [36]. MetaJava é uma arquitetura de *software reflexiva*, na forma de API, que permite a alteração do estado interno da máquina virtual e de seus objetos. *Reflexão* (mais conhecido por seu termo em inglês, *reflection*) é a capacidade de um sistema computacional de examinar seus metadados e realizar alterações em tempo de execução.

O objetivo de MetaJava é explicitar funcionalidades que não fazem parte do modelo de linguagem de programação Java. De modo geral, podemos dizer que esta API permite que o programador instancie objetos, realize chamadas de métodos, ou mesmo altere o valor de variáveis através da indicação de seus nomes em formato *string*. No código fonte, por exemplo, não haverá uma chamada explícita ao método, mas sim uma chamada genérica onde o nome do método é passado como uma cadeia de caracteres. Com este tipo de artifício, podemos orquestrar a execução de um programa, similarmente a um processo interpretado.

Entre as três técnicas apresentadas, esta última tende a ser a melhor indicada para reconfiguração de aplicações de CAD, tendo em vista seu aparente melhor desempenho. Porém, também possui desvantagens. A reconfiguração passar a ter forte dependência de ferramentas específicas. Tentativas de usar o modelo em outras linguagens podem ser inibidas por limitações no suporte de máquinas virtuais, API ou outras ferramentas, devido à ausência de suporte a essas novas linguagens que se deseja inserir. Além disso, será necessário encapsular esse mecanismo de alteração dentro de alguma entidade, o que pode incorrer na criação de novas abstrações, as quais, similarmente às técnicas anteriores, aumentam a complexidade conceitual do modelo.

Capítulo 3

Plataformas de Componentes para Computação de Alto Desempenho

Este capítulo tem o propósito de apresentar o estado-da-arte de plataformas de componente voltadas à computação de alto desempenho, com especial ênfase na definição de componentes exógenos, no uso de linguagens de descrição de arquitetura e no suporte à reconfiguração dinâmica.

Iniciaremos este capítulo apresentando plataformas de componentes para computação de alto desempenho, juntamente com os requisitos peculiares que motivam a existência de modelos focados nesse fim. Serão apresentados o CCA, o Fractal e o HASH. Em especial, serão discutidos maiores detalhes sobre o HPE (*Hash Programming Environment*), a plataforma de componentes paralelos que serve como pano de fundo para o problema que motiva esta dissertação.

3.1 Computação de Alto Desempenho e Componentes

Aplicações científicas são as que mais apresentam requisitos de Computação de Alto Desempenho (CAD). Os simuladores, os otimizadores e os grandes modelos matemáticos, além de demandarem a realização de cálculos paralelos intensivos em sua execução, exigem conhecimentos multidisciplinares para o seu desenvolvimento [6]. A combinação desses dois fatores eleva significantemente a complexidade das aplicações. Reutilizar esse tipo de *software* torna-se fundamental para diminuir o tempo de escrita da aplicação além de garantir a confiabilidade, uma vez que a validação, homologação e testes de um resolvedor numérico linear, por exemplo, não é algo trivial. Acompanhando a tendência de outras áreas, a partir dos anos 2000, os pesquisadores da área científica identificaram na orientação a componentes uma

alternativa viável para lidar com o problema apresentado, levando às plataformas de componentes dos modelos CCA e Fractal, introduzidos adiante.

3.2 CCA

Nos anos 1990, plataformas de componentes como CORBA [54], COM [60] e JavaBeans [27], já eram aplicadas com sucesso em *software* de aplicações corporativas. Suas propriedades levaram a sua avaliação em aplicações com requisitos de CAD. Entretanto, essas plataformas mostraram-se inviáveis para tratar as necessidades específicas de aplicações de alto desempenho [6].

A plataforma COM tinha foco na área comercial. Suas linguagens não possuíam abstrações para formatos de dados paralelos ou tipos de dados científicos, como números complexos. As plataformas da Sun, JavaBean e Enterprise JavaBean, não ofereciam suporte a interoperabilidade de linguagens, assumindo que os componentes sempre estavam escritos em Java. Entretanto, essa linguagem se mostrava muito ineficiente para o processamento de alto desempenho. Mesmo considerando a chamada direta a códigos C ou C++, através de *Java Native Interface* [42], continuava a possuir uma grande sobrecarga, devido à máquina virtual. O padrão CORBA, por sua vez, conseguia bons resultados de desempenho na computação distribuída, porém padecia das mesmas deficiências de definição de abstrações adequadas para a área de CAD. Além disso, a execução de componentes no mesmo espaço de endereçamento tinha desempenho aquém do esperado.

Em princípio, fundamentalmente, uma plataforma de CAD deveria dispor de [6]:

- canais eficientes de comunicação paralela entre componentes;
- abstrações adequadas para a modelagem científico-matemático;

A partir dessas conclusões, pesquisadores ligados na sua maioria aos laboratórios nacionais do Departamento de Energia (DoE) do governo dos EUA, bem como universidades norte-americanas, reunidos no projeto CCTTSS (*Center for Component Technology for Terascale Simulation Software*), pertencente ao programa SciDAC (*Scientific Discovery through Advanced Computing*), partiram para a especificação do *Common Component Architecture* (CCA) [1], uma especificação aberta de modelo de componentes para CAD. Inicialmente proposto em 1998, a especificação define o comportamento e as formas de interação entre os componentes [7]. A implementação concreta fica sob a responsabilidade das plataformas, as quais

são chamadas de *frameworks*, atualmente representados por Ccaffeine, MOCCA, XCAT, DCA, SCIRun2, entre outros.

A versão inicial do CCA tinha dois principais objetivos [4]:

- uma especificação de núcleo, que permitisse a publicação de interface de componentes e a possível junção deles;
- uma especificação de uma interface padrão de comunicação dos componentes entre si e entre a plataforma CCA.

O Fórum CCA, como passou a ser chamado o grupo de pesquisadores que definiam essa tecnologia, desejava criar uma especificação básica, de forma que *softwares* previamente desenvolvidos fossem facilmente transformados em componentes CCA [6]. A especificação é leve e simples de usar, de modo que a simples implementação de alguns métodos torna aplicações CAD monolíticas em componentes CCA [4].

Dois conceitos são fundamentais para compreender a definição do modelo, que corresponde a padronização de uma linguagem para definição de interfaces dos componentes e o mecanismo de relacionamento entre estes, conhecido como portas *provides/uses*.

Os componentes exportam as suas interfaces que oferecem serviço (portas *provides*) e obtém acesso a interfaces de outros componentes (portas *uses*). Na prática, essas conexões são feitas através da chamada de métodos, sendo a *plataforma* responsável pelo seu estabelecimento.

A existência de conectores como elementos de primeira classe também não faz parte da especificação do CCA. Em princípio, o relacionamento entre componentes é realizado formalmente apenas pelas portas *uses* e portas *provides*, porém as plataformas podem implementar esse conceito [5] [23].

Os componentes têm suas interfaces definidas através de uma linguagem padrão chamada *Scientific Interface Definition Language* (SIDL). É a partir da especificação dessa interface que a plataforma obterá as informações necessárias ao carregamento e estabelecimento de conexões entre os componentes escritos em várias linguagens.

3.2.1 Plataformas CCA

A especificação CCA abrange apenas a definição de interfaces interoperáveis entre componentes e a plataforma. Em princípio, espera-se que os componentes estejam previamente escritos, e que as suas interfaces exportadas para a plataforma possam

ser conectadas perfeitamente. Não é papel do CCA (pelo menos, não está em sua especificação) definir conectores exógenos que possam orquestrar as ações dos outros componentes. Em outras palavras, o CCA não possui nativamente uma linguagem de descrição de arquitetura. Da mesma forma, apesar de ser uma preocupação do Fórum CCA, não há definições dos métodos de reconfiguração dinâmica [6].

Conforme dito anteriormente, as lacunas existentes na especificação permitem que as implementações definam, segundo seus objetivos, a melhor formar de tratar cada um dos problemas. A seguir, apresentamos informações a cerca de algumas plataformas no tocante a possibilidade de escrita de conectores exógenos, a existência de linguagens de descrição de arquitetura e o suporte à reconfiguração dinâmica.

3.2.2 CCAFFEINE

Ccaffeine [4] é uma implementação do modelo CCA focada no atendimento dos seguintes requisitos:

- Criar uma plataforma e modelo de programação que suporte o estilo SPMD (*Single Program Multiple Data*) de programação paralela;
- atender os serviços da plataforma via outros componentes.

O estilo SPMD permite que o programador escreva um único programa paralelo, separando as responsabilidades de cada processo, alocado a uma unidade de processamento possivelmente distinta uns dos outros, através de um identificador representado por um número inteiro, através do qual também refereciam-se em operações de troca de mensagens. Certamente é o padrão mais largamente utilizado em CAD, o que motiva o desejo de transformar de modo simples programas existentes em componentes CCA.

O Ccaffeine assume que a plataforma executará sobre uma arquitetura de memória distribuída que suporte comunicação por passagem de mensagens. Cada componente SPMD desenvolvido terá instâncias idênticas em cada nó de processamento, incluindo as conexões das portas, formando *regimentos de componentes*[1]. Essas suposições permitem que as conexões entre componentes CCA sejam feitas localmente. A esse estilo, dá-se o nome de SCMD (do inglês *single component, multiple data*), em alusão ao estilo SPMD no qual é inspirado.

O Ccaffeine não possui uma linguagem de descrição de arquitetura. Como preconiza o padrão CCA, a plataforma apenas oferece uma simples lista de

[1] do inglês, *cohort*.

comandos para compor, iniciar e acompanhar a execução de programas. Entretanto, a plataforma permite encapsular um conjunto de componentes dentro de um pseudo-componente único [3]. Essa forma de especificação facilita a reutilização de blocos de componentes.

Todo o funcionamento comportamental da execução está inserida nos componentes. Dessa forma, até onde pudemos identificar, a plataforma não oferece qualquer tipo de ferramenta que possibilite a escrita de conectores exógenos.

Através da criação de configurações por sua linguagem de linha de comando, é possível realizar substituição dinâmica de componentes [3].

3.2.3 MOCCA

MOCCA é uma plataforma CCA com foco em grades computacionais, implementada sobre o sistema de metacomputação H2O [38], uma plataforma de compartilhamento de recursos computacionais. Metacomputação é a gestão e execução de aplicações que exploram vários níveis de distribuição: em memória compartilhada, em nós de *cluster*, e ainda em um *grid*.

Identificamos duas linguagens de descrição de arquitetura para a plataforma MOCCA. A primeira, chamada Moccaccino [47] é capaz de apenas definir a estrutura da aplicação. Sua função é listar os componentes CCA, ligando as suas portas *services* e *provides*. Através dessa ADL não é possível especificar comportamento, consequentemente a possibilidade de escrita de conectores exógenos torna-se similar as apresentadas no Ccaffeine.

Quanto a reconfigurações em tempo de execução, é possível realizar alterações estruturais dinamicamente.

A segunda linguagem ADL é chamada GScript [22], a qual é baseada em Ruby [2]. Por trata-se de uma linguagem de programação de propósito geral, Ruby, e consequentemente GScript, permite a especificação de comportamentos, de forma que a elaboração de conectores exógenos passa a ser viável[2]. Por trata-se de uma linguagem de *script*, é possível alterar o comportamento descrito em tempo de execução e assim realizar reconfigurações dinâmica.

3.3 Fractal

O Fractal [14] é um modelo de componentes caracterizado pela capacidade de definir componentes compostos utilizando a composição hierárquica.

[2]No artigo que apresenta a linguagem [22], os conectores exógenos são chamados de *composição no tempo* (do inglês *composition in time*)

Segundo a especificação do modelo, um componente é definido por uma *membrana* e um *conteúdo*. O conteúdo corresponde ao conjunto de componentes internos, também chamados de sub-componentes. A membrana delimita as interfaces desse componente com o mundo externo e configura internamente os sub-componentes fazendo uso de controladores e interfaces internas.

É possível que um componente tenha o conteúdo vazio, ou seja, não possua sub-componentes em sua estrutura. Nesses casos, eles são chamados de *componentes primitivos*.

Além do uso da composição, os componentes Fractal podem se relacionar através de *ligações* (*bindings*). Essas ligações devem ser estabelecidas entre portas cliente (demandante) e servidora (demandada), através de *ligações primitivas* ou *compostas*. As ligações primitivas possibilitam a conexão de apenas uma porta cliente com uma porta servidora, enquanto que as ligações compostas permitem a conexão de um número qualquer de portas, estabelecendo assim, comunicações coletivas.

O Fractal define as ligações compostas como componentes. Assim como já defendido anteriormente [33], o que o classifica como ligação é o seu papel de mediar a comunicação entre outros componentes.

Considerando as duas opções de interação de componentes oferecida pelo Fractal, é possível definir conectores endógenos e exógenos. No primeiro caso, faz-se uso das ligações enquanto que, no segundo caso, pode-se utilizar a composição.

Boa parte da especificação Fractal é opcional. Por um lado, essa característica traz muita resiliência, de modo que é possível realizar implementações que atendam apenas aos requisitos de determinado propósito. Entretanto, esta flexibilidade pode causar problemas de compatibilidade entre diferentes implementações.

Para tratar essa questão, foram definidos *níveis de conformidade*, de forma a indicar principalmente quais as interfaces disponíveis nos componentes implementados naquela plataforma. Abaixo apresentamos cada um dos níveis atualmente difinidos pela especificação [15].

Nível 0: neste nível, nada é obrigatório. Podemos dizer assim que qualquer modelo de componentes é uma implementação Fractal nível 0.

Nível 1: a partir desde nível, todos os componentes devem prover pelo menos a interface Component.

Nível 2: a implementação deve possuir a mesma exigência do nível anterior, além de requerer que todos os componentes possam ser convertidos para a interface Interface.

Nível 3: a implementação deve possuir as mesmas exigências do nível anterior, além de requerer que todos os componentes sejam especificados de acordo com o sistema de tipos padrão do Fractal.

Além desses três níveis básicos, cada um deles possui ainda algumas variações.

O nível 0.1 é uma variação onde, além de possuir as mesmas exigências do nível 0, deve garantir que os componentes com atributos configuráveis, com interfaces clientes, que expõe seu conteúdo e que expõe seu ciclo de vida devem respectivamente implementar interfaces específicas para cada fim. Similarmente, o nível 1.1 deve garantir as propriedades do nível 1, somados às exigência do nível 0.1. O mesmo ocorre analogamente para os níveis 2.1 e 3.1.

No tocante ao nível 3, há ainda outros duas variações. No nível 3.2, além das exigências do nível 3.1, componentes de inicialização (*bootstrap component*) devem implementar interfaces que permitam o uso de uma fábrica de componentes, enquanto que no nível 3.3, além das exigências do nível 3.2, a fábrica de componente deve ser capaz de criar gabaritos (*templates*) de componentes primitivos e compostos.

O modelo Fractal, desde a sua concepção, prevê o uso de reconfiguração dinâmica para realizar ajustes em suas aplicações [14]. Além disso, trabalhos específicos foram realizados a fim de certificar que os mecanismos utilizados satisfazem requisitos de consistência e segurança [41].

3.4 GCM (*Grid Component Model*)

O GCM é um modelo de componentes com foco em grades computacionais (*grid computing*), baseado no Fractal [12]. O seu objetivo é prover expressividade na construções de aplicações para ambientes heterogêneos, geograficamente distribuídos e dinâmicos.

Para os idealizadores do projeto, as aplicações em grade computacional são construídas amparadas em duas técnicas:

- uso de comunicações explícitas, através de passagem de mensagens ou chamadas remotas de procedimentos;
- uso de linguagens de especificação de fluxos de trabalho (*workflows*) para orquestração exógena de entidades distribuídas de processamento.

A proposta do GCM é unir essas duas técnicas sob um modelo de componentes, o qual seja adequado para expressar comunicações explícitas, juntamente com um gerenciamento de alto nível das entidades de processamento.

Conforme descrito, o GCM tem como base o Fractal, o que lhe faz herdar o mecanismo de definir componentes por composição e as capacidades de introspecção e de reconfiguração. Essas características irão possibilitar, respectivamente, a hierarquização dos componentes, a possibilidade de monitoramento e controle em tempo de execução e, finalmente, a reconfiguração dinâmica.

De modo geral, podemos afirmar que as formas de conexão dos componentes no GCM são similares ao Fractal. Entretanto, considerando o seu foco em um modelo geograficamente distribuído, o GCM propõe abordagens próprias para comunicações coletivas. Interfaces coletivas podem ser de três grupos: interfaces 1 para N (*multicast*), onde uma porta cliente é conectada a N portas servidoras, interfaces N para 1 (*gathercast*), onde N portas clientes são conectadas a uma porta servidora e por fim, interfaces M para N, onde múltiplas portas são interligadas nas duas extremidades.

O objetivo principal para definição das interfaces coletivas é permitir a simplificação em seu uso, fazendo com que, por exemplo, uma aplicação realize uma única chamada a uma interface *multicast*, a qual, por sua vez, as transforma nas várias chamadas destino, como também permitir que essas chamadas possam ocorrer em paralelo [12]. O mesmo ocorre simetricamente para o *gathercast*.

No caso específico da interface MxN, ao invés do uso de uma solução ingênua de ligação entre uma interface *multicast* com uma *gathercast*, o que geraria um gargalo, o GCM propõe o uso de M interfaces *multicast* e N interfaces gathercast, possibilitando assim a completa ligação entre todos os componentes participantes da comunicação coletiva [12].

Similarmente ao Fractal, os componentes GCM são formados por uma membrana, que define a composição interna e os aspectos não-funcionais e seu conteúdo responsável pelo funcional. Através da membrana, o GCM oferece a possibilidade de modificação dos componentes. Essas alterações podem ser realizadas tanto no conteúdo do componente e sua formação hierárquica, quanto na estrutura e nos aspectos não-funcionais da própria membrana, via pelo qual é possível a realização de reconfigurações em tempo de execução [12].

3.5 O Modelo de Componentes Hash

O modelo de componentes Hash junta-se aos outros modelos de componentes voltados a aplicações de CAD com uma proposta adicional: conceder aos componentes uma natureza intrinsecamente paralela [20].

Na programação paralela, os interesses de uma aplicação devem estar implementados de forma distribuída entre os processos que formam o programa paralelo. Porém, uma vez que a programação paralela está concentrada na decomposição do programa em processos [19], ao invés dos interesses, há a tendência natural de tratar processos como unidades dominantes da decomposição do *software*, a qual julgamos errônea. Na prática, em uma plataforma de componentes, essa limitação implica que uma instância de um componente realizará a sua execução apenas em uma única unidade de processamento, encapsulada em um processo. Assim, um interesse que deveria estar encapsulado em uma única unidade de decomposição do programa, como convém a boa prática de modularização de programas na engenharia de *software*, estaria espalhado em um conjunto de componentes responsáveis por implementar o interesse.

Esse problema é amenizado quando o paralelismo da plataforma de componentes é implementado por meio do paradigma SCMD (do inglês *single component, multiple data*), onde há a necessidade de um único componente que é instanciado em um conjunto de unidades de processamento, pois o interesse estaria encapsulado nesse componente. Porém, em algumas plataformas de componentes, como o *framework* CCAffeine, as instâncias de componentes comunicam-se na execução paralela por meio de passagem de mensagens usando alguma biblioteca padrão, como o MPI, quando deveriam comunicar-se somente através de suas interfaces regulares (portas *uses* e *provides*), conforme suposto pelo modelo de componentes. Nesse caso, há uma quebra do encapsulamento do componente tendo em vista a existência de *comunicações clandestinas* entre os componentes, as quais ocorrem sem que a plataforma esteja ciente. Para garantir a regularidade das comunicações, seria recomendável, do nosso ponto de vista, introduzir no modelo novos tipos de portas entre os componentes, que permitam a comunicação par-a-par entre um conjunto de instâncias de componentes paralelos que desejam realizar uma computação paralela. Porém, isso não resolveria o problema do encapsulamento do interesse, caso tais instâncias sejam de componentes distintos, o que seria necessário para o padrões MPMD (do inglês *multiple program, multiple data*) de interação entre processos.

A proposta do modelo Hash é eliminar essa limitação, possibilitando que os interesses sejam encapsulados em uma unidade de programa a despeito da sua implementação em várias unidades de processamento distribuídas e suportando programação MPMD. O modelo parte do pressuposto de que processos e interesses são conceitos ortogonais [19]. Assim, um componente do modelo Hash, também

conhecido como *componente #*, pode conter partes que executam em unidades de processamento distintas. A essas partes, atribuímos o nome de *unidades*.

Um sistema de programação baseado em composição hierárquica de componentes é compatível com o modelo Hash se possui as seguintes características:

- componentes são formados por partes, chamadas *unidades*, que podem ser multiplamente instanciadas em unidades de processamento distintas de um computador paralelo de memória distribuída;
- componentes podem ser *compostos* hierarquicamente *por sobreposição*, gerando novos componentes;
- suporta um conjunto de *espécies de componentes*, de tal forma que cada componente pertence a uma espécie;

Portanto, um único componente # executa distribuidamente em várias unidades de processamento, através de suas unidades. Na instanciação de um componente #, várias instâncias de uma unidade podem existir localizadas em unidades distintas de processamento. São essas unidades que tornam o componente intrinsecamente paralelo. É responsabilidade da plataforma oferecer uma visão consistente do componente #, o qual deve ser visto de forma indivisível.

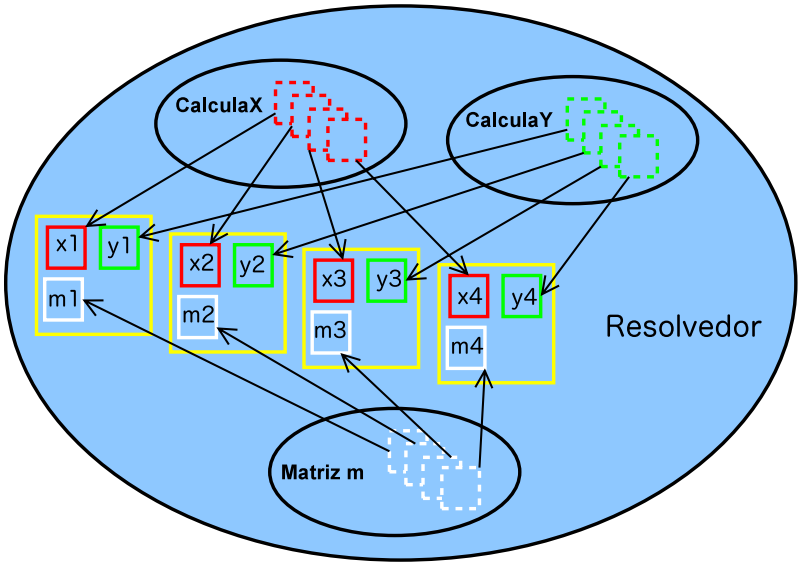
3.5.1 Sobreposição de Componentes

A sobreposição [19] concede ao modelo a capacidade de criação de componentes a partir de outros componentes de forma hierárquica, similar ao que se tem no modelo Fractal, porém adaptado às necessidades de um componente intrinsecamente paralelo.

Para melhor compreensão desse conceito, juntamente com os conceitos de *unidade*, *componente aninhado*, *fatia* e *ação*, iremos fazer uso de um exemplo. Suponhamos que se deseje escrever uma aplicação paralela para executar sobre um *cluster* de 4 nós. O processamento será realizado através da coordenação de dois componentes, os quais aplicam seus algoritmos sobre uma matriz. Assim, nessa solução, teremos quatro componentes envolvidos: os três já citados (a estrutura de dados matriz também é um componente) e um novo componente descrito como uma composição dos anteriores que tem o papel de coordenar a execução. A Figura 3.1 apresenta um diagrama, onde as elipses representam os componentes #. Componentes inscritos a outros representam *componentes aninhados*. Por exemplo, os componentes CalculaX, CalculaY e Matriz são componentes aninhados de

Resolvedor. Como implicação, Resolvedor é um componente definido por *sobreposição* de CalculaX, CalculaY e Matriz. Os retângulos representam unidades. Portanto, CalculaX possui 4 unidades, simbolizadas pelos retângulos no interior da elipse que o representa.

Figura 3.1: Representação de um componente #



Fonte: Próprio autor.

As unidades também podem ser compostas pelas unidades de componentes aninhados. Nessa relação, representada no diagrama por uma seta, as unidades constituintes são chamadas de *fatias* da unidade composta. No nosso exemplo, a unidade x1 do componente CalculaX, a unidade y1 do componente CalculaY e ainda a unidade m1 do componente Matriz são fatias de uma das unidades do componente Resolvedor. Como pode ser visto, esta mesma relação é mantida entre as demais unidades dos componentes.

A unidade é a menor parte de decomposição de um componente #. Assim, supondo a execução deste componente no citado *cluster* de 4 nós, cada uma das unidades x_n será implantada em um nó distinto da máquina, apesar de estarem presentes no mesmo componente. Este é conceito de *componente paralelo*: um único componente, executando distribuidamente em vários nós. Não se trata de um cópia do componente, mas sim fragmentos que, inclusive, podem ser diferentes. Por exemplo, as computações presentes em x1 podem ser distintas das definidas pelos algoritmos da unidade x2.

Conforme mencionamos, componentes que definem computações descrevem seus

algoritmos em suas unidades através de *ações*. Fazendo uma analogia com a programação orientada a objetos, ações são similares a métodos. Ao se definir uma nova ação em uma unidade, pode-se fazer uso das ações de suas fatias. Por exemplo, as unidades do componente Resolvedor podem fazer uso das ações definidas nas suas fatias x_n e y_n. Na prática, a unidade e suas fatias serão implantadas no mesmo nó de processamento.

Unidades também podem definir *condições*, cujo papel é descrever computações que forneçam um resultado booleano. Essas computações servem para simples verificação de estados da aplicação, não devendo ser usadas para processamento útil.

3.5.2 Espécies de Componentes

As espécies classificam os componentes quanto ao seu papel na aplicação, permitindo que a plataforma os trate de forma diferente, adaptando-se a diferentes modelos de conexão e implantação, bem como modelos de ciclo de vida, que sejam necessários dentro de um domínio de aplicação.

Nas plataformas de componentes tradicionais, define-se uma única espécie de componentes e um conjunto fixo de conectores entre estes, de forma que quando novas abstrações são necessárias para lidar com necessidades de novas aplicações dentro do domínio, é preciso estender o modelo. Em uma plataforma do modelo *Hash*, uma vez que componentes # podem estar associados a interesses funcionais ou não-funcionais, bem como concretos ou abstratos, basta introduzir-se na plataforma uma nova espécie de componente para tratar aquela nova abstração, bem como definir as regras da relação dos seus componentes com componentes das espécies já existentes.

Espécies de componentes podem também definir uma linguagem de composição de componentes dentro de um domínio, oferecendo suporte a desenvolvimento de aplicações pela combinação de partes, os componentes #, que representam abstrações dentro de domínios específicos.

Finalmente, como componentes # são intrinsicamente paralelos, podem representar conectores entre componentes paralelos. Dessa forma, uma plataforma compatível com o modelo Hash pode conter um conjunto evolutivo de conectores, de acordo com as necessidades dos computadores paralelos da tecnologia vigente.

3.6 A Plataforma HPE (*Hash Programming Environment*)

O HPE (*Hash Programming Environment*) é uma plataforma de componentes paralelos compatível com o modelo HASH, voltado a programação paralela de propósito geral para *clusters*. Além disso, o HPE é compatível com o modelo CCA [17], constituindo um *framework* com suporte ao paralelismo distribuído, assim como o *framework* DCA [13], porém suportando um conceito mais geral de componente MPMD devido ao uso de componentes #.

3.6.1 Espécies de Componentes Suportadas no HPE

As espécies suportadas pelo HPE são as seguintes: *plataforma*, *ambiente*, *topologia*, *estrutura de dados*, *qualificador*, *sincronizador*, *computação* e *aplicação* [20].

Componentes da espécie *plataforma* descrevem a arquitetura da máquina física onde os componentes # serão executados. Nesta espécie, cada unidade simboliza um nó de processamento. Assim, ao desenvolver um componente #, dentre as informações fornecidas, o programador pode indentificar qual a plataforma para o qual ele foi projetado incluindo um parâmentro de contexto desta espécie. O conceito de parâmentro de contexto será definido adiante.

Componentes da espécie *ambiente* descrevem as camadas de *software* que possibilitam o suporte ao paralelismo, como as bibliotecas de passagem de mensagem (MPI ou PVM). Junto com a anterior, essa espécie descreve completamente o contexto de execução paralela.

Componentes da espécie *topologia* definem como as unidades de um componente são mapeadas aos nós de processamento de uma plataforma de computação paralela sobre os quais estão sendo instanciados.

Componentes da espécie *qualificador* são responsáveis por caracterizar e classificar outros componentes segundo suas propriedades não-funcionais relevantes. O papel dessa espécie ficará mais claro adiante, na Seção 3.6.4, onde será apresentado o sistema de tipos HTS, voltado às necessidades de plataformas de componentes do modelo HASH.

Componentes da espécie *computação* realizam computações paralelas. O paralelismo é caracterizado pelo papel de cada uma das suas unidades. Conforme explicado na seção anterior, cada unidade é implantada num nó específico da máquina paralela.

Componentes da espécie *sincronizador* definem mecanismos de interação entre

outros componentes. Eles devem encapsular comunicação, conversão ou facilitação para a ligação de unidades no escopo de um componente #. Em outras palavras, sincronizadores atuam como conectores. Há sincronizadores nativos da plataforma HPE para alguns padrões de comunicação como passagem de mensagem orientada a canal, comunicações coletivas e invocações remotas de serviço (RPC/RMI). Porém, o modelo permite que o desenvolvedor crie seu próprio sincronizador.

Componentes da espécie *estrutura de dados* armazenam as dados processados por componentes # da espécie *computação* e *sincronizador*. Por esse motivo, normalmente são usados como componentes aninhados de componentes dessas espécies, bem como armazenam estado durante a execução, não sendo segura sua participação na maioria das operações de reconfiguração dinâmica.

Componentes da espécie *aplicação* são como componentes *computação*, porém possuindo algumas retrições. Componentes *aplicação* devem obrigatoriamente contar com parâmetros dos tipos *ambiente* e *plataforma*, de modo a definir o contexto de execução. Tais componentes também não permitem o acesso a seus componentes aninhados. Componentes dessa espécie atuam como o gatilho para iniciar a execução de um programa paralelo sobre o *middleware* de comunicação, como MPI, por exemplo. Para isso, são os componentes que oferecem a porta `GoPort`, do padrão CCA.

As unidades de componentes das espécies *aplicação*, *computação* e *sincronizador* estão associadas a ações computacionais. Por isso, cada unidade exporta um conjunto de *ações* amparados em um *protocolo*, o qual define uma restrição sobre a ordem em que essas ações podem ser executadas.

3.6.2 Arquitetura Geral

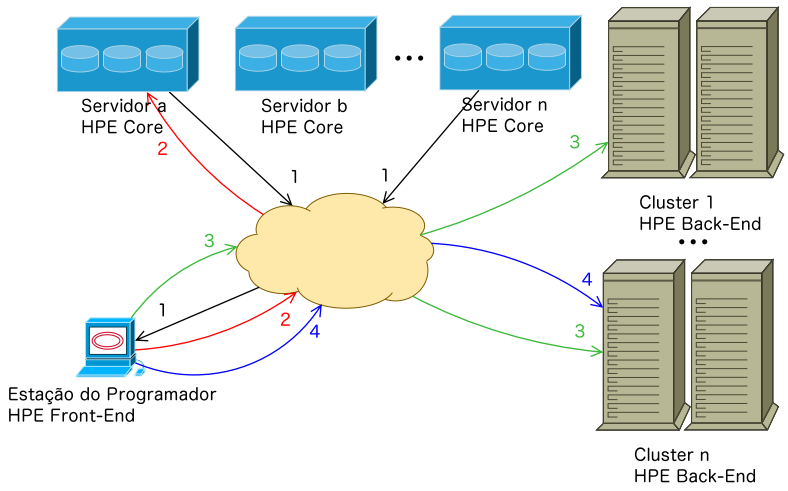
A plataforma HPE é composta por três serviços implantados em servidores possivelmente distribuídos geograficamente, denominados *Front-End*, *Core* e *Back-End*. Para isso, são atualmente conectados através de *Web Services*.

O *Front-End* é o ambiente de desenvolvimento e gerenciamento do ciclo de vida de componentes, através do qual os componentes têm sua arquitetura e configuração descritos por meio da linguagem HCL (Hash *Configuration Language*).

O *Core* representa um repositório distribuído de componentes, implementado através de um conjunto de servidores chamados de *Locations*. Para isso, oferece serviços ao *Front-End* e ao *Back-End*, para armazenamento e recuperação de configurações de componentes em tempo de desenvolvimento.

O *Back-End* é uma plataforma de componentes propriamente dita, um *framework* CCA. Cada instância de um *Back-End* é responsável por gerenciar a implantação, a instanciação e a execução de componentes # em um *cluster* particular. Para isso, também oferece serviços que são consumidos pelo *Front-End*. Além disso, consome um serviço do *Core* destinado a recuperação de componentes.

Figura 3.2: Desenvolvimento, implantação e execução de uma aplicação.



Fonte: Próprio autor.

A Figura 3.2 apresenta um típico cenário de desenvolvimento, implantação e execução de componentes na plataforma HPE. Em princípio, um desenvolvedor utilizando o *Front-End* em sua estação de trabalho desenvolve um componente #. Esse novo componente potencialmente fará uso de outros componentes já existentes, obtidos através de uma ou mais bibliotecas (*Locations*) por meio dos serviços do *Core* (passo 1). Em seguida, após o desenvolvimento dos componentes, o programador poderá também disponibilizá-lo na biblioteca, para seu usufruto e de outros usuários (passo 2). O programador pode então implantar o componente # em um computador paralelo através do serviço do *Back-End* oferecido pela plataforma através de um *Web Service* (passo 3). Componentes # podem então ser instanciados e conectados uns aos outros por meio do casamento de suas portas *uses* e *provides* através do serviço BuilderService oferecido pelo *Back-End*. Esse serviço é exigido pelo padrão CCA, ao qual o HPE adere. Caso o componente instanciado seja da espécie *aplicação*, é ainda possível disparar sua execução chamando-se o método go da sua porta GoPort, também do padrão CCA (passo 4). Opcionalmente, um usuário pode instanciar uma aplicação diretamente, através

do método run do serviço RunService do *Back-End*. Nesse caso, todos os componentes transitivamente aninhados da aplicação são instanciados de forma automática, formando a hierarquia de componentes da aplicação. Isso é possível devido ao uso do HTS (*Hash Type System*), um sistema de tipos de componentes desenvolvido para o HPE que permite especificar os tipos dos componentes aninhados de forma suficientemente expressiva para permitir sua instanciação automática, escolhendo-se um componente # adequado que implemente um interesse especificado para um determinado contexto de execução.

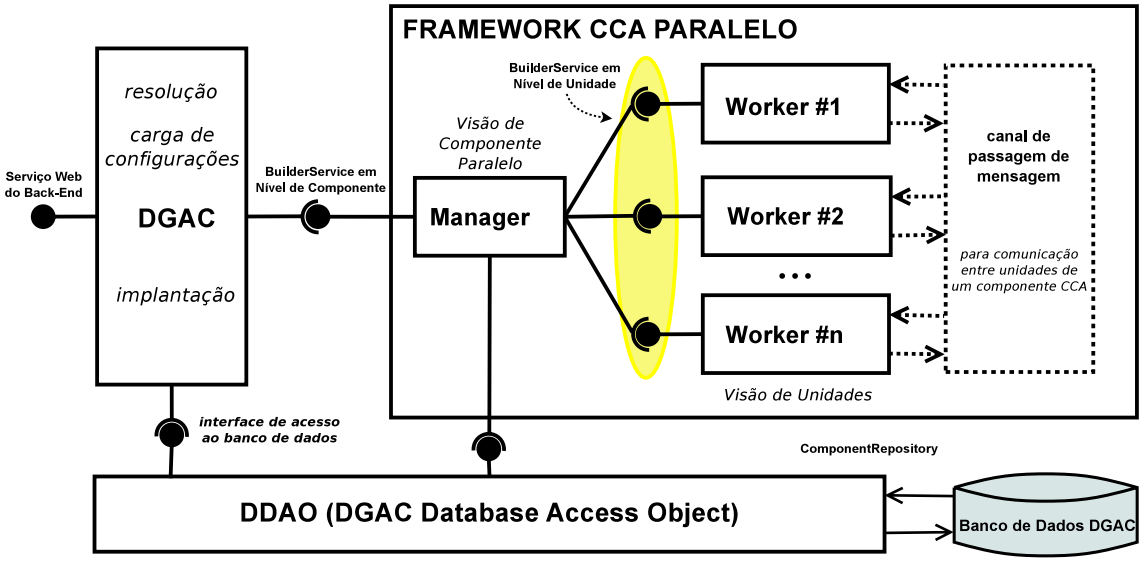
3.6.3 Arquitetura do *Back-End*

O *Back-End*, o qual merece atenção especial por ser a plataforma de componentes propriamente dita do HPE, está implementado sobre a plataforma de componentes Mono, do padrão CLI (*Common Language Infrastructure*), mais conhecido por ser o padrão implementado pela plataforma de componentes .NET da Microsoft, a qual é utilizada no sistema operacional Windows. Cada componente # é armazenado em um arquivo objeto chamado *assembly*, mais conhecido pela sua extensão "dll", os quais constituem os componentes propriamente ditos de plataformas CLI. Esses *assemblies* estão acessíveis a partir das unidades de processamento do *cluster* na implementação atual do *Back-End*, ainda exigindo a existência de um sistema de arquivos compartilhado entre as unidades de processamento.

Mono é uma plataforma de execução virtual de código aberto e, por suportar CLI, oferece funcionalidades importantes para CAD, a maioria das quais não suportadas por outras plataformas de execução virtual como Java, tais como suporte a *arrays* multidimensionais, suporte a múltiplas linguagens, controle de versão de componentes e compilação AOT (*Ahead-of-Time*) e JIT (*Just-In-Time*). O padrão CLI oferece ainda uma linguagem intermediária, chamada IL (*Intermediate Language*) que suporta um sistema de tipos polimórfico, chamado CTS (*Common Type System*), o qual oferece funcionalidades essenciais para implementação do sistema de tipos de componentes HTS.

A execução do *Back-End* implementa a orquestração de um objeto *Manager*, o qual executa na unidade de administração do *cluster*, e vários objetos *Worker*, executando em cada uma de suas unidades de processamento. A Figura 3.3 ilustra a relação entre o objeto *Manager* e os objetos *Worker* dentro do *Back-End*, onde surge o objeto DGAC (*Distributed Global Assembly Cache*) intermediando as requisições do *Web Service*.

Figura 3.3: Arquitetura do Back-End



Fonte: [17]

O objeto DGAC é responsável pelo registro, compilação e implantação dos componentes na plataforma. O registro corresponde a persistência das meta-informações que descrevem a arquitetura e configuração do componente, atualmente implementado por meio do gerenciador de banco de dados *MySQL Cluster*, representado pelo objeto DDAO (*DGAC Database Access Object*). A compilação e a implantação são os processos necessários para a disponibilização dos componentes #, gerando seus *assemblies* e os instalando no sistema de arquivos compartilhado que pode ser acessado pelos objetos *Worker*,

Os objetos *Manager* e *Worker* formam um *framework* CCA paralelo, capaz de gerenciar a instanciação de componentes # de qualquer espécie e a execução de componentes # da espécie aplicação, por meio da interface BuilderService. De uma maneira simples, tanto o objeto *Manager* quanto os objetos *Worker* implementam a interface BuilderService. Portanto, pode-se enxergar cada objeto *Worker* individual como um *framework* CCA que gerencia o ciclo de vida de unidades de componentes # através dele instanciados, vistos como componentes CCA usuais, isto é, instanciados em memória compartilhada. Então, o papel do objeto *Manager* é orquestrar o conjunto de objetos *Worker* onde estão executando as unidades de um componente # a fim de oferecer uma visão única e consistente do mesmo.

Uma vez que os componentes estejam implantados, ao solicitar a execução de um componente # da espécie aplicação, o *Manager* dispara a instanciação e execução de

suas unidades em cada objeto *Worker* e aguarda a sua finalização, disponibilizando o resultado final para o usuário.

Na execução de uma aplicação por meio do serviço RunService, os objetos *Worker*, onde as unidades de um componente # encontram-se instanciadas, decidem simultaneamente, porém de forma independente, isto é, sem comunicação ou sincronização, quais componentes # devem ser instanciados para serem ligados aos componentes transitivamente aninhados do componente # instanciado, e então instanciam as unidades que lhe dizem respeito. Pela interpretação de que um componente aninhado pode ser visto como uma porta *uses* de um componente CCA, isso equivale a instanciar um componente CCA cuja porta *provides* será ligada a porta *uses* que representa o componente aninhado [17]. Para isso, os componentes aninhados e os componentes # são tipados através de componentes abstratos, os quais especificam um interesse que deve ser implementado pelo #-componente, e um contexto de execução. Então é responsabilidade do *Back-End* buscar, dentre os componentes # implantados na plataforma, aquele cujo tipo que é o que mais se adequa ao tipo do componente aninhado, considerando o interesse especificado pelo componente abstrato e o contexto. Essa busca é implementada por meio de uma *algoritmo de resolução*, capaz de generalizar o contexto do componente aninhado até que seja escolhido um componente # que possa ser usado no contexto especificado.

3.6.4 Sistema de Tipos de Componentes

O HTS (*Hash Type System*) é um sistema de tipos de componentes adotado pelo HPE que distingue *componentes abstratos* e *componentes concretos*. Os últimos caracterizam os componentes # propriamente ditos, que podem ser instanciados, enquanto os primeiros caracterizam os seus tipos, especificando o interesse implementados por componentes concretos. Portanto, um componente concreto deve implementar os elementos constituintes (componentes aninhados e unidades) exigidos por um certo componente abstrato. Componentes abstratos especificam ainda um conjunto de *parâmetros de contexto* formais, caracterizados por: um nome identificador, para referenciá-lo; um nome de variável, para ligação ao valor de outros parâmetros de contexto; e um limite, representado por um *tipo instanciação*. Um tipo instanciação é definido recursivamente por um componente abstrato e um conjunto de tipos instanciação associados aos parâmetros de contexto formais, do componente abstrato, através de seus nomes identificadores, chamados parâmetros de contexto efetivos.

Tipos instanciação são usados para descrever os tipos de componentes aninhados e componentes concretos. O tipo instanciação de um componente aninhado estabelece o interesse que deve ser implementado pelo componente # que será a ele ligado, bem como um contexto que descreve critérios para que o componente mais apropriado seja escolhido dentre os que implementam o interesse. Isso é bastante relevante em computação paralela, tendo em vista que computações paralelas ou padrões de comunicação entre processos podem ser implementados conforme algoritmos diferentes de acordo com as peculiaridades do computador paralelo alvo ou da aplicação em si. O tipo de instanciação de um componente # define portanto o interesse implementado por ele, bem como para qual contexto ele deve ser aplicado. Através dessas informações, um algoritmo de resolução implementado pelo *Back-End* é capaz de decidir o componente # mais adequado a ser ligado a um componente aninhado de um outro componente #, conforme critérios computacionais, e portanto objetivos, definidos pelo HTS.

Uma importante característica do HTS é a existência de uma relação de subtipos entre tipos instanciação, de tal forma que é possível generalizar o contexto de um tipo instanciação a fim de encontrar uma versão mais genérica de um componente # que possa ser ligada a um componente aninhado de forma segura, ou seja, um componente # cujo tipo instanciação seja um subtipo do tipo requisitado pelo componente aninhado.

Exemplo

Apresentamos um exemplo de definição de um componente abstrato, o LinearSystemSolver, que representa os resolvedores paralelos de sistemas lineares algébricos. O componente possui cinco *parâmetros formais de contexto* que especializam as implementações quanto a arquitetura paralela alvo e quanto ao padrão e representação da matriz. A Figura 3.4 apresenta a assinatura do componente abstrato em questão.

Os parâmetros identificados pelos nomes **accelerator**, **multicore**, **matrix_property**, **matrix_partition**, e **matrix_type** são responsáveis por definir o contexto sob o qual um componente # é implementado. Assim, pode-se utilizar estas suposições para implementar o interesse tratado pelo componente abstrato LinearSystemSolver de forma mais eficiente. Seus valores são restringidos respectivamente pelos componentes abstratos AcceleratorType, MultiCoreSupport, MatrixProperty, MatrixPartition e MatrixType,

Figura 3.4: Assinatura de contexto do componente abstrato LinearSystemSolver.

```
LinearSystemSolver
    [accelerator = A : AcceleratorType,
     multicore = M : MultiCoreSupport,
     matrix_property = P : MatrixProperty,
     matrix_partition = R : MatrixPartition[multicore = M],
     matrix_type = T : MatrixType[property = P,  partition = R]
```

Fonte: Próprio autor.

Figura 3.5: Assinatura de contexto atendido pelo componente concreto SparseLinearSystemSolver.

```
LinearSystemSolver
    [accelerator = CUDABasedGPU,
     multicore = MultiplesCore,
     matrix_property = SparseMatrix,
     matrix_partition = R : MatrixPartition[multicore = MultiplesCore],
     matrix_type = CSRMatrixFormat[property = BlockTridiagonal,  partition = MatrixPartition]
```

Fonte: Próprio autor.

os quais constituem *limites* superiores dos parâmetros de contexto. Por sua vez, as *variáveis de contexto* A, M, P, R e T, respectivamente associadas aos parâmetros de contexto, servem para referenciar o valor de uma variável em uma assinatura, permitindo ligar os valores associados a dois parâmetros de contexto.

Com exceção de MatrixType, cuja espécie é *estrutura de dados*, todos os outros limites são da espécie *qualificador*. Esta é a maneira usado no HPE para especificar requisitos não-funcionais.

Os componentes abstratos que definem os limites também podem possuir parâmetros de contexto. O componente abstrato MatrixPartition possui o parâmetro de contexto **multicore**, indicando que a estratégia de particionamento da matriz entre suas unidades depende do nível de suporte a processadores de múltiplos núcleos. De forma similar, os parâmetros de contexto do componente abstrato MatrixType, **property** e **partition**, o especializam de acordo com as propriedades da matriz e sua estratégia de particionamento, respectivamente.

Os componentes concretos, ao implementar componentes abstratos, definem o contexto para o qual foram desenvolvidos, tendo isso otimizados para esse. Por exemplo, suponhamos a implementação do componente concreto SparseLinearSystemSolver que atenda ao tipo definido na Figura 3.5.

Nessa implementação, admitimos que as unidades do componente concreto SparseLinearSystemSolver fazem uso de GPUs e processadores de múltiplos

Figura 3.6: Assinatura de contexto de componente aninhado com variáveis livres.

```
LinearSystemSolver
    [matrix_property = BlockTridiagonal,
     matrix_type = CSRMatrixFormat[property = BlockTridiagonal]
```

Fonte: Próprio autor.

núcleos para a resolução dos sistemas lineares. Além disso, o componente é especificamente desenvolvido para tratar de matrizes esparsas armazenadas no formado de *linhas esparsas comprimidas*, CSR (do inglês *compressed sparse row*). Considerando que o parâmetro de contexto **matrix_partition** é o próprio limite, ele pode ser omitido desta descrição, tornando esse parâmetro *livre*. Parâmetros livres indicam que o componente concreto não fez suposições a respeitos dessas variáveis de forma que podem ser considerados adequados para qualquer valor que ela venha a assumir.

De forma similar, o desenvolvedor pode tornar parâmetros de contextos livres na especificação de componentes aninhados. Na Figura 3.6, o desenvolver apenas especifica a propriedade da matriz e seu tipo, tornando livre todos os outros parâmetros.

Nesse exemplo, a plataforma deverá instanciar um componente que atenda às variáveis indicadas, mas ficará livre quanto a escolha das variáveis não definidas. É esperado que, mesmo ao tornar variáveis livres, apenas um componente concreto seja encontrado. Entretanto, caso sejam encontrados mais de um, o HPE fará uma escolha arbitrária.

Ainda sobre o exemplo, sabemos que matrizes de blocos tridiagonais são também matrizes esparsas e, naturalmente, o tipo BlockTridiagonal deve ser definido a partir de uma herança do tipo SparseMatrix. O mecanismo de resolução irá buscar componentes concretos que casem com as variáveis de contexto. Entretanto, caso não encontre, os tipos serão generalizados. No caso acima, assumindo a hierarquia de tipos e as variáveis livres, o componente concreto SparseLinearSystemSolver pode ser atribuído de forma segura a um componente aninhado cujo tipo está especificado na Figura 3.6.

3.6.5 Descrição Arquitetural e Configuração de Componentes

HCL (*Hash Configuration Language*) é uma linguagem de descrição arquitetural e configuração de componentes #, desenvolvida para as necessidades do HPE. A sintaxe abstrata do HCL está apresentada na gramática da Figura 3.7.

Figura 3.7: Sintaxe Abstrata de HCL

$\texttt{config} \rightarrow kind\ \ header\ \ inner^*\ \ unit^+$

$header \rightarrow$ configId innerId* $\ paramType^*\ \ instTypeNoVar^?$

$paramType \rightarrow$ parId varId $\ instTypeNoVar$

$instType \rightarrow instTypeNoVar \mid$ varId

$instTypeNoVar \rightarrow$ configId $\ instType^*$

$inner \rightarrow$ innerId $\ instType\ \ innerId^*$

$unit \rightarrow \texttt{parallel}^?$ unitId $\ slice^*$ actionId*

$slice \rightarrow$ innerId unitId $\ \texttt{index(i)}^?$

$kind \rightarrow$ `application` | `computation` | `synchronizer` | `data` |
`environment` | `architecture` | `qualifier` | `topology`

Fonte: Próprio autor.

De acordo com a gramática, um componente abstrato, representado pelo símbolo não-terminal (variável) inicial ***config***, declara como elementos de configuração constituintes um conjunto de componentes aninhados (*inner*) e um conjunto não-vazio de unidades (*unit*). Além disso, declara a espécie à qual pertence (*kind*) e um cabeçalho, onde estão declarados os seguintes elementos de configuração:

- o seu nome (configId), usado para referenciá-lo em outras configurações HCL;
- uma lista de identificadores de componentes aninhados ditos *componentes públicos*;
- uma lista de parâmetros de contexto, onde cada um deles (*paramType*) é definido pelo seu nome identificador (parId), seu nome de variável (varId) e seu limite (*instTypeNoVar*);
- e um tipo instanciação opcional que representa o componente a partir do qual está sendo derivado, por herança.

A variável gramatical *instType* representa um tipo instanciação que pode ser usado para definir o tipo de um componente aninhado. Por isso, pode derivar

apenas o nome de uma variável. A variável $instTypeNoVar$ é um tipo instanciação completo, sendo necessário para delimitar um parâmetro formal de contexto, pois esse não pode assumir o valor de uma variável.

Para as unidades (*unit*), são declaradas suas fatias e ações. A unidade possui ainda um qualificador opcional *parallel* indicando que esta correponde a um conjunto homogêneo de unidades. O mapeamento de unidades em uma unidade paralela sobre as unidades de um componente que o contém é determinado dinamicamente pelo componente da espécie *topologia* associado ao componente, levando em consideração suas partições

Uma fatia é descrita pelo identificador de componente aninhado do qual provém (innerId), o nome da unidade do componente aninhado (unitId) e opcionalmente o seu índice (*index(i)*), o qual define uma partição de uma unidade paralela. Cada unidade pode importar apenas uma unidade (fatia) por componente aninhado, de forma que innerId passa a identificar uma fatia unicamente no contexto de uma unidade.

Uma unidade paralela dita particionada (*split*) deve possuir fatias de declaradas em unidades diferentes da configuração com os índices 0 a $n-1$, onde n corresponde ao número de partições.

Na implementação de cada componente que representa uma ação, das espécies *aplicação*, *computação* e *sincronizador*, é definida um método `go`, que representa a sua ativação.

A Figura 3.8 apresenta um exemplo de um componente abstrato especificado em HCL, chamado MatVecProduct, destinado a calcular o resultado multiplicação de uma matrix a por vetor x, guardando o resultado em um vetor v. MatVecProduct possui os seguintes elementos de descrição arquitetural:

- seis parâmetros de contexto, identificados por *number_type*, *mp_interface*, *matrix_partition*, *matrix_partition*, *input_vector_partition* e *output_vector_partition*;
- três componentes aninhados, identificados por a, x e v;
- uma unidade paralela, identificada por *calculate*.

Respectivamente, os parâmetros de contexto determinam que as implementações de MatVecProduct devem ser especializadas conforme o tipo de dado dos elementos da matriz, a interface de passagem de mensagens utilizada, as estratégias de particionamento da matriz, do vetor de entrada e do vetor de saída.

Figura 3.8: Exemplos de Programas HCL (Sintaxe Concreta)

```
computation MatVecProduct(a, x, v)
                    [number_type = T : Number,
                     mp_interface = E : MessagePassingInterface,
                     matrix_partition = Da : MatPartition,
                     input_vector_partition = Dx : VecPartition,
                     output_vector_partition = Dv : VecPartition]
begin
  data a : ParallelMatrix[number_type = T, mp_interface = E, partition = Da]
  data x : ParallelVector[number_type = T, mp_interface = E, partition = Dx]
  data v : ParallelVector[number_type = T, mp_interface = E, partition = Dv]

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

Fonte: Próprio autor.

Uma *unidade paralela* é uma abreviação para uma enumeração de um número arbitrário de unidades, que varia de acordo com a execução. Isso permite construir componentes # escaláveis, os quais podem adaptar-se a uma quantidade arbitrária de unidades de processamento restringida pelo tamanho da instância do problema. Através de componentes # pré-definidos da espécie *topologia*, derivados do componente # denominado Topology, é possível a identificação da unidade na enumeração em operações de comunicação, coletiva ou ponto-a-ponto, possivelmente segundo topologias não-lineares, como no MPI. Para isso, existem componentes # denominados LinearTopology, GraphTopology e CartesianTopology, derivados de Topology, respectivamente associados a topologias de processos lineares, em grafo e cartesianas.

A ação do componente abstrato MatVecProduct será implementada usando a linguagem hospedeira do HPE, a linguagem C#, na forma de um método denominado `go`.

3.7 Considerações sobre Reconfiguração Dinâmica

Diante da apresentação dos modelos de componentes acima, concluímos que tanto o CAA quanto o Fractal, em tese, dão algum tipo de suporte a reconfiguração

dinâmica. Entretanto, nas diversas publicações referenciadas, nenhum delas apresenta detalhes aprofundados de como essas reconfigurações são implementadas, sendo os mecanismos descritos de forma macro.

Até então, para o modelo HASH, não há essa funcionalidade, sendo esta dissertação motivada a oferecer esse novo recurso. Mesmo sem conhecer bem a implementação das reconfigurações nos outros modelos, há uma clara diferença no mecanismo a ser implementado no HPE. O modelo HASH, e mais particularmente o HPE, define *as estruturas de dados* como um componente a parte. Em função da estratégia de armazenar os dados em componentes de uma espécie específica, as aplicações do HPE conseguem concentrar e isolar o estado da aplicação. Essa característica é fundamental para viabilizar o nosso método de reconfiguração, o qual pressupõe que a alteração ou substituição de outras espécies de componentes, se realizada em momentos adequados, não interfere no estado da aplicação. O método proposto será apresentado em detalhes no Capítulo 4, logo a seguir.

Capítulo 4

Configuração e Reconfiguração Dinâmica de Conectores Exógenos

Aplicações em CAD, por definição, são de computação intensiva, apresentando requisitos de uso de memória e tempo de execução para obterem-se os resultados que só podem ser tratados por plataformas de computação paralela. Não são exceções as situações nas quais o tempo de execução desse tipo de aplicação alcança a magnitude de horas, dias ou meses. Portanto, como já foi motivado no capítulo introdutório desta dissertação, é relevante que os usuários de sistemas de *software* com requisitos de CAD possam acompanhar e intervir na sua execução, a fim de realizar correções ou melhorar a sua eficácia e eficiência, sem que haja desperdício das computações previamente realizadas, tendo em vista o potencial custo financeiro e de tempo envolvido no processo.

Neste capítulo, são apresentados os fundamentos de estratégias que temos proposto, para a linguagem HCL e a plataforma HPE, que viabilizem:

- a configuração de conectores exógenos;
- a realização de reconfigurações em tempo de execução;
- o monitoramento em tempo de execução de computações paralelas.

4.1 Configuração de Conectores Exógenos

Como vimos na Seção 3.6.5, a HCL necessita de extensões para possibilitar a especificação de conectores exógenos que sejam capazes de combinar a execução de unidades de componentes. Nesta seção, apresentaremos os novos construtores da

linguagem que deverão, entre outras atribuições, ser capazes de orquestrar as ações das unidades e fatias dos componentes # para o processamento da aplicação.

Neste momento, a compreensão dos conceitos que permeiam o modelo HASH é fundamental para a compreensão das propostas, de sorte que apresentaremos novamente alguns deles para reforçar as definições descritas em capítulos anteriores.

O *conector exógeno* é um componente que tem o papel de encapsular interesses de iteração entre componentes de computações, ou seja, a comunicação, coordenação, conversão, facilitação ou suas combinações. O conector assume a responsabilidade de efetuar chamadas aos componentes e gerenciar a comunicação e o fluxo de execução (ver detalhes na Seção 2.3).

Componentes # são formados por partes, chamadas *unidades* que podem ser implantadas independentemente em unidades de processamento distintas de uma máquina paralela. Componentes podem ser compostos hierarquicamente a partir de outros componentes, chamados de *componentes aninhados*. As unidades dos componentes aninhados são combinadas para compor as unidades do componente no qual fazem parte, dito *componente recipiente*. Neste caso, as unidades dos componentes aninhados constituem *fatias* de unidades do componente recipiente (ver detalhes na Seção 3.5.1).

Em tempo de execução, o componente recipiente e seus componentes aninhados podem ser vistos sob a ótica da relação cliente/servidor, onde o componente recipiente constiui um cliente que faz uso dos serviços oferecidos pelos componentes aninhados, nesse caso vistos como os servidores.

No HPE, as unidades de componentes # das espécies *computação* e *sincronizador* definem ações, que caracterizam a operação que realizam uma vez que são ativadas durante a execução.

A sintaxe abstrata inicialmente definida para a HCL foi apresentada na Figura 3.7. Para tornar possível a descrição de conectores exógenos, concentramos a nossa alteração no não-terminal *unit*. A Figura 4.1 apresenta esta mesma gramática, porém com modificações, destacados pelos elementos sublinhados, referentes a declaração de ações e condições.

4.1.1 Ações e Condições

A primeira extensão importante da linguagem HCL para suporte aos conectores exógenos diz respeito a declarações de ações em componentes das espécies *computação* e *sincronizador*. Ao invés de uma única ação implícita associada à

Figura 4.1: Sintaxe Abstrata de HCL com Novas Extensões

$\texttt{config} \rightarrow kind\ \ header\ \ inner^*\ \ unit^+$

$header \rightarrow$ configId innerId* $\ paramType^*\ \ instTypeNoVar^?$

$paramType \rightarrow$ parId varId $\ instTypeNoVar$

$instType \rightarrow instTypeNoVar \mid$ varId

$instTypeNoVar \rightarrow$ configId $\ instType^*$

$inner \rightarrow$ innerId $\ instType\ \ innerId^*$

$unit \rightarrow \texttt{parallel}^?$ unitId $\ slice^*\ \ \underline{action^*\ \ condition^*\ \ restriction^?}$

$slice \rightarrow$ innerId unitId $\ \texttt{index(i)}^?$

$\underline{action \rightarrow \text{actionId}\ \ protocol}$

$\underline{condition \rightarrow guard\_condition}$

$kind \rightarrow$ `application` | `computation` | `synchronizer` | `data` |
`environment` | `architecture` | `qualifier` | `topology`

Fonte: Próprio autor.

Figura 4.2: Sintaxe Concreta da Linguagem de Especificação de Protocolos de HCL

$protocol \rightarrow action\_call$

$action\_call \rightarrow \text{callId}^{?}\ repeat^{?}\ guard^{?}\ (action\_block \mid action\_primitive)$

$action\_block \rightarrow \texttt{seq}\ \{action\_call_1;\ action\_call2;\ \ldots;\ action\_call_n\} \mid$
$\texttt{par}\ \{action\_call_1;\ action\_call_2;\ \ldots;\ action\_call_n\} \mid$
$\texttt{alt}\ \{action\_call_1;\ action\_call_2;\ \ldots;\ action\_call_n\}$

$repeat \rightarrow \texttt{true} \mid \texttt{false}$

$action\_primitive \rightarrow (\text{innerId}.)^{?}\text{actionId}$

$guard \rightarrow guard\_block \mid \texttt{else}$

$guard\_block \rightarrow (\text{innerId}.)^{?}\text{guardId} \mid guard_1\ \texttt{and}\ guard_2 \mid guard_1\ \texttt{or}\ guard_2 \mid \texttt{not}\ guard$

$guard\_condition \rightarrow \texttt{guard}\ \text{guardId}\ guard\_block$

Fonte: Próprio autor.

unidade, cada unidade poderá declarar agora um conjunto de ações (não-terminal *action*), uma das quais poderá ser identificada pelo identificador `main`, constituindo sua ação principal. Cada ação poderá ser associada a um protocolo.

Um protocolo define como uma ação, incluindo a ação principal, é definida em termos das demais ações, declaradas dentro da própria unidade ou publicadas por suas fatias. Para isso, é definida uma linguagem de combinadores de ações cuja sintaxe é definida pela variável gramatical *protocol*, cujo papel é descrever a orquestração das ações. A gramática do protocolo está definida na Figura 4.2.

Além das ações, uma unidade pode declarar um conjunto de *condições* (não-terminal *condition*), cujo papel principal é o controle do fluxo de execução de ações na descrição de protocolos, como será mostrado adiante.

Existe uma regra simples que define a publicação de ações por parte de uma unidade, ou seja, quais dentre as ações e condições declaradas em uma unidade poderão ser usadas nos protocolos das ações de unidades recipientes. Ela estabelece que uma ação declarada em uma unidade é publicada se não faz parte do protocolo de nenhuma outra ação declarada dentro da mesma unidade. Quanto às condições, todas são publicadas.

4.1.2 Linguagem de Especificação de Protocolos

Um protocolo é definido através da chamada de ações primitivas e compostas, conforme definido pelo não-terminal *action_call*. Independetemente do tipo de ação a ser invocada, três atributos podem ser especificados: callId, *repeat* e *guard*. O atributo callId nomeia unicamente uma chamada com o objetivo de conceder a esta um valor semântico. Esse atributo será útil nos processos de reconfiguração que descreveremos nas seções adiante. *Repeat* é um atributo booleano que indica se a ação deverá ser executada repetidamente, constituindo um modificador que habilita a execução iterativa de ações (*ação iterativa*). Seu valor *default* é falso. Por fim, o não-terminal *guard* define uma guarda que condiciona a execução da ação ao seu resultado verdadeiro. Caso contrário, ignora-se a execução da ação. Em ações iterativas, uma guarda definie a condição de controle de terminação da repetição. Caso seja omitida, constitui um laço infinito.

O não-terminal *guard* é definido por um bloco ou pela constante **else**. A definição de um bloco corresponde à composição de várias guardas com operadores da álgebra booleana, conforme definido pelo não-terminal *guard_block*. A constante **else** tem o seu uso restrito a ações chamadas diretamente por um *action_block*. O seu significado corresponde à negação da disjunção de todas as guardas presentes nas outras ações imediatamente chamadas pelo mesmo *action_block* onde esta foi declarada. Em outras palavras, a ação guardada por **else** será executada, se e somente se, todas as guardas presente nas outras ações imediatamente referenciadas pelo *action_block* tiverem resultados falso.

Para descrever uma chamada de ação, iremos utilizar a notação "*guard***?***slice.action*", onde *guard* representa uma guarda, *slice* representa o nome de uma fatia da unidade recipiente e *action* corresponde ao identificador de uma das ações da fatia referenciada. Caso não exista uma condição de guarda a parte inicial poderá ser suprimida, mantendo-se "*slice.action*", a qual tem "**true?***slice.action*" como significado default.

Para inclusão dos blocos de ação, buscamos inspiração em $Haskell_{\#}$ [24]. $Haskell_{\#}$ é uma linguagem de programação de paralelismo explícito desenvolvida como uma extensão de Haskell, uma linguagem de programação funcional. Como o nome sugere, ela implementa o modelo HASH, descrevendo as computações por meio de unidades, onde estas estabelecem comunicações através de canais. Um programa $Haskell_{\#}$ é escrito em uma linguagem de configuração que orquestra as unidades através de combinadores inspirados na linguagem Occam [61]. Occam é

uma linguagem de programação baseada nos conceitos de ELP [48] (*Experimental Programming Language*) e CSP [30] (*Communicating Sequencial Processes*), projetada na década de 1980 para expressar a implementação de algoritmos concorrentes para arquiteturas de computadores baseadas em processadores de múltiplos núcleos chamados *transputers*. Na linguagem Occam, uma aplicação descreve um conjunto de *processos* paralelos que se comunicam através de *canais*. Os processos podem ser *primitivos* ou *compostos*[1]. Os processos primitivos são definidos a partir de instruções de fina granularidade como definição e atribuição de valor a variáveis, operações aritmética ou entrada e saída de dados via canais. Os processos compostos são arranjos dos processos primitivos através de *combinadores de ação* que expressam sequência (combinador SEQ), paralelismo (combinador PAR), alternativa (combinador ALT), dentre outros.

Assim, por analogia, as configurações dos conectores exógenos do HPE serão definidos por combinadores de modo a compôr processos compostos, onde as ações dos componentes nativos assumem o papel de processos primitivos.

Para constituir a linguagem de especificação de protocolos da HCL, selecionaremos um subgrupo dos combinadores da Occam, conforme listados a seguir:

Sequência: O operador de sequenciamento de ações, representado por **seq** $\{g_1?s_1.a_1;\ g_2?s_2.a_2;\ \ldots;\ g_n?s_n.a_n\}$, descreve a execução sequencial de um conjunto de ações $s_i.a_i$, para $1 < i \leq n$, onde g_i é o identificador da condição de guarda, s_i é um identificador de uma fatia da unidade e a_i é o identificador de uma de suas ações. A ação a_i da fatia s_i só iniciará se a condição g_i for satisfeita, de modo que a sua avaliacão ocorrerá após a avaliação e possível execução da ação $g_{i-1}?s_{i-1}.a_{i-1}$. Caso contrário, a execução de a_i é apenas ignorada.

Paralelo: O operador de paralelização de ações, representado por **par** $\{g_1?s_1.a_1;\ g_2?s_2.a_2;\ \ldots;\ g_n?s_n.a_n\}$, executa concorrentemente todas as ações para as quais a guarda for verdadeira. Caso a unidade de processamento onde as ações executarão seja multiprocessada ou com múltiplos núcleos de processamento, serão executadas em paralelismo real, o que seria a situação desejável. Caso contrário, as ações são intercaladas.

Alternativa: O operador de escolha entre ações, representado por **alt** $\{g_1?s_1.a_1;\ g_2?s_2.a_2;\ \ldots;\ g_n?s_n.a_n\}$), executa uma única ação arbitrária

[1]Em inglês são chamados de *constructed processes* [61].

dentro do conjunto de ações para a qual a condição booleana seja verdadeira. As condições são avaliadas paralelamente. Quando a guarda retorna um valor verdadeiro, um *thread*, independente das demais, executará a ação escolhida.

Conforme anteriormente mencionado, opcionalmente o programador pode declarar uma ação de *escape*, designada pela condição de guarda *else*. A ação será executada se todas as condições de guarda anteriores forem falsas.

Assim, a sintaxe da linguagem de protocolos é generalizada para *blocos de ações* (*action_block*), distintas de *ações primitivas* (*action_primitive*), de maneira a permitir composições entre ações de forma hierárquica através desses combinadores.

Para dar valor semântico e simplificar o reuso de condições de guarda complexas, o programador poderá nomeá-las (guardId) conforme definido pelo não-terminal *guard_condition*.

4.1.3 Restrição e o Processo de Validação

Uma unidade poderá descrever uma restrição, usando a palavra reservada restriction. Ela tem a função de definir restrições para a execução das ações publicadas pela unidade, através da descrição de um protocolo. Há casos em que as ações de uma unidade devem obedecer a uma determinada ordem, seja total ou parcial, para que sejam utilizadas em outras configurações.

Por exemplo, assuma que uma determinada unidade de um componente possui três ações chamadas a_1, a_2 e a_3. Existe, porém, uma restrição nesta unidade que exige que as duas primeiras ações sejam chamadas, em qualquer ordem, antes da terceira. Assim, uma declaração restriction irá especificar um protocolo que expressa a restrição desejada. A sintaxe de definição deste protocolo é similar à utilizada para definir o procolo das ações, onde **seq** e **par** atuam na definição de ordem parciais e totais de execução, enquanto que as guardas expressam obrigatoriedade de associação entre condições e ações dentro do contexto onde a ações está sendo ativada. Ou seja, uma restrição que inclua a chamada "*cond***?***action*" indica que a ação *action* deverá ter sempre a sua chamada condicionada à condição *cond*.

Utilizando o exemplo das três ações, a_1, a_2 e a_3, caso desejemos que a terceira ação só possa ser chamada após as duas anteriores, estas em qualquer ordem, um possível protocolo seria **seq**{**par**{a_1; a_2}; a_3}.

Em termos formais, uma restrição define uma linguagem formal cujos elementos são *traços* de execução válidos para as ações restringidas. Qualquer traço de execução que inclua essas ações só é considerado válido com relação a restrição se,

ignorando-se as ações que não estão referenciadas na restrição, está incluído dentro da linguagem formal definida pela restrição. No exemplo apresentado, a linguagem formal definida pela restrição é formada pelos traços $\{a_1a_2a_3, a_2a_1a_3\}$. Nesse caso, uma ação cujo protocolo seja $\mathbf{seq}\{a_1;\ b_1; \mathbf{par}\{b_2; \mathbf{seq}\{a_2;\ a_3\}\}\}$ é válido com relação a essa restrição, uma vez que ignorando-se as ações b_1 e b_2, que não são restringidas pela restrição, temos o conjunto de traços $\{a_1a_2a_3\} \subseteq \{a_1a_2a_3, a_2a_1a_3\}$.

4.2 Conector

Com as extensões apresentadas na seção anterior, a nova sintaxe da HCL é capaz de descrever completamente um conector exógeno. Entretanto, para que seja funcional, deve existir um arcabouço que tenha a capacidade de interpretar e executar os protocolos das ações descritas por essa sintaxe. A implementação buscou reduzir ao máximo a necessidade de modificações sobre a plataforma HPE, uma vez que estamos interessados em uma simples, embora abrangente, prova de conceitos introduzidos nessa dissertação.

No Front-End, foram incluídas a descrição de ações, condições e protocolos no editor visual de configurações de componentes abstratos. Caso um programador deseje que um componente abstrato seja um conector, pode gerar uma configuração HCL para aquela configuração, a qual ficará armazenada em um arquivo de extensão "hcl". Por simplicidade, codificamos a sintaxe concreta do HCL usando XML, evitando a necessidade da construção de um compilador para a linguagem intermediária usada pelo Back-End para execução de ações. Ao ser implantado, se o componente abstrato possuir uma configuração HCL previamente gerada, ou seja, trata-se de um conector, o arquivo da configuração é enviado para o Back-End e armazenado para uso subsequente.

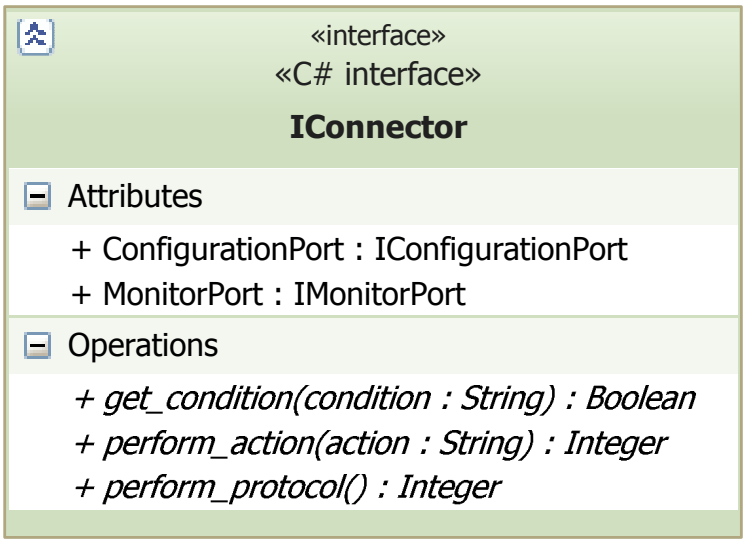
Para concluir a implantação de um conector, ainda utilizando o Front-End, o programador pode definir um componente concreto a partir do componente abstrato que representa o conector, porém sem especificar o código C# que descreve a implementação de cada uma das suas unidades, o qual será gerado automaticamente pelo Back-End a partir da configuração HCL, usando a API CodeDOM da plataforma .NET. A geração automática desse código é necessária para manter a uniformidade e compatibilidade dos componentes conectores com os demais componentes, nativamente programados em C#, de modo até que possam ser intercambiáveis.

Espera-se que o contexto do componente concreto conector criado seja o mais

genérico possível para o componente abstrato. Porém, essa abordagem ainda permite que o usuário defina um contexto mais específico no qual o conector é instanciado. Por exemplo, isso permitiria que um desenvolvedor possa incluir no componente abstrato um parâmetro de contexto qualificador para controlar explicitamente se deseja que um componente conector seja usado.

O código automaticamente gerado das unidades do componente conector implementa a interface IConnector, apresentada na Figura 4.3, cujos métodos serão apresentados adiante.

Figura 4.3: Definição da interface IConnector



Fonte: Próprio autor.

A manipulação da configuração HCL no formato XML mostrou-se bastante adequada para a prototipação de um ambiente de execução, o qual será apresentado na próxima seção.

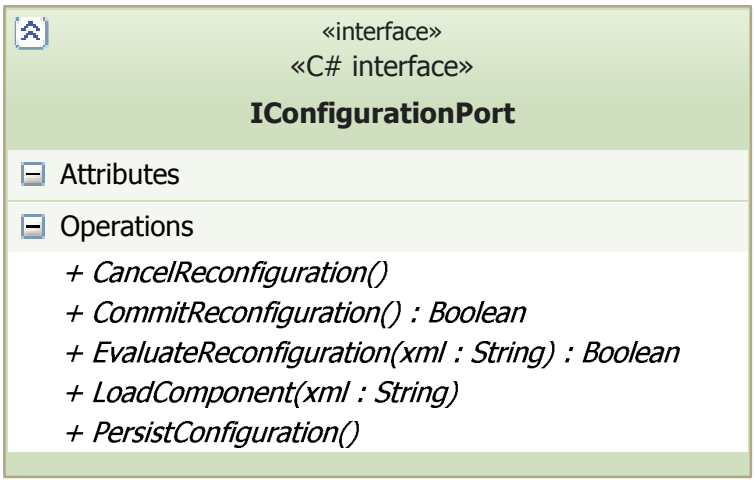
4.2.1 O Ambiente de Execução

O processo de execução de um componente # qualquer é iniciado quando este já está implantado no Back-End do HPE. Os conectores serão naturalmente escolhidos pelo sistema de resolução de tipos do HTS supondo que tenham sido implantados no Back-End segundo o procedimento descrito na seção anterior, e não exista algum componente concreto, não conector, implantado para um contexto mais específico.

Seguindo a especificação CCA, a forma de interação de um componente # com seus componentes aninhados e com o próprio HPE se dá através de portas. Além das

portas já suportadas por componentes #, como a GoPort e AutomaticSlicePort, outras duas foram adicionadas para as necessidades especiais de componentes conectores, assim chamadas: ConfigurationPort e MonitorPort. Essa última será apresentada juntamente com os processos de monitoração, enquanto a interface da primeira é apresentada na Figura 4.4.

Figura 4.4: Definição da interface IConfigurationPort



Fonte: Próprio autor.

A implementação da porta ConfigurationPort é realizada pela classe ConfigurationManager. Porém as suas responsabilidades não se limitam à implementação dessa interface, sendo elas subdivididas em 4 macro atribuições:

- Carga dos metadados e criação dos autômatos;
- Execução dos autômatos;
- Monitoração da execução;
- Reconfiguração dos componentes e ações.

Um objeto ConfigurationManager centraliza essas responsabilidades, porém delegando para outras classes cada um dos itens listados. A seguir, definiremos o conceito de *autômato de protocolo* e apresentaremos essas classes responsáveis, assim como a sua forma de implementação. A exceção é a reconfiguração, a qual será apresentada somente na Seção 4.3.

Carga dos Metadados

A porta ConfigurationPort, implementada pelo ConfigurationManager, possui cinco métodos, dos quais iremos nos ater apenas ao LoadComponent[2].

Ao instanciar um conector para execução, a descrição de sua configuração em forma de linguagem intermediária é repassada através do método LoadComponent, o qual realiza uma nova tradução, desta vez da linguagem intermediária para metadados de execução. Estes dados são organizados em objetos de forma otimizada a favorecer uma manipulação eficiente. Especificamente no que tange o protocolo de configuração, há a tradução para um autômato de execução, inspirado em um autômato finito conforme veremos adiante.

A carga dos metadados foi delegada à classe XmlLoader, a qual chamamos de *carregador*. Considerando que a linguagem intermediária do modelo é uma representação em XML, o carregador deve realizar um *parsing* e estruturar os dados contidos no XML em classes do modelo. Antes da realização deste procedimento, o XmlLoader realiza uma verificação do formato do arquivo submetido através da classe XmlFormatValidation, a qual faz uso de um XSD.

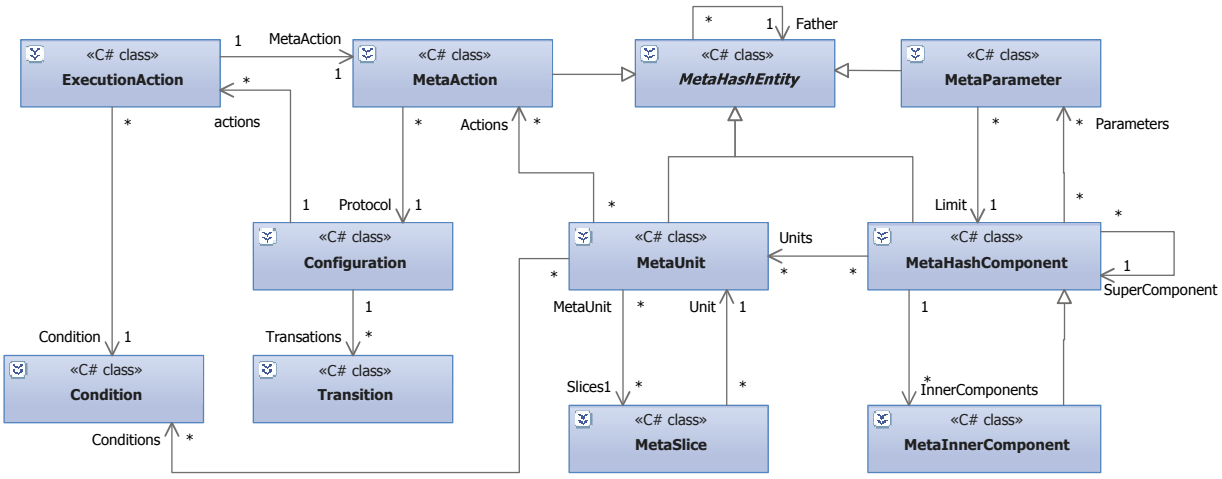
Uma vez validados, conforme já mencionamos, os metadados contidos no arquivo XML são estruturados em classes que armazenam essas informações de forma hierárquica, buscando representar fielmente a estrutura do conector e suas unidades, assim como seus componentes aninhados e fatias. A Figura 4.5 apresenta um diagrama UML expondo as relações entre as classes. Por questões de simplicidade, ocultamos todos os atributos e métodos das classes de forma a ressaltar apenas a relação entre elas.

O preenchimento dos dados do conector inicia através da classe MetaHashComponent, que herda da classa MetaComponent. Além de armazenar as informações básicas, como *name* e *package*, dentre outras, são armazenadas as informações de parâmetros de contexto, através da relação com a classe MetaParameter. De forma semelhante, são relacionados os dados das unidades e componentes aninhados, respectivamente pelas classes MetaUnit e MetaInnerComponent. Existe ainda um auto-relacionamento para representar o super-componente.

MetaInnerComponent é definido a partir de uma herança com MetaComponent, de forma que os parâmetros, unidade, e por ventura, componentes aninhados, também

[2]Os outros métodos serão tratados em seções futuros considerando outras responsabilidades concentradas nesta porta.

Figura 4.5: Diagrama das Classes de Metadados



Fonte: Próprio autor.

sejam modelados. A classe diverge apenas quanto a inclusão de novos atributos que indique o seu identificador para o componente pai, além do indicador de publicidade do componente aninhado.

Todas as classes listadas, seja de forma direta ou indireta, herdam da classe abstrata MetaHashEntity. Essa classe define os atributos básicos de um meta elemento, como um nome, um identificador único e um pai, esse último implementado através de um auto-relacionamento. Essas informações são de extrema importância para a futura aplicação dos processos de reconfiguração. Além dessas, a classe MetaAction também realiza essa herança.

A classe MetaUnit representa unidades de componentes #. Além da relação com o componente pai, ela possui três outras relações, referentes às suas fatias, às suas condições e às suas ações, cada qual representada por uma lista, respectivamente, das classes MetaSlice, Condition e MetaAction, respectivamente.

A MetaSlice possui um atributo do tipo MetaUnit que representa a unidade do componente aninhado que assume a papel de fatia.

O atributo do tipo Configuration da classe MetaAction merece destaque. Ela representa o protocolo das ações, cuja representação se dá na forma de um *autômato de protocolo*, a ser definido na próxima seção. Esse autômato faz uso da classe ExecutionAction, a qual representa cada ocorrência de uma ação na configuração, relacionando-a com a respectiva condição. A título de exemplo, suponha que em trechos diferentes da configuração, a mesma ação a_r da fatia s_j seja ativada. Porém,

na primeira ocorrência, a ação está relacionada com uma condição c_b e na segunda sem condicional. Nesse caso, são instanciados dois objetos ExecutionAction, sendo o primeiro possuindo um condicional c_b e o segundo possuindo condicional nulo, porém ambos se relacionando com a mesma instância de MetaAction.

Autômato de Protocolo

Segundo uma definição formal [65], um autômato finito é uma tupla (Q, Σ, δ, q_0, F), onde:

- Q é um conjunto finito de *estados*;
- Σ é um conjunto finito de símbolos chamado *alfabeto*;
- $\delta : Q \times \Sigma \longrightarrow Q$ é uma *função de transição*;
- $q_0 \in Q$ é o *estado inicial*; e
- $F \subseteq Q$ é o *conjunto de estados de finais.*

No nosso contexto, o alfabeto é composto pelo conjunto de ações participantes do protocolo, ou seja, as instâncias de MetaAction. As transições representam portanto a ativação de ações, e são definidas de acordo com o significado dos combinadores do protocolo, como mostrado pelas regras de tradução adiante. O conjunto de estados finais será sempre unitário, representando o estado posterior a execução da última ação do protocolo.

Para a HCL, iremos estender o autômato finito básico incluindo mais um função na tupla, simbolizadas por τ, onde:

- $\tau : \Sigma \longrightarrow$ (*booleano*) é uma *função de condição.*

A função τ representa os condicionais associados a execução das ações. Conforme adiantado, essa função corresponde a associação definida na classe ExecutionAction, onde é possível obter a condicional que guarda a ação.

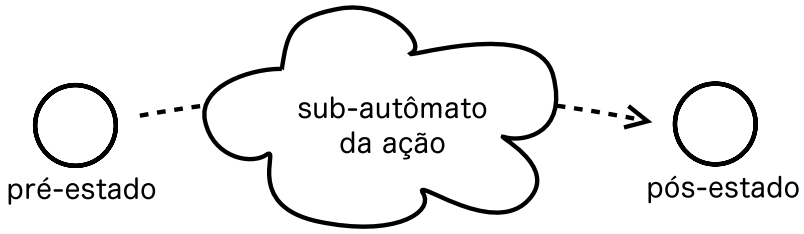
Construção do Autômato

Para cada tipo de combinador definido na Seção 4.1.1, há um procedimento padrão para transformá-lo, juntamente com as ações que este coordena, em estados e casos das funções δ e τ. Os procedimentos são sempre aplicados sobre uma ação primitiva ou combinador, mediante a passagem do estado antecessor e sucessor, os quais chamaremos de *pré-estado* e *pós-estado.*

Cada ação coordenada por um combinador, por definição, pode ser outro combinador ou uma ação primitiva. Conforme veremos, as ações primitivas tem resolução imediata do autômato gerado, enquanto que os combinadores exigem um processo em que o mesmo procedimento é aplicado recursivamente, porém atribuindo pré e pós-estados diferentes. Em função dessa característica, na apresentação do processo de geração dos autômatos de combinadores, chamaremos os passos intermediários de *sub-autômatos*.

Por padrão todo protocolo tem apenas um estado inicial e um estado final. Assim, o processo de geração do autômato inicia definindo esses estados e os repassando para a ação que define todo o protocolo, conforme a Figura 4.6.

Figura 4.6: Processo de Construção de Autômato



Fonte: Próprio autor.

Conforme definido na gramática da Figura 4.2, todo o protocolo pode ser entendido como uma única ação, a qual pode ser primitiva, referente a uma ação de uma fatia, ou uma ação de bloco, a qual corresponde a uma combinador. Independentemente do tipo da ação, o nosso tratamento seguirá um procedimento padrão: definir os estados inicial e final da ação e realizar o tratamento das suas ações aninhadas, quando houver.

Por esta linha de raciocínio, partiremos de caso mais simples de tratamento, o nosso caso base, o qual corresponde a um protocolo que descreve apenas a execução de uma ação primitiva, sem repetição e sem condição.

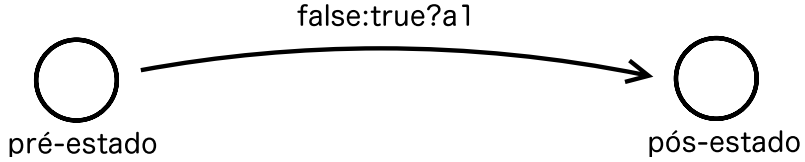
Ação Primitiva: Tratamento de uma ação primitiva incondicional sem repetição (a_1):

- Não é necessário a adição de nenhum novo estado;
- É criado uma nova transição na função δ:
  $\delta\ (q_{pre-estado},\ a_1)\ \longrightarrow q_{pos-estado}$

- É criado uma nova transição na função τ:
  $\tau\ (a_1)\ \longrightarrow true$

Diagramaticamente, a tradução é representada pela Figura 4.7. A transição é nomeada por *repeat:cond?*a_1, onde *repeat* indica se a transição tem repetição (nesse caso o valor é falso), *cond* corresponde à condição de guarda (nesse caso o valor é verdadeiro) e a_1 indica a ação a ser realizada. Caso não sejam especificados, assumimos que *repeat* e *cond* assumam os valores *false* e *true*, repectivamente, já que a suas escritas são opcionais.

Figura 4.7: Construção de um sub-autômato de ação primitiva.



Fonte: Próprio autor.

Se existir *repetição* ou *condição*, o tratamento deverá ser diferenciado. A repetição é expressa no autômato pela substituição do pós-estado pelo pré-estado, formando assim um ciclo. Para esse caso, o pós-estado é ignorado. Por mais que esta construção seja permitida, em geral espera-se que as repetições estejam associadas a uma condição. Neste caso, além da transição acompanhada da guarda, uma nova é criada, sendo associada a uma ação padrão λ, chamada de *lambda*. Uma ação *lambda* não realiza computação alguma, tendo o papel apenas de transitar do pré-estado para o pós-estado, expressando assim a continuidade da execução quando a condição assumir o valor falso. Assim, uma ação a_1 condicional com repetição, expressa diagramaticamente pela Figura 4.8, tem a seguinte definição:

Ação Genérica com Repetição: Tratamento de uma ação genérica condicional com repetição (a_1):

- Não é necessário a adição de nenhum novo estado;
- São criadas duas transições na função δ:
  $\delta\ (q_{pre-estado},\ a_1)\ \longrightarrow q_{pre-estado}$
  $\delta\ (q_{pre-estado},\ \lambda)\ \longrightarrow q_{pos-estado}$
- São criadas duas transições na função τ:

$\tau\ (a_1) \longrightarrow cond$
$\tau\ (\lambda) \longrightarrow else\ (not\ cond)$

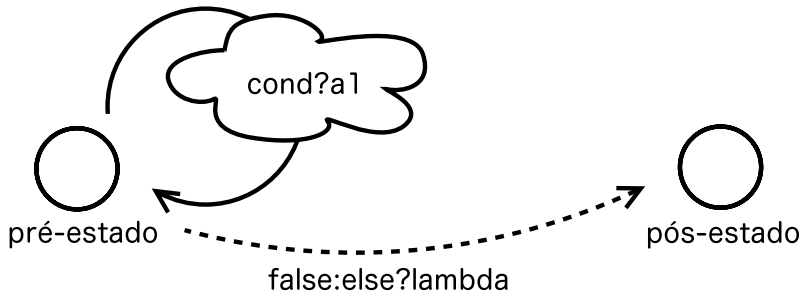
Em último caso, com este exemplo, fica fácil compreender que uma ação condicional, porém sem repetição assume o mesmo tratamento, devendo substituir a transição:

- $\delta(q_{pre-estado},\ a_1) \longrightarrow q_{pre-estado}$

Pela transição:

- $\delta(q_{pre-estado},\ a_1) \longrightarrow q_{pos-estado}$

Figura 4.8: Construção de um sub-autômato de repetição condicional.



Fonte: Próprio autor.

Uma vez que já temos o nosso caso base, podemos definir os casos em que traduziremos uma ação composta, ou seja, combinadores, em autômatos. No caso em que os combinadores sejam guardados por condicionais ou possuam repetição, podemos generalizar os entendimentos apresentados até então, considerando a_1 como um sub-autômato, ao invés de uma ação simples, conforme apresentado pela nuvem da Figura 4.8.

Combinador SEQ: Tratamento para **seq**$\{a_1;\ a_2;\ \ldots;\ a_n\}$:

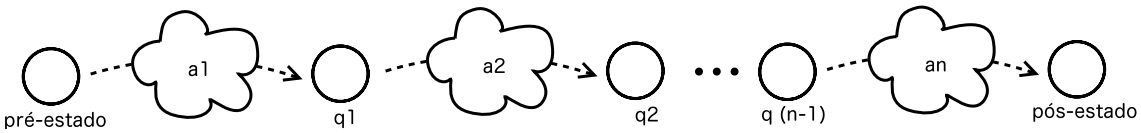
- Serão adicionados $n-1$ novos estados ao autômato, onde n é o número de ações a ser sequenciada[3]. Nos referenciaremos a esses estados por n_x, onde x é um inteiro positivo não nulo, atribuído de forma sequencial;

[3]Assumimos que $n > 1$. No caso $n = 1$, o tratamento é o mesmo do caso base.

- Serão criadas n novas transições na função δ:
  $\delta\ (q_{pre-estado},\ a_1)\ \longrightarrow q_1$
  $\delta\ (q_x,\ a_{x+1})\ \longrightarrow q_{x+1},\ para\ 1 < x < n-2$
  $\delta\ (q_{n-1},\ a_n)\ \longrightarrow q_{pos-estado}$

Ao contrário do caso base, as ações descritas nos combinadores podem ser ações primitivas ou outros combinadores. Representamos diagramaticamente estas últimas como nuvens, conforme a Figura 4.9, tendo em vista que elas definirão seu próprio sub-autômato.

Figura 4.9: Construção de um sub-autômato para o combinador SEQ.



Fonte: Próprio autor.

O tratamento do combinador SEQ consiste em criar e definir os novos pré e pós-estados de cada uma das transições em sequência. Assumindo que a_x pode ser uma ação qualquer, primitiva ou composta, cada uma delas deverá posteriormente ser avaliada para se determinar precisamente o seu sub-autômato. Por este motivo é que, para este caso, não definimos a função τ.

Combinador PAR: Tratamento para $\mathbf{par}\{a_1;\ a_2;\ \ldots;\ a_n\}$

- Não é necessário a adição de nenhum novo estado;
- Serão criadas n novas transições na função δ:
  $\delta\ (q_{pre-estado},\ a_x)\ \longrightarrow q_{pos-estado},\ para\ 1 \leq x \leq n$

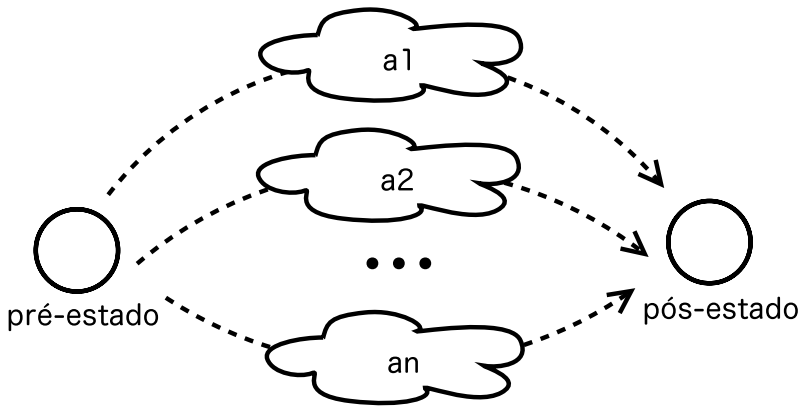
A Figura 4.10 apresenta a representação do autômato gerado pelo combinador PAR.

Similarmente ao que ocorre com o combinador SEQ, o tratamento de PAR se resume a definir os estados predecessores e sucessores das transições, sendo a definição da função τ feita mediante a avaliação dos sub-autômatos a_x.

Combinador ALT: Tratamento para $\mathbf{alt}\{cond_1?a_1;\ cond_2?a_2;\ \ldots;\ cond_n?a_n\}$

- Não é necessário a adição de nenhum novo estado;
- Serão criadas n novas transições na função δ:
  $\delta\ (q_{pre-estado},\ a_x)\ \longrightarrow q_{pos-estado},\ para\ 1 \leq x \leq n$

Figura 4.10: Construção de um sub-autômato para o combinador PAR.



Fonte: Próprio autor.

Comparativamente a PAR, não há diferenciação nos estados ou na geração da função δ. Porém, o combinador ALT irá interferir no condicional de cada uma das ações combinadas, motivo pelo qual explicitamos a existência de uma condição associada a cada uma das ações combinadas.

Cada um das condições $cond_x$ irá ser substituida por uma conjunção ($cond_x$ **and** $i == k$), onde i representa um número inteiro exclusivo atribuído a cada condição e k um interior único a ser determinado em tempo de execução pelo estado predecessor de ALT. Desta forma, além de depender da condição inicialmente determinada, a execução de a_x também dependerá de k, o qual será escolhido aleatoriamente entre as ações em que $cond_x$ tem valor verdadeiro. Estas alterações serão expressas na definição da funcão τ realizadas pelo tratamento dos sub-autômatos a_x.

Diagramaticamente falando, o combinador ALT é similar ao combinador PAR, apresentado na Figura 4.10.

Implementação do Autômato

Durante o processo de geração do autômato, seja qual for o combinador descrito, a classe Transition é utilizada para armazenar todos os dados envolvidos na transição. Esses dados posteriormente são organizados como matrizes, de forma a permitir uma execução mais eficiente. As ações presentes nos Transition são representadas pela classe ExecutionAction, as quais realizam o mapeamento entre os condicionais na forma de classes Condition e as ações das unidades e fatias, na forma de MetaAction.

Conforme vimos na Seção 3.6.3, os componentes # são implantados no Back-End

na forma de arquivos *assemblies* (.dll). Ao ser instanciado, em matéria de código-fonte, um componente # corresponde apenas às suas unidades que por sua vez, tratam-se de objetos C#. A finalização da estruturação dos dados de execução corresponde a ligação entre os metadados e o objeto da unidade. Através de um processo de criação de *delegates*, as MetaAction e Condition são ligadas aos seus respectivos metadados, precedido das junções das unidades aos objetos MetaUnit e das fatias aos objetos MetaSlice. A partir desse ponto, o processo executor descrito a seguir poderá disparar a execução das ações e condições tomando por base os metadados.

Implementação do Interpretador

O ConfigurationManager confere à classe Interpreter a responsabilidade pela execução interpretada do autômato. Na prática, além do Interpreter, as classes BranchInterpreter e StateControl também colaboram para a realização desta tarefa.

O processamento é realizado de forma *multithread*, de modo que o *thread* inicial, responsável pelo início do execução, dispara o método go de uma instância da classe Interpreter. Esta instância inicial é criada juntamente com o ConfigurationManager, o qual repassa para este objeto os metadados de uma unidade, tão logo haja a carga do componente # pelo método LoadComponent.

A instância do Interpreter, por sua vez, delega a execução da configuração a uma instância da classe StateControl, repassando o estado inicial do autômato. A partir de agora, a nossa instância de Interpreter ficará responsável apenas por receber a notificação do término da execução.

Ao contrário da classe anterior, StateControl terá várias instâncias envolvidas na execução. Como o nome sugere, para cada estado do autômato, uma instância desta classe será criada. Um estado de um autômato é definido por três informações:

- Número de transições de chegada;
- Conjunto de transições de partida, e;
- Transição de escape (definida pelo condicional **else**).

O StateControl é o responsável por disparar as transições (ações) que partem de si. O *número de transições de chegada* é utilizado para determinar o momento em que o estado já está apto a executar. É ele quem condiciona o início da execução. Podemos entender cada estado como sendo um mecanismo de *fork-join*. Assim que todas as

transições esperadas tenham chegado, o StateControl dispara paralelamente, através de múltiplos *threads*, a execução do seu *conjunto de transições de partida*. O disparo destas transições ocorrem através da terceira classe que compõe o interpretador: BranchInterpreter.

As instâncias da classe BranchInterpreter realizam a chamada às ações nativas das unidades dos componentes aninhados e também ficam responsáveis por realizar as diversas notificações, que abrangem:

- Para o StateControl, o resultado da avaliação do condicional atrelado a ação e o término da execução;
- Para os monitores, o início e término da execução;

Para o StateControl, é fundamental receber como retorno de cada um dos BranchInterpreter o resultado da avaliação do condicional da ação a ele delegada. Após receber todos os resultados, o StateControl irá decidir se deve ou não disparar a execução da ação de escape associada àquele estado. Uma ação de escape é aquela cujo condicional tem valor **else**. Como vimos, esse condicional corresponde a negação da disjunção de todos os condicionais associados as ações das transições que partem do mesmo estado inicial. Para não ser necessário executar novamente cada condicional e obter o valor booleano para **else**, o StateControl armazena os resultados já calculados pelos BranchInterpreter, de forma a otimizar a execução, uma vez que o nível de complexidade computacional de cada condição é indeterminada.

Essa mesma classe precisa obter a informação do término da execução para certificar que todas as ações do seu estado já foram concluídas e, consequentemente, possibilitar o início da execução dos estados subsequentes, quando assim for aplicado.

Vale ressaltar que a implementação dessa execução se dá de uma forma em que não há um *thread* principal. Após disparar os BranchInterpreter, o ramo de execução do StateControl é finalizado. Quem dará início aos estados subsequentes é o último *thread* a notificar a finalização de sua execução.

Implementação do Monitor

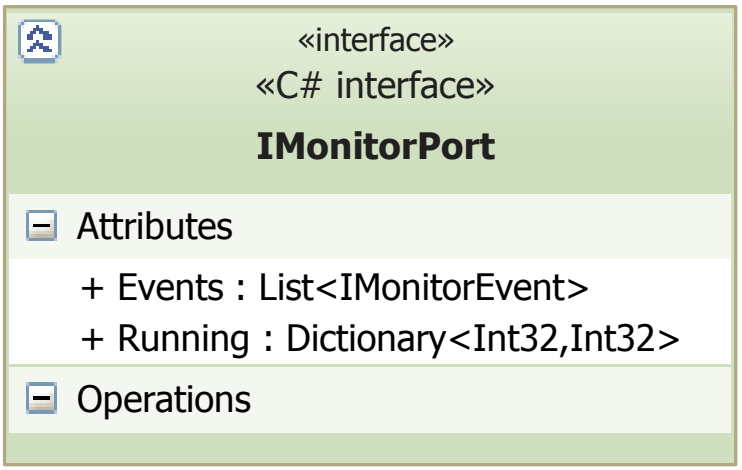
O processo de monitoração tem o objetivo de fornecer informações sobre o andamento da execução de um componente conector. Além de dar uma visão do processamento, a monitoração objetiva apresentar ao programador informações que o permita definir se futuras reconfigurações são desejáveis, além de verificar se o desempenho de sua aplicação está dentro do esperado.

O processo de reconfiguração também faz uso das informações do monitor, de forma a subsidiar os analisadores de segurança de reconfiguração dinâmica. Detalhes dos analisadores e o uso das informações dos monitores, serão apresentados na Seção 4.3.

Cada componente conector exporta uma porta do tipo MonitorPort definida pela interface apresentada na Figura 4.11, a qual possui duas propriedades. A primeira trata-se de lista, disposta em ordem cronológica, dos eventos ocorridos durante a execução. O monitor registra apenas um tipo de eventos, o qual corresponde a finalização de uma ação, indicada pelo par transição-estado, representada pela interface IMonitorEvent.

A segunda propriedade, Running, indica a quantidade de instâncias de uma dada ação que está sendo executada em um dado momento.

Figura 4.11: Definição do MonitorPort.



Fonte: Próprio autor.

Ao unir as duas propriedades, temos a informação completa sobre o andamento da execução. As propriedades foram organizadas desta maneira para facilitar o analisador de segurança de reconfigurações.

4.3 Reconfigurações Estáticas e Dinâmicas

Como premissa para inclusão de reconfigurações na plataforma HPE, precisamos avaliar quais os casos em que esse tipo de operação poderá ser aplicada. Assim, iremos classificar os diversos tipos de mudanças e de reconfigurações e esclarecer qual o seu significado no contexto do modelo, a fim de definir a estratégia de

reconfiguração a ser utilizada para cada caso.

4.3.1 Classificação das Mudanças na Plataforma HPE

A taxonomia apresentada na Seção 2.1 é bastante completa e extensa. Se por um lado ela permite a avaliação da evolução de *software* por várias visões diferentes, por outro gera um número muito grande de conceitos e vertentes, tornando a classificação uma atividade complexa. Em função disso, realizamos uma seleção e agrupamento das dimensões dessa classificação gerando a nossa própria taxonomia, mais apropriada aos nossos propósitos. Assim, essa revisão buscou-se julgar empiricamente a relevância de cada dimensão dentro do contexto de uma taxonomia de mudanças para a plataforma HPE.

Ao longo dessa seção, iremos fazer uso dos termos e conceitos da taxonomia para reconfigurações apresentada na Seção 2.1.

A Figura 4.12 apresenta, através de notação própria, a taxonomia de mudança para a plataforma HPE. Essa nova classificação não respeita os temas da classificação inicial, mas une dimensões correlatas, criando uma hierarquia entre elas.

Antes de apresentar os critérios utilizados para classificar as mudanças do HPE, iremos explicar o motivo da não utilização de algumas das dimensões da taxonomia inicial. Ao considerar o contexto do HPE, pudemos constatar que alguns classificações irão sempre possuir o mesmo valor, seja qual for a mudança proposta. Isso quer dizer que apenas uma das categorias da classificação se aplica ao contexto do HPE. Assim, na prática, esses critérios não especializam a taxonomia. A Tabela 4.1 apresenta as dimensões que se enquadram nesse item.

As dimensões *segurança* e *nível de formalidade* não serão utilizadas na classificação por não identificarmos categorias adequadas para classificar as alterações neste contexto. Entretanto, os seus conceitos serão utilizados nas avaliações das reconfigurações, conforme veremos a diante.

Uma vez explicados os motivos da supressão de algumas dimensões, iniciaremos a apresentação dos *critérios* utilizados na nova taxonomia. Definiremos como critério um conjunto de dimensões organizadas de forma hierárquica. Este agrupamento não segue, obrigatoriamente, o mesmo agrupamento dos temas. Como pode ser visto, restam poucas dimensões para a aplicação nesta classificação, o que garante uma taxonomia extremamente simples e de fácil compreensão. A taxonomia é apresentada na Figura 4.12.

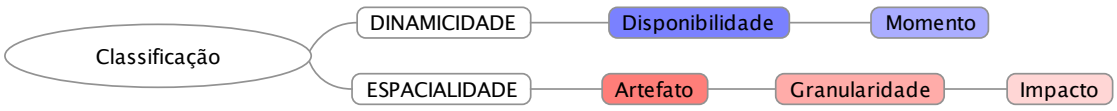
Foram definidos dois critérios: *dinamicidade* e *espacialidade*.

Tabela 4.1: Dimensões de valores padrão.

Dimensão	**Valor Padrão**	**Comentário**
Histórico da mudança	*sequencial*	O HPE sempre perceberá as alterações como sequenciais, tendo em vista que possíveis modificações paralelas serão tratadas por ferramentas de controle de configuração, apresentando ao final uma única versão.
Frequência da mudança	*arbitrárias*	O desenvolvimento de aplicações científicas, o qual corresponde ao principal perfil de aplicações do HPE, tem possível caráter experimental. Assim, pressupomos por natureza, a ausência de padrão da frequência da modificação.
Antecipação	*Não-previsíveis*	Pelos mesmo motivos listados no item anterior, pressupomos as mudanças como imprevisíveis.
Nível de automatização	*manuais*	Não faz parte do atual escopo do HPE realizar modificações automatizadas em suas aplicações.
Tipo de mudança	*semânticas*	A classificação entre estrutural e semântico, dependerá do nível do escopo avaliado. No HPE, pressupomos o maior grau de especificidade do escopo e portanto as modificações serão sempre consideradas semânticas.
Atividade	*reativas*	Similarmente ao nível de automatização, não faz parte do atual escopo do HPE modificações proativas em suas aplicações.
Abertura	*fechado*	O HPE não prevê abertura do ambiente.

Fonte: Próprio autor.

Figura 4.12: Classificação das mudanças no HPE

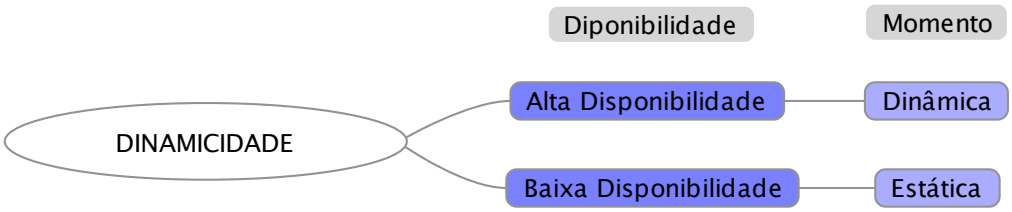


Fonte: Próprio autor.

Critério da Dinamicidade

Este critério tem o objetivo de unir dimensões que representem características da dinâmica da mudança, os quais consideramos como um agrupamento hierárquico das dimensões *disponibilidade* e *momento da mudança*, conforme apresentamos na Figura 4.13.

Figura 4.13: Critério de dinamicidade



Fonte: Próprio autor.

Apesar de estarem em temas diferentes, entendemos que essas dimensões estão inter-relacionadas. Seus critério de classificação possuem dependência, formando uma hierarquia onde cada folha identifica um tipo dessa classificação. Porém, verificamos que os critério são dependentes, ou seja a *disponibilidade* define o *momento* e vice-versa. Assim, é suficiente dizer que no HPE, reconhecem-se os seguintes tipos de reconfigurações conforme a dinamicidade:

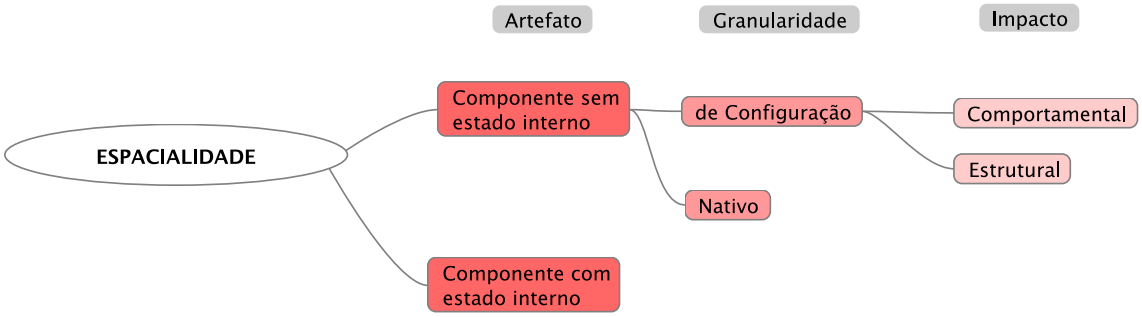
- Mudanças dinâmicas;
- Mudanças estáticas.

Critério da Espacialidade

Assim como ocorre na dinamicidade, agrupamos hierarquicamente um conjunto de dimensões interdependentes para formar uma novo critério que expresse o artefato alvo da mudança e a sua forma de alteração. Este novo critério reflete por completo o tema *objeto da mudança*. A Figura 4.14 apresenta o critério *espacialidade*.

Os itens dessa classificação merecem comentários adicionais. Para se ajustar ao HPE, adequamos o domínio aos conceitos do modelo HASH. Dividimos o grupo *artefato* em *componentes sem estado interno* e *componentes com estado interno*. Quanto a *granularidade*, dentre os componentes sem estado interno, separamos aqueles que são escritos por linguagem de propósito geral, chamados de *componentes nativos* e os que são escritos pela linguagem de descrição de arquitetura,

Figura 4.14: Critério de espacialidade



Fonte: Próprio autor.

chamados de *componentes de configuração*. No grupo *impacto*, distinguimos as mudanças que apenas substituem componentes aninhados (*estrutural*), daquelas que realizam alteração de alguma ação, condicional ou protocolo do componente (*comportamental*).

Dessa forma, o nosso espectro de classificação para o HPE é:

- Componente sem estado interno, de configuração, comportamental;
- Componente sem estado interno, de configuração, estrutural;
- Componente sem estado interno, nativo;
- Componente com estado interno.

A HCL é direcionada para a descrição de conectores exógenos, os quais correspondem a componentes sem estado interno. Por esse motivo, os componentes com estado interno serão sempre nativos. Além disso, conforme descrevemos na Seção 3.6.1, componentes com estado interno não descrevem ações e portanto não são passíveis de sofrer mudanças comportamentais.

Tipos de Reconfiguração

Uma vez definidos os critérios de avaliação, podemos definir os tipos de mudança a que os componentes do HPE estão sujeitos ao assumir a perspectiva da HCL. O cruzamento dos dois critérios nos mostra que, para cada objeto de mudança definido na *espacialidade*, é possível aplicar mudança *dinâmicas* ou *estáticas*. Avaliando detalhadamente os itens de classificação da espacialidade, concluímos que:

Componente sem estado interno, de configuração, comportamental

A alteração do comportamento de um conector exógeno ocorre na forma de *adição* ou *alteração* de uma ação ou condicional de suas unidades. Essa mudança demandará a definição do novo comportamento. A esse tipo de mudança chamaremos de ***reconfiguração comportamental***.

Componente sem estado interno, de configuração, estrutural

A alteração da estrutura do conector ocorre através da *adição* ou *alteração* dos seus componentes aninhados. Essa alteração demandará a definição explícita dos novos componentes e as respectivas novas relações entre unidades e fatias. A esse tipo de mudança chamaremos de ***reconfiguração estrutural explícita***. Como o objetivo desta reconfiguração é inserir novas fatias às unidades, os seus ganhos só serão realmente sentidos quando realizada conjuntamente com uma reconfiguração comportamental, a qual deverá fazer uso da nova fatia em alguma ação ou condicional.

Componente sem estado interno, nativo

Conforme vimos, para os componente nativo, não há possibilidade de alterações em suas definições. Estes sofrerão alterações apenas quando por influência indireta de alterações no conector exógeno. Considerando o HTS, há dependência dos componentes instanciados com os parâmetros de contexto. Assim, para realizar tais mudanças, esta alteração demandará a *alteração* dos valores dos parâmetros de contexto do conector exógeno ou de seus dos componentes aninhados. Mediante este procedimento, o HPE poderá identificar um componente mais adequado para o novo contexto apresentado. A este tipo de mudança chamaremos de ***reconfiguração estrutural implícita***.

Componente com estado interno

O processo de reconfiguração proposto limita a reconfiguração há componentes sem estado interno, de forma que este grupo não sofre qualquer tipo de reconfiguração. Estudos no futuro podem permitir tais reconfiguração a partir da existência de um processo de transferência de estado, o qual proposto na Seção 6.3.

Figura 4.15: Sintaxe Abstrata de HCL com Novas Extensões

$\texttt{reconfig} \rightarrow header\ \ inner^*\ \ unit^*$

$header \rightarrow$ targetConfigId innerId* $paramType^*$

$paramType \rightarrow$ parId varId $instTypeNoVar$

$instType \rightarrow instTypeNoVar \mid$ varId

$instTypeNoVar \rightarrow$ configId $instType^*$

$inner \rightarrow$ innerId $instType$ $innerId^*$

$unit \rightarrow$ unitId $slice^*$ $action^*$ $guard^*$ $validation^?$

$slice \rightarrow$ innerId unitId $\texttt{index(i)}^?$

$action \rightarrow$ actionId reconfigPointId$^?$ $protocol^?$

$guard \rightarrow guard\_condition$

$validation \rightarrow protocol$

Fonte: Próprio autor.

4.3.2 Estratégias de Reconfiguração

Dependendo do tipo de reconfiguração que se deseje realizar sobre um componente, considerando o critério da dinamicidade, estratégias diferentes devem serão aplicadas. No caso das alterações dinâmicas, faremos uso da porta de reconfiguração, enquanto que para as alterações estáticas a estratégia será a escrita de componentes utilizando herança funcional.

Porta de Reconfiguração

A realização de reconfigurações dinâmicas na plataforma HPE se dará através da especificação de um *script* que descreve as alterações a serem realizadas no componente. Para isso, foi definida uma linguagem de reconfiguração, também baseada em XML, de forma a permitir a descrição de um procedimento de reconfiguração. A gramática apresentada na Figura 4.15 define a sintaxe abstrata do *script* de reconfiguração.

Como podemos verificar, a linguagem de reconfiguração é baseada na linguagem

de configuração definida na gramática 4.1, sofrendo pequenas alterações. A reconfiguração é definida pelo símbolo não terminal ***reconfig***, o qual declara um cabeçalho (*header*), um conjunto de componentes aninhados (*inner*) e um conjunto de unidades (*unit*). O cabeçalho indica o nome do componente alvo a ser reconfigurado pela plataforma através do terminal targetConfigId. As variáveis InnerId e *paramType* assumem um significado levemente diferente na reconfiguração. Deverão ser descritas pelo innerId apenas novos componentes aninhados a ser inseridos ou componentes aninhados que sofrerão modificação. Estas alterações estarão detalhadas no não-terminal *inner*. *paramType* segue o mesmo raciocínio, sendo explicitado apenas os parâmetros que terão o seu valor alterado. Caso deseje-se suprir algum, poderá ser atribuído valor *null* (nulo) para o respectivo *instTypeNoVar*.

Como adiantamos, *inner* deverá citar apenas os componentes aninhados que sofrerão alteração em sintonia com os InnerId.

Unidades (*unit*) também só deverão ser declaradas caso haja alteração no comportamento de suas ações, condições ou validação. Caso haja necessidade de inclusão de novas fatias, esta operação também poderá ser realizada. A variável *unit* dispensa explicações tendo em vista que o seu uso é exatamente como na descrição de uma configuração, com exceção de que não há necessidade de declarar as fatias já pertencentes ao conector original.

A novidade fica por conta da *action*. Além de declarar o identificador da ação a ser alterada, o programador opcionalmente poderá indicar um ponto de reconfiguração (reconfigPointId). Este ponto serve para alterar parcialmente o comportamento desta ação. Uma vez indicado o ponto de reconfiguração, o protocolo descrito na variável *protocol* irá substituir a respectiva chamada a ação primitiva ou combinador, indicado pelo reconfigPointId. Caso a *protocol* seja nula, entende-se que a ação indicada pelo ponto de reconfiguração deverá ser removida.

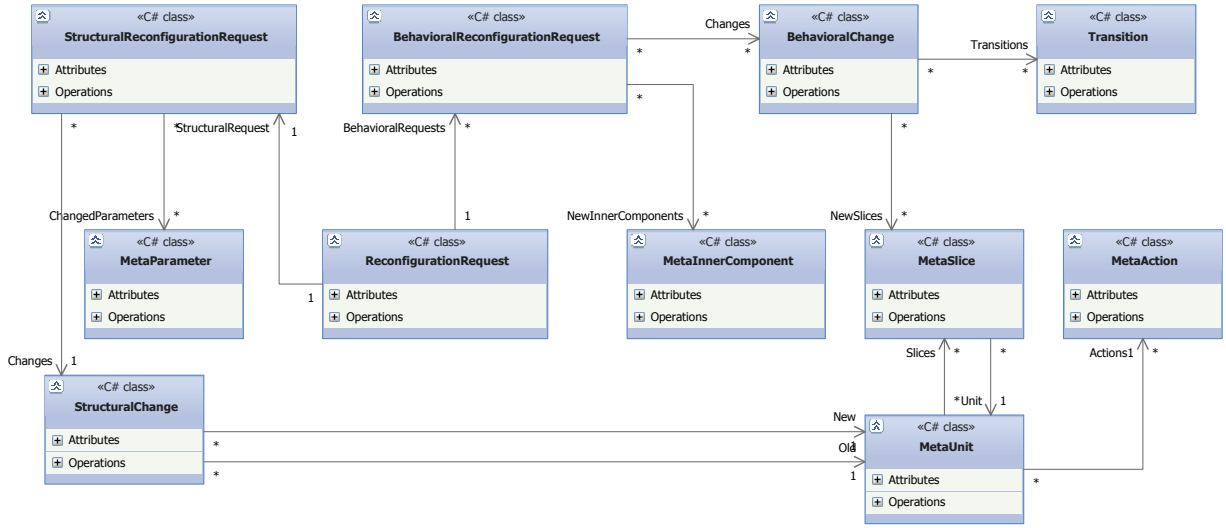
Com todos os recursos oferecidos por esta sintaxe, é possível a realização, mediante *script*, de qualquer uma dos três tipos de reconfiguração, a saber: *reconfiguração comportamental*, *reconfiguração estrutural explícita* e *reconfiguração estrutural implícita*.

Seja qual for o tipo de reconfiguração a ser realizada, o procedimento será o mesmo. Considerando que o conector exógeno já encontra-se em execução, o programador deverá submeter o seu *script* ao HPE através do método EvaluateReconfiguration da porta ConfigurationPort, a qual, como vimos, atende a

interface IConfigurationPort exibida na Figura 4.4. Após a recepção da requisição, similarmente ao que ocorre com as configurações, os *scripts* de reconfiguração passam por um processo de tradução da HCL para a linguagem intermediária.

Novamente o XmlLoader assume a responsabilidade de gerar os metadados a partir do linguagem intermediária. Os metadados são traduzidos em classes que descrevem a reconfiguração, as quais são apresentadas na Figura 4.16 por um diagrama UML.

Figura 4.16: Diagrama das Classes de Metadados



Fonte: Próprio autor.

Cada requisição de reconfiguração é instanciada como um objeto da classe ReconfigurationRequest. Nessa requisição, pode opcionalmente haver uma requisição de mudança estrutural e, também opcionalmente, várias requisições de mudanças comportamentais. A requisição de mudança estrutural é representada pela classe StructuralReconfiguraionRequest. Essa classe representa apenas as reconfigurações estruturais implícitas. Conforme vimos, as reconfigurações estruturais explícitas só trazem real benefício quando associadas à uma reconfiguração comportamental. Assim, as duas serão tratadas conjuntamente pela classe BehavioralReconfigurationRequest.

O XmlLoader irá criar uma StructuralReconfiguraionRequest sempre que houver reescrita de um ou mais parâmetros de contexto, os quais serão instanciados na forma de MetaParameter e associados ao atributo ChangedParameter. A instanciação

das StructuralChange ocorrerá num segundo momento.

Por sua vez, instâncias do BehavioralReconfigurationRequest serão criadas mediante a reescrita de ações, podendo incluir novos componentes aninhados, representados pelo relacionamento com a classe MetaInnerComponent. Será criada uma nova instância para cada unidade que sofrer alteração, sendo uma instância da classe BehavioralChange para cada ação modificada nesta unidade. Esta última, irá descrever o trecho do protocolo da ação a ser alterada, através da propriedade Transitions, além de incluir as novas fatias que são citadas nesta ação, apresentada através do relacionamento com a classe MetaSlice.

Uma vez carregada a requisição, é iniciado o processo de geração das StructuralChange. Cada instância de StructuralChange corresponde ao efeito causado sobre um componente aninhado pela substituição dos parâmetros de contexto sobre o componente #. O processo de geração consiste em avaliar se o novo conjunto de parâmetros resulta em um componente concreto diferente do atual mediante a avaliação do HTS. Para cada caso em que isso ocorra, uma instância de StructuralChange é criada relacionando-a a um par de MetaUnit onde o primeira, indicado por *old*, é a unidade a ser removida, enquanto que a segunda, indicada por *new*, é a sua substituta.

Ao finalizar este procedimento, todas as informações da requisição estarão presentes nos metadados, restando apenas a validação e verificação de segurança.

Assim, o próximo passo é submeter a requisição ao SecurityAnalyzer. Os critérios utilizados nesta validação serão expostos na Seção 4.3.3. Assumindo que a requisição seja aprovada, o SecurityAnalyzer irá retornar um conjunto de avaliações na forma da classe ExecutionStateEvaluation, as quais listam os possíveis estados do autômato onde a execução deverá ser suspensa para que a reconfiguração ocorra com segurança.

Essas informações são então passadas ao programador, de modo a solicitar a confirmação ou suspensão da reconfiguração, o qual informa a sua decisão através dos métodos CommitReconfiguration e CancelReconfiguration, respectivamente. Em caso de cancelamento, o processo de reconfiguração é suspenso e o ConfigurationManager volta a estar apto a receber novas requisições de reconfiguração. Em caso de confirmação, esta mesma classe inicia o processo de modificação do conector mediante a suspensão da execução dos estados listados na ExecutionStateEvaluation.

Uma vez que a execução alcança um estado desejado, o ConfigurationManager procede com as substituições das unidades, como também a reescrita das transições do autômato alterado. No primeiro caso, esse gerenciador faz uso da porta

BuilderService para realizar a desconexão dos componentes antigos e conexão dos novos. Conforme veremos, o SecurityAnalyzer garante que essa substituição não atinja componentes que possuam estado interno observável.

Ao final de todo o procedimento, os estados do autômato que estavam em suspensão são novamente ativados para execução e o ConfigurationManager volta a estar apto a receber novas requisições de reconfiguração.

Herança Funcional

Suponha que o programador realize frequentemente a mesma reconfiguração em um determinado componente. Apesar dessa frequente reconfiguração, o componente não pode ser alterado em definitivo, seja pela ausência de permissão sobre alteração do código-fonte, ou por seu uso original em outras aplicações.

Ao invés de replicar todo o código-fonte da configuração, alterando apenas os pontos de mudança, a HCL será capaz de realizar *herança funcional*.

Em função da popularidade de linguagens como C++, Java e C#, tende-se a compreender o conceito de herança de forma limitada. Apesar de ser a mais difundida, a *herança de subtipo* empregada nessas linguagens não é a única forma de utilizar esta técnica. Sejam A e B abstrações dos conjuntos A' e B'. Dizemos que a relação B herda de A é uma herança de subtipo se B' é um subconjunto de A' e o conjunto modelado por qualquer outro subtipo que herde de A é disjunto de B' [53].

Na prática, uma das consequências da herança de subtipos é a necessidade de manutenção de compatibilidade de interfaces entre A e B. Todas as formas de interação externas de A devem ser mantidas nos seus subtipos.

Na *herança funcional*, essa suposição não é mais verdadeira. Nessa classe de herança, se B herda de A, B tem a liberdade de alterar tanto o comportamento quanto as interfaces herdadas de A.

No HPE, a herança entre componentes abstratos segue a herança de subtipos. Já para os componentes concretos, não era previsto pelo modelo nenhum tipo de herança, apenas a relação de implementação de componentes abstratos. Agora, iremos permitir a realização de heranças funcionais para derivação tanto de componentes abstratos quanto concretos. Esta é a estratégia utilizada para a realização de *mudanças estáticas*.

As mudanças estáticas não irão fazer uso do *script* de reconfiguração, mas sim da sintaxe de escrita da configuração.

4.3.3 Segurança

Conforme explicamos na Seção 2.5, as operações de reconfiguração dinâmica precisam garantir a consistência das aplicações. Esta garantia é atingida mediante a manutenção da integridade estrutural, garantia da preservação dos estados mútuos de consistência e a manutenção das invariantes do estado da aplicação.

Uma vez que o modelo HASH concentra os estados da aplicação em componentes com estado interno, os quais, mediante restrições impostas pelo mecanismo proposto, não podem sofrer reconfiguração, conseguimos atender de imediato as restrições quanto a manutenção da consistência dos estados mútuos e das invariantes da aplicação. Assim, resta aos mecanismo de segurança apenas certificar que a integridade estrutural não será negligenciada. Isso, conforme veremos adiante, é obtido através da correta identificação do momento em que a reconfiguração poderá ser aplicada, somada as restrições de tipos impostas pelo HTS.

Desta forma, independentemente do tipo de reconfiguração a ser realizado, o modelo fará uma avaliação de sua confiabilidade para o sistema através da classe SecurityAnalyzer. Essas verificações buscam identificar se a nova configuração mantém o sistema em um estado válido.

A avaliação de segurança de reconfigurações no HPE assume algumas premissas do modelo. Conforme definido, apenas componentes sem estado interno podem ser substituídos na reconfiguração. Essa premissa nos garante que as substituições de componentes não irão afetar os estados da execução, uma vez que estes estados, por definição, não estão presentes nesses componentes.

A avaliação de segurança corresponde às seguintes etapas:

- Verificação estática da consistência da configuração;
- Avaliação dinâmica dos estados do autômato que devem ter a execução suspensa;
- Consolidar os estados a suspender de forma a garantir a ausência de *deadlock* no processo de reconfiguração.

A primeira avaliação corresponde a uma análise estática que busca identificar se a configuração resultante é válida. São realizadas três verificações: a certificação que, caso existam, os parâmetros informados através da reconfiguração estrutural implícita são de tipos válidos, a certificação que a reconfiguração estrutural implícita não resulta em substituição de componentes com estado interno e a

certificação de que as fatias estão sendo ativadas corretamente, segundo o seu próprio *validationprotocol*.

Na segunda etapa o SecurityAnalyzer irá realizar uma varredura no autômato buscando identificar os estados críticos. São considerados estados críticos aqueles que antecedem a execução de uma ação que faz referência a uma fatia a ser removida ou faz referência a uma fatia que, por sua vez, possui uma fatia, em qualquer nível, a ser removida. Quanto a reconfiguração comportamental, também são considerados críticos todos os estados que são concomitantemente, imediatamente posteriores ao início e imediatamente predecessores ao fim de uma ação a ser reconfigurada. Como já adiantamos, esta avaliação é formalizada através de uma instância da classe ExecutionStateEvaluation, a qual será utilizada para a última etapa de avaliação segurança.

A terceira e última etapa consiste em consolidar os estados críticos descritos na ExecutionStateEvaluation de modo a evitar a ocorrência de *deadlocks*. Uma mesma requisição de reconfiguração que, por exemplo, substitua dois componentes distintos através da reconfiguração estrutural implícita, irá avaliar os estados a suspender de forma independente para cada componente. Ao final das avaliações, o resultado conjunto poderá resulta em *deadlock*. Assim, esta etapa tem o papel de consolidar essa informação de forma a sugerir a suspensão de estados que atendam aos interesses de todas as reconfigurações avaliadas, sem entretanto causar o eterno bloqueio da aplicação.

Capítulo 5

Estudo de Caso: Configuração e Reconfiguração de aplicações do *NAS Benchmark*

Este capítulo tem o propósito de apresentar um estudo de caso para validação do protótipo de interpretação e reconfiguração proposto nesta dissertação, o qual tem os seguintes objetivos:

- Atestar a viabilidade funcional da solução, apresentando o funcionamento prático do protótipo proposto;
- Mensurar a sobrecarga da execução pelo interpretador de protocolos, através da comparação entre um conector e sua versão de execução nativa;
- Demonstrar o funcionamento das reconfigurações dinâmicas estruturais e comportamentais mais importantes.

Para realização dos experimentos, iremos fazer uso da implementação de duas aplicações simuladas do *NAS Parallel Benchmark* (NPB) [9], chamadas de BT e SP, sobre a plataforma de componentes HPE, as quais serão apresentadas em detalhes na Seção 5.1. Na Seção 5.2, é apresentada uma breve explicação do processo de componentização das aplicações SP e BT para o modelo HASH. Na Seção 5.3, são apresentadas refatorações realizadas neste trabalho para transformar alguns componentes de SP e BT em conectores exógenos, para os fins dos estudos de caso. Na Seção 5.4, os estudos de caso são apresentados, bem como a metodologia dos

experimentos realizados. Finalmente, os resultados e discussões referentes ao estudo de caso são apresentados na Seção 5.5.

5.1 *NAS Parallel Benchmarks (NPB)*

O NPB é um conjunto de *benchmarks* e aplicações simuladas desenvolvidos pela NASA (*National Aeronautics and Space Administration*)[1], a partir de códigos desenvolvidos pela sua divisão NAS (*Numerical Aerodynamic Simulation*), para avaliar o desempenho de "supercomputadores" em aplicações de dinâmica dos fluidos [9], de seu interesse específico mas também de amplo interesse no meio industrial.

No início dos anos 1990, os pesquisadores da NASA acreditavam que os pacotes de *benchmarks* disponíveis até então eram inapropriados para a avaliação de computadores paralelos de alto desempenho, por restringirem o uso de instruções específicas da arquitetura alvo e por não avaliar satisfatoriamente todos os recursos de memória e computação disponíveis. Considerando que cada máquina paralela, baseada em sua arquitetura de *hardware*, tem uma abordagem diferenciada para implementar eficientemente cada algoritmo, um *benchmark* deveria possuir as seguintes premissas [9]:

- Ser genérico e não favorecer nenhum arquitetura paralela em particular;
- Possuir fácil verificação da corretude dos resultados e da verificação do seu desempenho;
- Ser facilmente ajustável quanto ao tamanho da instância de execução em relação à memória e volume de computação.

Buscando atender a esses requisitos, o NPB foi definido como um *benchmark* apresentado por uma especificação *papel-e-lápis*[2]. Essa abordagem defende que os problemas sejam especificados na forma de algoritmos, sem oferecer uma implementação padrão. Isso possibilita que cada máquina defina a melhor implementação diante de sua arquitetura de *hardware*, mediante o cumprimento de regras estabelecidas pela especificação do NPB. A entrada de dados do problema é definida *em papel*, no sentido de que nenhum tipo de estrutura de dados é imposta.

O NPB define oito problemas, sendo cinco *kernels* e três *aplicações simuladas*, os quais são resumidamente descritas a seguir [9]:

[1] A agência espacial dos Estados Unidos.
[2] Do inglês *paper and pencil benchmark.*

- EP: Problema chamado de *embaraçosamente paralelo* onde se busca realizar testes pseudo-randômicos com inteiros. Este é o único problema que não demanda comunicação entre processos.
- MG: Problema de *multigrid* que avalia a comunicação de curta e longa distância na máquina paralela.
- CG: Problema do método dos *gradientes conjugados*, utilizado com o objetivo de computar uma aproximação do menor auto-valor de uma grande matriz, esparsa, simétrica e definida positiva, sendo utilizado para avaliar a comunicação irregular de longa distância.
- FT: Problema que aplica a *Transformada Rápida de Fourier* para solução de equações diferenciais parciais.
- IS: Problema de *ordenação de inteiros*, com o objetivo de aferir o desempenho de operações com esse tipo de dado.
- LU: Problema para solução de sistemas triangulares resultantes de fatoração *lower-and-upper*;
- SP: Problema para solução de múltiplos sistemas independentes de equações, descritos por matrizes esparsas pentadiagonais;
- BT: Problema para solução de múltiplos sistemas independentes de equações, descritos por matrizes esparsas bloco-tridiagonais.

Por mais que os cinco primeiros problemas apresentem alto grau de intensidade computacional, eles são insuficientes para aferir o potencial desempenho de uma máquina paralela em uma aplicação científica real [9]. As implementações individualizadas de cada problema definem, por exemplo, as estruturas de dados de acordo com a visão restrita daquela operação. Em aplicações reais, vários tipos de algoritmos diferentes são potencialmente aplicados sobre os mesmos dados, de forma que o desempenho não pode ser diretamente comparado com as execuções individuais. Para tornar a avaliação mais próxima desses cenários, foram propostos os últimos três problemas que se referem a implementação de trechos de algoritmos de aplicações de computação dinâmica dos fluidos, constituindo um rigoroso teste de usabilidade de sistemas paralelos.

Depois de especificado, a NASA realizou inicialmente duas implementações do *benchmarck*, sendo uma delas em Fortran e outra em C. Anos mais tarde, uma nova implementação foi disponibilizada, dessa vez em Java [28].

5.2 Refatoração do *NPB* em Componentes

Buscando avaliar o desempenho da plataforma de componentes paralelos HPE, recentemente foi realizado um trabalho de refatoração do código NPB para o modelo HASH [25]. Considerando que atualmente o HPE oferece suporte apenas para a linguagem C#, foi necessário a criação de uma implementação do *benchmark* nessa linguagem antes da componentização.

Para realizar a implementação em C#, foi utilizada uma estratégia divida em três etapas. Na etapa inicial, diante da similaridade sintática, o código da implementação sequencial Java passou por uma tradução para o C#. Em posse desse código, foi estudada a estratégia de paralelismo por passagem de mensagem da versão Fortran para incorporação na nova versão C#. Por fim, a última etapa buscou otimizar o código, principalmente no sentido de reutilizar trechos comuns das diversas aplicações.

A versão baseada em componentes em C# foi implementada apenas para três aplicações simuladas (LU, SP e BT) e o *kernel* FT, uma vez que apenas essas quatro foram consideradas importantes para o trabalho realizado [25].

A Tabela 5.1 apresenta a quantidade de componentes gerados para cada aplicação, como também os seus compartilhamentos, resultantes de reuso.

Como pode ser visto, as aplicações SP e BT possuem um significativo grau de compartilhamento de componentes, resultante da similaridade de suas aplicações, as quais serão detalhadas adiante. Por esse motivo, essas duas aplicações foram escolhidas para o nosso estudo de caso, tendo em vista serem mais adequadas a aplicação de processos de reconfiguração não triviais. Em especial, é possível a troca dinâmica de resolvedores entre as duas aplicações durante a execução pela simples alteração do contexto de execução, definido pelos parâmetros de contexto efetivos aplicados ao componente da espécie aplicação.

5.2.1 SP e BT

As aplicações SP e BT resolvem três sistemas de equações sobre os eixos x, y e z, através do método ADI (*alternating direct implicit method*). Essencialmente, as duas aplicações têm o mesmo comportamento, com exceção de que SP trata um sistema pentadiagonal enquanto que BT trata um sistema bloco-tridiagonal. Detalhes sobre

Tabela 5.1: Quantidade de componentes fatorados e compartilhados.

Espécies	**SP**	**BT**	**LU**	**FT**	**(1)**	**(2)**	**(3)**
Componentes Abstratos							
Computação	16	24	13	14	11	-	-
Sincronização	3	3	2	1	3	1	1
Qualificador	16	15	21	17	14	7	7
Estrutura de Dados	4	4	5	2	4	-	-
Ambiente	7	7	8	9	5	3	3
Total	**46**	**53**	**49**	**43**	**37**	**11**	**11**
Componentes Concretos							
Computação	27	37	15	28	10	-	-
Sincronização	4	4	11	2	4	2	2
Estrutura de Dados	4	4	5	8	4	2	2
Ambiente	10	10	9	5	5	3	3

(1) Compartilhamento entre **SP** e **BT**;
(2) Compartilhamento entre **SP**, **BT** e **LU**;
(3) Compartilhamento entre **SP**, **BT**, **LU** e **FT**.
Fonte: [25].

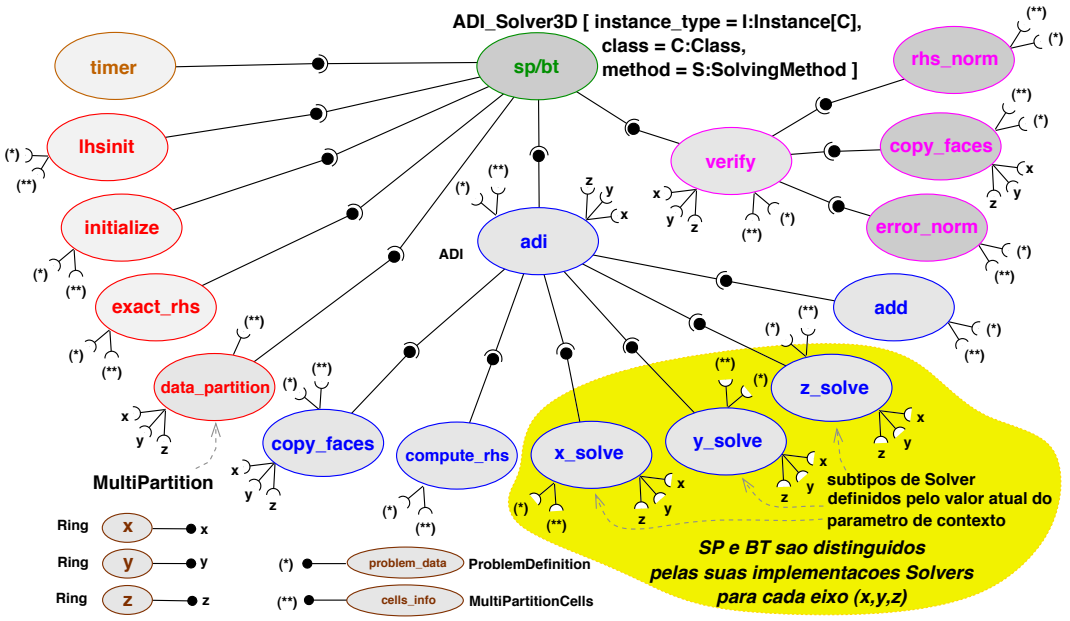
os métodos de solução das aplicações podem ser encontrados em [8].

Durante o processo de componentização, o componente abstrato ADI_Solver3D foi designado para representar as aplicações SP e BT, conforme apresentado na Figura 5.1, onde as elipses correspondem aos componentes e os traços representam a relação de aninhamento entre eles, onde o componente ligado a extremidade com circulo preenchido é considerando componente aninhado do componente ligado a outra extremidade (relações cliente-servidor, ou de dependência entre componentes).

Considerando a já ressaltada similaridade entre as aplicações, ADI_Solver3D foi modularizado de forma a isolar, através de seus parâmetros de contexto, as divergências entre os métodos SP e BT, respeitando a divisão dos interesses de acordo com as espécies de componentes do HPE. Nesse contexto, dentre os oito componentes aninhados de ADI_Solver3D, as diferenças entre SP e BT estão concentradas em apenas um componente chamado *adi*, definido pelo componente abstrato ADI.

O componente abstrato ADI, por sua vez, é definido através de cinco componentes aninhados, dos quais três correspondem aos resolvedores *x_solve*, *y_solve* e *z_solve*, todos do tipo Solver. Isso quer dizer que existem diferentes implementações de componentes concretos para o componente abstrato Solver, onde

Figura 5.1: Arquitetura SP/BT



Fonte: [25].

cada implementação traz consigo as especificidades referentes à SP e BT, como também as especificidades referentes a cada eixo.

Considerando o funcionamento do HTS, ao descrever um conector ADI_Solver3D, as escolhas das implementações concretas serão realizadas mediante a avaliação dos parâmetros de contexto *method*, *class* e *instance_type*. O parâmetro *method* do tipo SolvingMethod determina a escolha entre BT e SP através dos sub-tipos BTSolvingMethod e SPSolvingMethod, respectivamente.

O parâmetro *class*, do tipo Class, determina a classe do problema. As possíveis classes de problemas são Class_S (teste), Class_W, Class_A, Class_B e Class_C, as quais definem o tamanho do problema mediante o número de interações, as constantes numéricas para a validação do resultado, entre outras.

O parâmetro *instance_type*, limitado pelo componente Instance determina os valores de parâmetros da aplicação para uma determinada classe de problema.

Dessa forma, no momento da instanciação de um ADI_Solver3D, através da escolha adequada de parâmetros de contexto, pode-se definir o método de solução (BT ou SP), como também o tamanho do problema (Class_*x*). A Tabela 5.2 apresenta dois exemplos onde são apresentadas as instanciações de uma aplicação BT de tamanho W e outra aplicação SP de tamanho B.

Tabela 5.2: Exemplos de Instanciações para ADISolver3D

BT – Classe W	
ADISolver3D [	method=BTSolverMethod, class=Class_W, instance=Instance_BT[Class_W]]
SP – Classe B	
ADISolver3D [	method=SPSolverMethod, class=Class_B, instance=Instance_SP[Class_B]]

Fonte: [25].

Para os propósitos do nosso estudo de caso, não se faz necessário a descrição de detalhes a respeito dos outros componentes aninhados, os quais podem ser conhecidos no trabalho Cenez Rezende [25].

5.3 Extraindo Conectores Exógenos de SP e BT

Para os fins dos estudos de caso apresentados nesta dissertação, foram realizadas refatorações nos componentes SP e BT, com o objetivo de tornar os componentes ADI e Solver conectores exógenos. Em outras palavras, deseja-se que esses componentes tenham as suas ações completamente descritas mediante a combinação das ações das unidades dos seus componentes aninhados por meio de um protocolo da linguagem HCL, que seja interpretável.

A implementação original, além de possuir resolvedores do tipo Solver específicos para cada uma das aplicações BT e SP, exigia ainda implementações específicas para cada eixo (x, y e z) de cada resolvedor. Isso quer dizer, por exemplo, que no caso do SP, são necessárias três implementações distintas, como os resolvedores impl.sp.solve.XSolver, impl.sp.solve.YSolver e impl.sp.solve.ZSolve, cada uma delas específica para cada um dos eixos, variando o valor do parâmetro de contexto *axis* entre common.axis.ZAxis, common.axis.YAxis e common.axis.XAxis, respectivamente. Analogamente, o mesmo acontece para o BT.

O primeiro passo da refatoração consistiu na escrita em C# de novos componentes impl.sp.solve.SolverImpl e impl.bt.solve.SolverImpl, genéricos em relação aos eixos, porém específicos quanto a aplicação. Esse passo objetiva simplificar o processo de escrita dos resolvedores na forma de conectores exógenos, que agora demanda a escrita de apenas dois componentes, ao contrário dos seis necessários na

versão inicial. Esses componentes são chamados de *versões nativas*, pelo fato de que a execução das ações do resolvedor estão implementadas diretamente em C#, ao invés da execução interpretada do protocolo.

A segunda etapa corresponde a descrição das ações dos componentes sp.solve.Solver e bt.solve.Solver através da linguagem HCL, usando todo o poder expressivo dos protocolos descritos no capítulo anterior. O novo código gerado tem o mesmo valor semântico da versão nativa e pode ser executado usando o interpretador de protocolos.

No caso dos conectores, não haveria necessidade da implementação de um componente concreto. Entretanto, por facilidade de implementação do mecanismo de interpretação sobre a plataforma HPE, permitindo o tratamento uniforme de componentes nativos e conectores, sendo intercambiáveis, foi necessário criar os componentes impl.sp.connector.solve.SolverImpl e impl.bt.connector.solve.SolverImpl e submetê-los para implantação no *Back-End*. Porém, estes componentes concretos correpondem apenas a *wrappers* para ligar a plataforma ao mecanismo de interpretação. Os códigos das unidades, em C#, são gerados automaticamente pelo *Back-End*, de forma transparente ao usuário.

5.4 Descrição dos Experimentos e Metodologia

Para atestar o protótipo desenvolvido por este trabalho, foram realizados experimentos para demonstrar a viabilidade funcional da solução, apresentando indícios do seu funcionamento prático a respeito das seguintes capacidades:

- Interpretação de configurações na forma de ações de conectores exógenos;
- Aplicação de reconfiguração dinâmica comportamental sobre os conectores;
- Aplicação de reconfiguração dinâmica estrutural sobre os conectores;

Para realizar tais experimentos, devemos fazer uso de uma ou mais aplicações paralelas que sejam descritas na forma de conectores HCL. Conforme adiantamos, na nossa avaliação, as aplicações SP e BT desenvolvidos para o HPE possuem os requisitos necessários para seu uso neste experimento. Reforçamos abaixo os motivos que nos levam a esta escolha:

- SP e BT são aplicações de computação intensiva, cujo código foi desenvolvido por programadores especialistas no domínio de CAD e em aplicações de dinâmica dos fluidos computacional, de interesse amplo nesse domínio;

- O componente adi.ADI, que implementa o método *Alternating Direct Implicit* (ADI), bem como os componentes resolvedores sp.solve.Solver e bt.solve.Solver, que diferenciam o SP e o BT, podem ser facilmente descritos na forma de conectores exógenos com protocolos de relativa complexidade, como apresentado na Seção 5.3.

- Na versão de componentes HPE, a diferenciação entre SP e BT corresponde a substituição do parâmetro de contexto *method* do componente aplicação adi.ADI_Solver3D, tornando possível demonstrar a troca dinâmica dos resolvedores, de SP para BT e vice-versa.

Diante dessas considerações, apresentaremos nas próximas seções a metodologia e os resultados obtidos nos experimentos realizados.

5.4.1 Metodologia

A descrição da metodologia é dividida em duas partes, onde a primeira define o método de avaliação da interpretação do conector, mensurando numericamente os seus tempos de execução e comparando com as versões nativas, enquanto que a segundo apresenta evidências do funcionamento das reconfigurações dinâmicas comportamentais e estruturais.

Desempenho do Processo de Interpretação

Para a interpretação do conector, a metodologia adotada consiste na comparação de medições de desempenho, na forma de tempo de execução, confrontando os resultados obtidos pelo conector e pela versão nativa (HPE/C#).

Considerando que as versões na forma de conector possuem exatamente o mesmo valor semântico da versão nativa, e ainda que estas utilizam os mesmos componentes aninhados, o experimento busca evidenciar a sobrecarga do processo de interpretação. Naturalmente, quanto maior o esforço realizado pelas computações dos componentes aninhados, menor será a sobrecarga do processo de interpretação, quando avaliamos este indicador de forma proporcional ao tempo total da execução. Por esse motivo, visando a confirmação dessa hipótese, as execuções contemplaram cenários que comparam classes de problemas distintas, ou seja, tamanhos de problemas distintos. A Tabela 5.3 descreve os 12 cenários executados neste experimento.

Conforme pode ser observado, os cenários de 1 a 6 definem semanticamente as mesmas aplicações executadas nos cenários de 7 a 12. Dessa forma, foi possível

Tabela 5.3: Cenários de experimentos para avaliação de desempenho de execução.

EXPERIMENTOS												
Cenário	1	2	3	4	5	6	7	8	9	10	11	12
Classe	S	W	A	S	W	A	S	W	A	S	W	A
Aplicação	BT			SP			BT			SP		
Versão	Nativo						Conector					

Fonte: Próprio autor.

realizar comparações de desempenho onde a única alteração entre as versões correspondeu ao uso do processo de interpretação.

Buscando o enriquecimento dos experimentos, os 12 cenários foram executados em três versões de distribuição, contemplando a execução em uma única unidade de processamento, bem como em 4 e 9 unidades de processamento, respectivamente.

A métrica utilizada na comparação dos resultados é a média dos tempos das execuções obtidos em segundos. Foram realizados 15 execuções de cada cenário, onde a composição dos valores para o cálculo das médias desprezou o conjunto de valores discrepantes (*outliers*) usando o método *Box Plot* [58], a fim de aproximar a amostra de resultados de uma distribuição normal. Assim, foram realizados 15 execuções de duas aplicações simuladas (SP e BT) variando-se a carga de trabalho (S, W e A) e o número de processadores (1, 4 e 9) nas versões nativas e conector, totalizando um total de 540 experimentos distintos.

Reconfiguração Dinâmica

Ao contrário do ensaio anterior, para o processo de reconfiguração dinâmica não há um cenário pré-existente que nos sirva de parâmetro na comparação dos resultados de desempenho. Entretanto, ao avaliar as motivações que expõem a relevância dessa funcionalidade, conseguimos observar que a medição do tempo necessário para a realização do procedimento é uma métrica válida.

Chegamos a essa conclusão ao recordar que uma das principais motivações para a existência da reconfiguração dinâmica é evitar que uma aplicação tenha a necessidade de ser desligada para sofrer uma alteração, uma vez que esse desligamento ocasiona a perda de todo o processamento já realizado. Assim, podemos admitir que todo o processo de reconfiguração que seja capaz de alterar uma aplicação em tempo inferior ao processamento já realizado pode ser considerado como útil.

Entretanto, ainda temos dois grandes dilemas. Em primeiro lugar, obter

ganhos de poucos segundos muito provavelmente não justifica o esforço para o desenvolvimento de um módulo de reconfiguração, de forma que é preciso definir quão melhor deverá ser o desempenho desse processo, ao comparar com a execução já realizada. O nosso segundo dilema está intimamente relacionado com o primeiro. Considerando a dinamicidade dos processos de reconfiguração, não temos como precisar em que momento da execução o processo será iniciado. Mesmo que pudéssemos limitar esse momento, por ser um procedimento capaz de ser aplicado a qualquer aplicação, não conseguiríamos determinar o tempo mínimo que essa execução levaria.

A argumentação acima indica que a avaliação de um procedimento de reconfiguração dinâmica nos moldes propostos só traria reais benefícios se aplicado a um conjunto definido de aplicações onde tais indicadores pudessem ser fielmente definidos.

A busca por uma avaliação rigorosa de um experimento qualquer a fim de certificar uma solução genérica é inviabilizada pelos seguintes aspectos:

- O tamanho da aplicação alvo, com relação à quantidade de componentes envolvidos e com a extensão dos protocolos de ações, determina diretamente o tempo gasto com a avaliação da reconfiguração. Esse procedimento, de complexidade computacional significativa, corresponde a verificação da viabilidade da mudança e a verificação dos instantes em que esta pode ser aplicada;

- O tamanho da mudança a ser realizada pode variar de uma simples substituição de componentes, até uma completa reengenharia algorítmica do protocolo, associada a adição de vários componentes aninhados;

- O estado em que a aplicação se encontra pode exigir que o processo de reconfiguração fique aguardando a finalização de ramos de execução para que possa ser realizada. Por exemplo, a tentativa de substituir um componente que está em uso deverá suspender o processo de reconfiguração até que tal componente esteja inativo. No caso de componentes de execução de longa duração, essa espera pode chegar a horas, impactando diretamente o processo de reconfiguração.

Portanto, aferir o tempo de realização de uma reconfiguração sem uma base comparativa e sem um alvo específico, não definindo a aplicação, arquitetura

e finalidade, nos parece despropositado. Por esse motivo, os experimentos de reconfiguração foram efetuados com o único intuito de demonstrar o real funcionamento da solução exposta.

Nesses moldes, foram realizados dois experimentos de reconfiguração, a partir dos quais foi possível fazer uso das reconfigurações comportamental e estrutural.

O primeiro experimento consiste na inclusão de um componente aninhado e a alteração do protocolo de uma ação aplicados sobre um conector. A alteração do protocolo visa adicionar chamadas as ações do novo componente inserido. Essa reconfiguração termina por assumir características comportamentais, como também estruturais, já que insere um novo elemento na configuração.

Já o segundo experimento corresponde uma reconfiguração estritamente estrutural, baseada na troca de componentes mediante a alteração de parâmetros de contexto.

Experimento 1 (Reconfiguração Comportamental): Neste experimento, buscamos reproduzir uma situação prática. Ao executar aplicações de longa duração, é comum aos programadores incluir no código chamadas a funções que realizam escrita no console com o objetivo de acompanhar o andamento da execução. No nosso estudo de caso, após iniciar a execução da aplicação sp.adi.SP_ADI na forma de conector, adicionamos um componente de *log* ao conector aninhado x_solve, a fim de relatar, em cada iteração, o início e o término de sua execução.

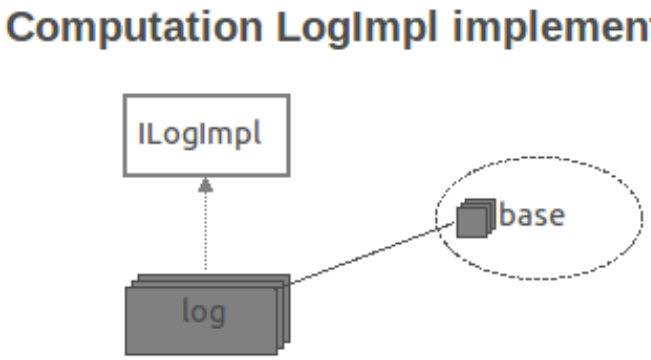
Para isso desenvolvemos, em linguagem nativa C#, um simples componente abstrato log.Log, o qual foi implementado pelo componente concreto impl.log.LogImpl, possuindo uma única unidade paralela chamada log, conforme pode ser visto na Figura 5.2.

A unidade log possui duas ações, definidas por notify_start e notify_end, onde cada uma delas tem o objetivo de escrever na saída padrão uma mensagem confirmando, respectivamente, o início e o fim de algum procedimento.

A reconfiguração visa adicionar o componente log.Log no conector adi.Solver, incluindo uma chamada à notify_start antes da primeira instrução da ação principal do conector e incluindo uma chamada à notify_end depois da última instrução desta mesma ação.

Entendemos que a simplicidade da tarefa do novo componente em nada altera o mérito do experimento, tendo em vista que o procedimento para inclusão de um componente e a adição de chamadas às suas ações, independe da complexidade dos algoritmos que estes descrevem.

Figura 5.2: Componente # concreto impl.log.LogImpl



Fonte: Próprio autor.

Experimento 2 (Reconfiguração Estrutural): Conforme apresentamos na Seção 3.5, as aplicações do modelo HASH são desenvolvidas utilizando um sistema de tipos (HTS) especificamente desenvolvidos para modelos de componentes paralelos, sob premissas de aplicações no contexto de CAD. Segundo sua definição, as aplicações são iniciadas descrevendo os tipos abstratos de componentes que se desejam executar, juntamente com os parâmetros de contexto que definem e restringuem os possíveis tipos de componentes que podem ser executados. Considerando essa característica, a mudança de um parâmetro de contexto de um componente pode gerar a escolha de um componente concreto diferente.

Fazendo uso disso, é possível realizar um procedimento de reconfiguração ao simplesmente trocar o valor efetivo de um parâmetro de contexto. Para isso, os seguintes requisitos devem ser atendidos:

- O componente aplicação possuir pelo menos um parâmetro de contexto;
- Um ou mais componentes transitivamente aninhados ao componente aplicação ser determinados por um ou mais de seus parâmetros de contexto;
- Existir componentes concretos diferentes, para variações viáveis do parâmetro de contexto efetivo, implantadas no *Back-End*;
- Tais componentes concretos não devem possuir estado, o que é verdade para todas as espécies de componentes com exceção de *estruturas de dados*.

Conforme exemplificado na Tabela 5.2, a aplicação adi.ADI_Solver3D possui todas essas características. Possui três parâmetros de contexto, um dos quais,

chamado *method*, define a instância concreta do componente aninhado **adi**, o qual de forma transitiva define as instâncias concretas dos componentes aninhados **x_solver**, **y_solver** e **z_solver** que também depende deste mesmo parâmetro.

Dessa foma, o nosso cenário de reconfiguração estrutural implícita consistiu da inicialização do SP na classe de problema A, através da execução de um componente adi.ADI_Solver3D com o parâmetro *method* associado ao valor common.method.SPMethod. Após algum tempo de execução, iniciamos o processo de reconfiguração, substituindo o valor do parâmetro *method* pelo valor common.method.BTMethod. Com isso, a execução é interrompida em um ponto seguro e o algoritmo de resolução de componentes é chamado, instanciando os devidos componentes concretos dos tipos adi.ADI e adi.Solver para assumir o lugar dos componentes aninhados **adi**, **x_solver**, **y_solver** e **z_solver**. Os componentes que encapsulam as estruturas de dados processadas pelos resolvedores não são modificadas, por não dependerem do parâmetro *method*.

5.4.2 Plataforma de Execução do Experimento

Os experimentos foram executados no *Castanhão*, *Cluster* do Departamento de Computação da Universidade Federal de Ceará. O Castanhão possui 16 nós de execução, cada um deles com as seguintes características:

- Dois processadores *Intel Xeon*, 1.8 GHz, 512 KB de memória *cache*;
- 1 GB de memória RAM;
- Duas interfaces de redes, sendo uma *Fast Ethernet* (100Mbps) e uma *Gigabit Ethernet* (1 Gbps).

O *Cluster* é gerenciado pelo sistema operacional Rocks 5.0 (Linux 2.6.18). Para suporte ao HPE, estão instalados o Mono na versão 2.2 e biblioteca de troca de mensagem MPICH2 na versão 1.2.1.

5.5 Resultados e Discussões

Apresentamos abaixo os resultados e discussões a respeito dos experimentos realizados.

5.5.1 Processo de Interpretação

Ao iniciar este projeto, como primeira suposição, esperávamos que o desempenho do conector fosse significativamente inferior à versão nativa em qualquer dos

cenários, embora o peso da interpretação seja constante e independente da carga de trabalho útil da execução. Esse raciocínio parte da analogia bastante discutida entre linguagens compiladas *versus* linguagens interpretadas. Entretanto, a execução dos experimentos não confirmou esta suspeita em todos os casos experimentais testados.

As tabelas 5.4, 5.5, 5.6 apresentam, respectivamente, os resultados dos experimentos executados sobre os problemas de classe S, W e A, associados a cargas de trabalho útil de tamanho crescente em ordem de magnitude. Cada uma das colunas, a partir da segunda, representa a medição em segundos de um cenário de execução, onde os as colunas **1**, **4** e **9** indicam a quantidade de processadores no experimento distribuído, enquanto que as colunas **N** e **C** referem-se respectivamente à versão **nativa** e **conector**.

As tabelas mostram apenas os resultados sumarizados, onde os símbolos $\overline{X}$ e δ, apresentam a média aritmética e o desvio padrão de cada uma colunas, as quais, conforme metodologia definida na Seção 5.4.1, desconsidera as valores discrepantes citados anteriormente. A última linha apresenta a razão entre as médias dos tempos de execução das versões nativa e conector, representado por C/N(%).

De modo geral, as execuções dos experimentos se mostraram muito regulares, conforme pode ser constatada pela baixa variância das amostras, as quais não excedem a 4% em nenhum dos casos e em mais de 90% dos cenários mostrou-se inferior a 1%.

Avaliando os dados, podemos inferir três grandes indícios:

- O processo de interpretação de conectores tende a exigir um esforço computacional superior à execução de um componente nativo;
- A inclusão do paralelismo influência negativamente o custo de interpretação de forma desproporcional à execução em um nó de processamento;
- Os resultados sugerem que o crescimento da carga de trabalho útil amortiza a sobrecarga computacional em relação ao tempo total.

Tabela 5.4: Resultados dos experimentos para a Classe S

Classe S												
	SP						**BT**					
	1		**4**		**9**		**1**		**4**		**9**	
Exp.	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**
$\overline{X}$	5,45	7,20	1,98	3,83	1,50	4,19	22,18	22,78	6,40	36,11	4,06	20,76
δ	1,34%	0,77%	0,26%	3,00%	0,37%	3,95%	0,19%	0,26%	0,25%	0,46%	0,32%	0,80%
C/N(%)	1,32		1,94		2,80		1,03		5,64		5,11	

Fonte: Próprio autor.

Tabela 5.5: Resultados dos experimentos para a Classe W

Classe W												
	SP						**BT**					
	1		**4**		**9**		**1**		**4**		**9**	
Exp.	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**
$\overline{X}$	844,18	846,98	228,90	245,11	104,27	128,66	716,03	718,80	193,04	194,37	100,25	101,13
δ	0,28%	0,28%	0,66%	0,22%	0,05%	0,68%	0,91%	0,23%	0,73%	0,26%	0,15%	0,38%
C/N(%)	1,00		1,07		1,23		1,00		1,01		1,01	

Fonte: Próprio autor.

Tabela 5.6: Resultados dos experimentos para a Classe A

Classe A												
	SP						**BT**					
	1		**4**		**9**		**1**		**4**		**9**	
Exp.	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**	**N**	**C**
$\overline{X}$	4944,14	5179,22	1337,29	1349,67	640,41	645,17	15221,36	15513,79	3979,14	3986,25	1859,65	1842,69
δ	0,62%	0,23%	0,58%	0,13%	0,40%	0,10%	0,66%	0,19%	0,75%	0,25%	0,38%	0,12%
C/N(%)	1,05		1,01		1,01		1,02		1,00		0,99	

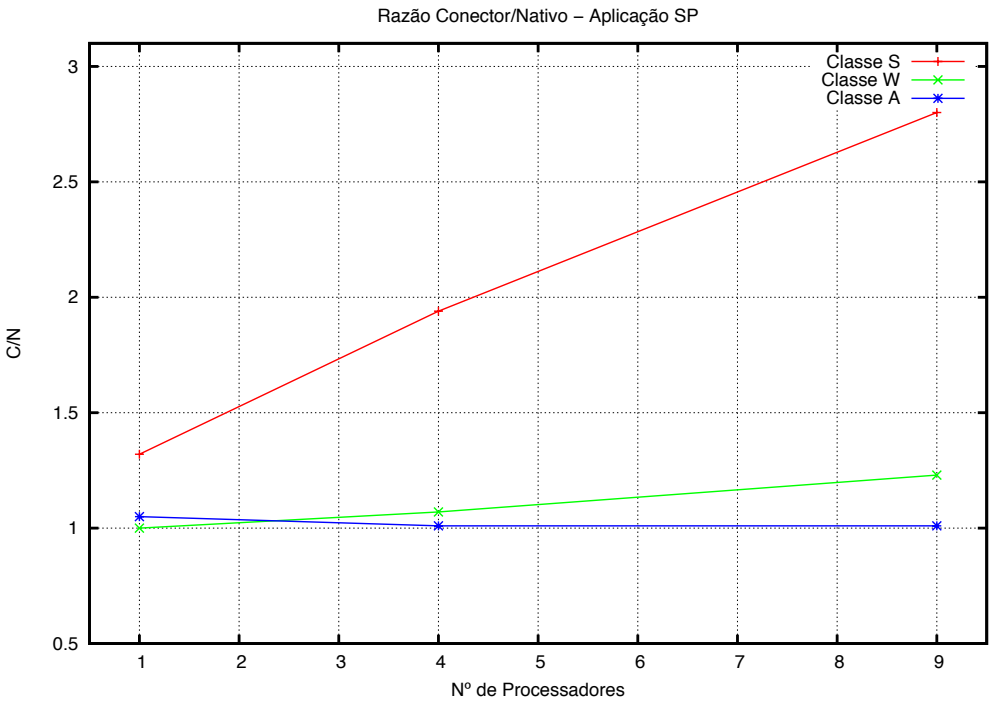
Fonte: Próprio autor.

A rigor, com exceção de um cenário, em todos os experimentos executados os custos de interpretação foram superiores à execução nativa, resultado este já esperando, tendo em vista a sobrecarga da execução do módulo de interpretação. Entretanto, em alguns experimentos esse custo adicional, para nossa surpresa, mostrou-se extremamente baixo a ponto de ser desprezível, como também negativo, onde a execução interpretada foi mais eficiente que a nativa. Estes casos podem ser observados nos cenários S.BT.1, W.SP.1 e em todos os W.BT e em praticamente todos os A. Atribuímos este último fenômeno a possíveis otimizações internas da máquina virtual que automaticamente aplica melhoria em alguns padrões de códigos os quais contra-intuitivamente sem mostram mais desvantajosos. Entretanto, garantir esta motivação ao problema demanda uma investigação profunda, a qual foge do escopo deste trabalho.

Verificamos que o crescimento do custo relativo não é constante ao compararmos os experimentos da versão sequencial (1 nó de processamento) com as versões distribuídas (4 e 9 nós de processamento). Para justificar este fenômeno, acreditamos que, ao aumentar o número de processadores, dividimos a carga útil entre os nós, porém mantemos fixo, em média, o custo de interpretação de forma que o processamento útil relativo é diminuído. Por exemplo, suponhamos que a aplicação demande, em relação a carga útil, a execução de x operações, e a sobrecarga de execução acresça outras y operações. Ao realizar a execução distribuída, dividimos o número de operações x entre os n nós de processamento. Devemos ainda, no cenário de um algoritmo paralelo, adicionar o número de operações referente a especificidades desta versão, como por exemplo as operações de comunicação, as quais referenciaremos por α. Assim, a carga útil por nó passa a ser $(x/n) + \alpha$. Considerando que o mesmo interpretador será executado em todos os nós, este custo permanece constante. Assim, admitindo x, y, n e α positivos e α pequeno, naturalmente $(y + x)/x < (y + (x/n) + \alpha)/(x/n) + \alpha$. A ausência de um comportamento crescente nos dados do BT, acompanhando o acréscimo do número de processadores similarmente ao que no SP, pode ser justificado pela variação do fator α, porém essa hipótese não é facilmente identificada pelos dados, sendo necessária uma maior investigação.

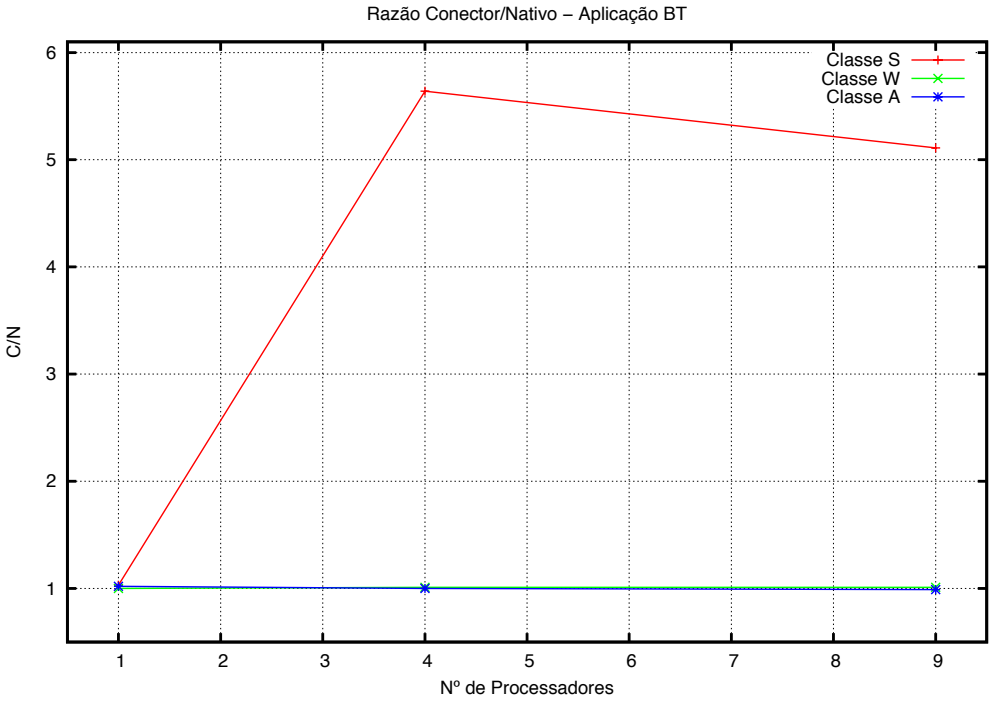
Também conforme esperávamos, o crescimento do tamanho do problema amortiza o custo de interpretação em relação ao processamento útil, mediante podemos constatar ao comparar os resultados apresentados entre os experimentos sobre as classes S, W e A. Os gráficos exibidos nas Figuras 5.3 e 5.4 facilitam

Figura 5.3: Gráfico de Resultados para a Aplicação SP



Fonte: Próprio autor.

Figura 5.4: Gráfico de Resultados para a Aplicação BT



Fonte: Próprio autor.

Figura 5.5: Trechos inicial e final da ação main do conector impl.sp.solve.connector.SolverImpl.

```
action main
begin
    SEQ
    begin
        START : forward.begin
        lhs.begin
        read_buffer_forward.begin

(..... ↑ trecho inicial ↑ .....)

        ... várias outras instruções ...

(..... ↓ trecho final ↓ .....)

                backward.go
                backward.advance
        end

        END : matvecproduct.go
    end
end
```

Fonte: Próprio autor.

essa observação. As amostras SP apresentam resultados bem comportados de forma que a compreensão dos números são plenamente justificados pelas hipóteses já apresentadas. A aplicação BT, similarmente ao caso anterior, demanda maior investigação para sua compreensão, de forma que não conseguimos identificar uma hipótese para justificar a brusca redução dos tempos de execução das versões paralelas nas classes W e A. O comportamento dos resultados das duas últimas classes sugere que a partir de determinado esforço computacional útil, o custo de interpretação torna-se desprezível.

5.5.2 Reconfiguração Dinâmica Comportamental

Conforme apresentamos anteriormente, a reconfiguração proposta por este estudo de caso visa incluir um componente com capacidade de *log* em um conector adi.Solver, seguido da realização de chamadas a esse componente no início e no término da execução. Para tal, inicialmente identificamos as instruções que devem sofrer modificação. A Figura 5.5 apresenta os fragmentos inicial e final da ação main do componente impl.sp.solve.connector.SolverImpl.[3].

O protocolo da ação main é iniciado por um combinador seq, que por sua

[3]A descrição completa da ação pode ser vista no Apêndice A.

vez, tem como primeira instrução a chamada forward.begin, a qual nomeamos de start. No final, a instrução matvecproduct.go, a qual nomeamos de end, finaliza o procedimento.

De acordo com a estratégia de reconfiguração proposta na Seção 4.3.2, a alteração deve ser realizada mediante a elaboração de um *script* de reconfiguração que adicione o novo componente ao conector e o faça realizar chamadas sobre ele. O *script* de reconfiguração escrito em linguagem intermediária para este fim é apresentado na Figura 5.6.

Vale recordar que o componente sp.adi.SP_ADI, que implementa o método ADI para o SP, possui três componentes resolvedores aninhados, chamados **x_solve**, **y_solve** e **z_solve**, cada qual relacionado a uma dimensão (x, y e z, respectivamente). Neste experimento, deseja-se alterar apenas o *solver* relacionado com a dimensão x. Para realizar a reconfiguração, necessitamos identificar exatamente o nome dado pelo plataforma à instância do conector. Assim, foi incluído no *Back-End* uma funcionalidade que permita obter esta informação em tempo de execução através de uma consulta. A Figura 5.7 apresenta a resposta do *Back-End* a uma consulta com a chave "x_solve", durante a execução de um componente sp.adi.SP_ADI.

Uma vez identificado, podemos indicar o nome do componente alvo da reconfiguração através da *tag* targetComponent do *script*. Logo em seguida, descrevemos o componente aninhado que desejamos adicionar, representado na linguagem intermediária pela *tag* complexa innerComponent.

A *tag* changeAction, responsável por definir as alterações comportamentais, possui quatro atributos. Os atributos unit e action indicam, respectivamente, qual a unidade e ação que sofrerá a alteração. O atributo point indica qual a chamada da ação a ser mudada. Neste *script*, os atributos assumem valores que visam a alteração da instrução nomeada por *start* (atributo point) da ação *main* (atributo action) da unidade *solve* (atributo unit). *Start* corresponde a um rotulamento explícito do ponto de reconfiguração, o qual estamos fazendo uso para conceder maior valor semântico ao protocolo. Entretanto, conforme definido na Seção 4.3.2, caso este rótulo não exista, há uma identificação implícita indicada por um inteiro que corresponde a ordem de aparição da instrução no *script*. Nesse caso, o ponto *start* também pode ser referenciado por *1*, já que o combinador SEQ é definido como o ponto *0* na ordem de aparição.

O atributo type indica se a alteração tem por objetivo a remoção da instrução

Figura 5.6: Script de reconfiguração do Solver SP em linguagem intermediária

```
<?xml version="1.0" encoding="UTF-8" ?>
<reconfigurationRequest>

  <targetComponent>app.adi_solver3D-adi-x_solve</targetComponent>

  <innerComponent>
    <kind>computation</kind>
    <identifier>log</identifier>
    <type>
      <componentName>log.Log</componentName>
    </type>
    <access>public</access>
    <exportActions>true</exportActions>
  </innerComponent>

  <changeAction unit="solve" action="main" point="start" type="include">
    <slice inner="log" unit="log"/>
    <protocol>
      <seq>
        <action>
          <perform action_id="notify_start" slice_id="log"/>
        </action>
        <action>
          <perform id="start" repeat="false" action_id="begin" slice_id="forward"/>
        </action>
      </seq>
    </protocol>
  </changeAction>

  <changeAction unit="solve" action="main" point="end" type="include">
    <slice inner="log" unit="log"/>
    <protocol>
      <seq>
        <action>
          <perform id="end" repeat="false" action_id="go" slice_id="matvecproduct"/>
        </action>
        <action>
          <perform action_id="notify_end" slice_id="log"/>
        </action>
      </seq>
    </protocol>
  </changeAction>
</reconfigurationRequest>
```

Fonte: Próprio autor.

Figura 5.7: Resposta do Back-End a uma consulta com a chave 'x_solve'

```
Enter your command:  'select <key>' or 'reconfigure <file-path>'
> select x_solve
Components who name contains 'x_solve'

Component:  app.adi_solver3D-adi-x_solve
Component:  app.adi_solver3D-adi-x_solve-backward
Component:  app.adi_solver3D-adi-x_solve-forward
Component:  app.adi_solver3D-adi-x_solve-input_buffer_backward
Component:  app.adi_solver3D-adi-x_solve-input_buffer_forward
Component:  app.adi_solver3D-adi-x_solve-lhs
Component:  app.adi_solver3D-adi-x_solve-matvecproduct
Component:  app.adi_solver3D-adi-x_solve-mpi
Component:  app.adi_solver3D-adi-x_solve-output_buffer_backward
Component:  app.adi_solver3D-adi-x_solve-output_buffer_forward
Component:  app.adi_solver3D-adi-x_solve-read_buffer_backward
Component:  app.adi_solver3D-adi-x_solve-read_buffer_forward
Component:  app.adi_solver3D-adi-x_solve-shift_backward
Component:  app.adi_solver3D-adi-x_solve-shift_forward
Component:  app.adi_solver3D-adi-x_solve-write_buffer_backward
Component:  app.adi_solver3D-adi-x_solve-write_buffer_forward
Component:  app.adi_solver3D-adi-x_solve-instance
```

Fonte: Próprio autor.

start ou se esta objetiva realizar uma inclusão de uma nova instrução (*include*). A inclusão é melhor definida como uma substituição, uma vez que a instrução rotulada pelo ponto de reconfiguração é removida para dar lugar ao novo protocolo presente no *script* através da *tag* protocol.

A inserção de novas fatias à unidade em reconfiguração fica a cargo da *tag* slice, através da indicação do componente aninhado (atributo inner) e sua unidade (atributo unit) que assumirá o papel de fatia. Uma vez que, tanto o componente aninhado quanto a unidade possuem o nome de log, ambos assumem este mesmo valor no *script*.

A última informação da changeAction é o protocolo que substituirá o anterior. Esse protocolo chama sequecialmente a ação de *log* e em seguida insere novamente a ação pré-existente. Esta inclusão é necessária tendo em vista o caráter substitutivo da ação de reconfiguração do tipo *include*.

A segunda *tag* changeAction é semelhante a primeira, com exceção do protocolo a ser inserido, o qual corresponde a notificação do término da ação.

A Figura 5.8 apresenta a saída da execução da aplicação adi.ADI_Solver3D fazendo uso dos conectores sp.adi.SP_ADI e impl.sp.solve.connector.SolverImpl, sendo este último o alvo a ser reconfigurado.

Figura 5.8: Saída da execução do sp.adi.SP_ADI

```
NAS Parallel Benchmarks C# version (NPB3_0_CS)
Multithreaded Version ADI_Solver3D.S np=1
(XmlLoader.loadComponent) Iniciando carga de arquivo de configuração...
(XmlLoader.loadRequest) Carga realizada com sucesso!
(ConfigurationManager.LoadComponent) Unidade app.adi_solver3D-adi pronta.
(XmlLoader.loadComponent) Iniciando carga de arquivo de configuração...
(XmlLoader.loadRequest) Carga realizada com sucesso!
(ConfigurationManager.LoadComponent) Unidade app.adi_solver3D-adi-x_solve pronta.
(XmlLoader.loadComponent) Iniciando carga de arquivo de configuração...
(XmlLoader.loadRequest) Carga realizada com sucesso!
(ConfigurationManager.LoadComponent) Unidade app.adi_solver3D-adi-y_solve pronta.
(XmlLoader.loadComponent) Iniciando carga de arquivo de configuração...
(XmlLoader.loadRequest) Carga realizada com sucesso!
(ConfigurationManager.LoadComponent) Unidade app.adi_solver3D-adi-z_solve pronta.
STARTING
Time step 1
Time step 20
Time step 40
Time step 60
Time step 80
Time step 100
Verification being performed for class S
Accuracy setting for epsilon = 1E-08
Comparison of RMS-norms of residual
0.  0.0274703154513896 0.0274703154513395 1.82361737447087E-12
1.  0.0103607467052864 0.0103607467052854 9.89524793400545E-14
2.  0.0162357450650762 0.0162357450650955 1.18898164286717E-12
3.  0.0158405572244557 0.0158405572244556 2.1902303705562E-15
4.  0.0348490406094186 0.0348490406093625 1.61142078468681E-12
Comparison of RMS-norms of solution error
0.  2.72892585574258E-05 2.72892585573772E-05 1.77928310387456E-12
1.  1.03644466408388E-05 1.03644466408373E-05 1.48739244594952E-13
2.  1.61547982871388E-05 1.61547982871665E-05 1.71201888764644E-12
3.  1.57507049944802E-05 1.57507049944801E-05 4.08708715031404E-15
4.  3.417766618345E-05 3.41776661833905E-05 1.73879138676362E-12
SP.S: Verification Successful
***** NAS Parallel Benchmarks Java version (NPB3_0_JAV) ADI_Sol*v
* Class = S *
* Size = 12 X 12 X 12 *
* Iterations = 100 *
* Time in seconds = 7.368 *
* ACCTime = 0.000 *
* Mops total = 96.673 *
* Operation type = floating point *
* Verification = Successful *
* *
* Please send all errors/feedbacks to:  *
* NPB Working Team *
* npb@nas.nasa.gov *
************************************************************
```

Fonte: Próprio autor.

Conforme podemos constatar, a atual implementação do componente impl.sp.solve.connector.SolverImpl não realiza qualquer notificação sobre sua execução na saída, padrão. É apenas relatado o carregamento dos conectores e as notificações nas iterações realizadas pela aplicação ADI_Solver3D. O que desejamos agora é incluir notificações sobre o início e término da ação main do solver.

A execução de uma aplicação paralela no *Back-End* é realizada por um executável chamado *run_hpe*. Após inicializar a execução da aplicação adi.ADI_Solver3D, o *run_hpe* disponibiliza um *prompt* onde é possível submeter o comando de reconfiguração. Este é o mesmo *prompt* a partir de onde é possível realizar a consulta dos nomes dos componentes apresentada na Figura 5.7. Após realizada a execução da aplicação, uma vez submetido o *script* da Figura 5.6, a nova saída obtida é apresentada na Figura 5.9.

A nova saída, além das informações anteriores, descreve todos os passos realizados durante o processo de reconfiguração, desde o carregamento do *script*, a avaliação dos estados críticos, a suspensão da execução desses estados, a efetivação da alteração, até o reinício da execução da unidade.

Logo em seguida, passamos a receber a notificação do início e término da execução do componente **x_solve**, conforme esperávamos.

5.5.3 Experimento 2 (Reconfiguração Estrutural)

O segundo experimento diz respeito a realização da reconfiguração dinâmica através da substituição do parâmetro de contexto *method*, de forma que seu valor passe de common.method.SPMethod para common.method.BTMethod, com objetivo de trocar a instância do componente aninhado **adi** e consequentemente as instâncias dos resolvedores **x_solve**, **y_solve** e **z_solve**. A troca busca a inserção de novos componentes que sejam adequados para o novo parâmetro de contexto.

Para realizar esta reconfiguração, similarmente à estratégia utilizada no experimento anterior, elaboramos um *script*, apresentado na Figura 5.10 na forma de linguagem intermediária, o qual possui alguns elementos distintos do *script* desenvolvido para o procedimento anterior, tendo em vista a diferente natureza da reconfiguração.

A parte inicial deste novo *script* é similar ao anterior, apresentando a *tag* principal reconfigurationRequest e sendo sua primeira *tag* filha, a identificação do componente a ser reconfigurado. Desta vez, a *tag* targetComponent recebe como valor *app.adi_solver3D*, de forma a identificar o componente ADI_Solver3D que

Figura 5.9: Saída obtida após o processo de reconfiguração

```
NAS Parallel Benchmarks C# version (NPB3_0_CS)
Multithreaded Version ADI_Solver3D.S np=1
(XmlLoader.loadComponent) Iniciando carga de arquivo de configuração...
(XmlLoader.loadRequest) Carga realizada com sucesso!
(ConfigurationManager.LoadComponent) Unidade app.adi_solver3D-adi pronta.
(XmlLoader.loadComponent) Iniciando carga de arquivo de configuração...
(XmlLoader.loadRequest) Carga realizada com sucesso!
(ConfigurationManager.LoadComponent) Unidade app.adi_solver3D-adi-x_solve pronta.
(XmlLoader.loadComponent) Iniciando carga de arquivo de configuração...
(XmlLoader.loadRequest) Carga realizada com sucesso!
(ConfigurationManager.LoadComponent) Unidade app.adi_solver3D-adi-y_solve pronta.
(XmlLoader.loadComponent) Iniciando carga de arquivo de configuração...
(XmlLoader.loadRequest) Carga realizada com sucesso!
(ConfigurationManager.LoadComponent) Unidade app.adi_solver3D-adi-z_solve pronta.
STARTING
Time step 1
(ConfigurationManager.EvaluateReconfiguration) Iniciando avaliação da
reconfiguração...
(XmlLoader.loadRequest) Iniciando carga de script de reconfiguração...
(XmlLoader.loadRequest) Carga de script realizada com sucesso!
(SecurityAnalyzer.Evaluate) Avaliando a ação:  solve.main
(SecurityAnalyzer.Evaluate) Avaliando as reconfigurações comportamentais...
(SecurityAnalyzer.Evaluate) Definindo os estados críticos...
(ConfigurationManager.EvaluateReconfiguration) Avaliação Concluída

(ConfigurationManager.Commit) Aplicando a reconfiguração...
(Configuration.stopStates) Suspendendo a execução do estado 0...
(Configuration.stopStates) Suspendendo a execução do estado 18...
(ConfigurationManager.WaitForSafeState) Aguardando estado seguro...
(ConfigurationManager.WaitForSafeState) Estados críticos OK...
(ConfigurationManager.Commit) Estado seguro para reconfiguração!
(ConfigurationManager.Commit) Adicionando fatia 'log'...
(ConfigurationManager.CommitReconfiguration) Criando a unidade concreta 'log'...

(Configuration.runStates) Reiniciando a execução do estado 0...
(Configuration.runStates) Reiniciando a execução do estado 18...
(ConfigurationManager.Commit) Reconfiguração aplicada!
(solve x) Execução Finalizada as 05:38:14 AM
(solve x) Execução Iniciada as 05:38:14 AM
(solve x) Execução Finalizada as 05:38:14 AM
(solve x) Execução Iniciada as 05:38:14 AM
(solve x) Execução Finalizada as 05:38:14 AM

<varias notificações semelhantes...>

Time step 100
(solve x) Execução Iniciada as 05:38:19 AM
(solve x) Execução Finalizada as 05:38:19 AM
Verification being performed for class S
Accuracy setting for epsilon = 1E-08
Comparison of RMS-norms of residual
0.  0.0274703154513896 0.0274703154513395 1.82361737447087E-12

<continua com a mesma finalização da saída anterior...>
```

Fonte: Próprio autor.

Figura 5.10: Script de reconfiguração do ADI_Solver3D linguagem intermediária

```
<?xml version="1.0" encoding="UTF-8" ?>

<reconfigurationRequest>

    <targetComponent>app.adi_solver3D</targetComponent>

    <parameter>
        <identifier>method</identifier>
        <constraint>
            <componentConstraint>bt.solve.BTMethod</componentConstraint>
        </constraint>
    </parameter>

</reconfigurationRequest>
```

Fonte: Próprio autor.

desejamos reconfigurar.

A diferença estrutural do novo *script* fica a cargo da *tag parameter*, a qual indica que o parametro de contexto *method* deve assumir um novo valor: bt.solve.BTMethod.

Uma vez submetido ao *Back-End* essa requisição de mudança, a avaliação inicial demanda um esforço computacional significativamente maior ao comparar a outra modalidade de reconfiguração. Por tratar-se de uma reconfiguração implícita, deve ser realizado um processo de analise para investigar em quais componentes as alterações deverão ocorrer. Essa investigação é feita em dois passos:

- Identificar, dentre os componente aninhados do conector alvo, aqueles que têm sua instanciação dependente dos parâmetros alterados;
- Para cada um dos componentes identificados, realizar uma consulta, fazendo uso do HTS, em vistas a constatar se a alteração daquele parâmetro de contexto resulta em alteração do componente concreto associado ao componente aninhado.

De modo a melhor esclarecer o segundo item, vamos relembrar alguns conceitos relacionados ao modelo de componentes em questão. No modelo HASH, os componentes abstratos são as entidades que definem os tipo de componentes, podendo estes possuir paramentros que especializam as suas futuras implementações. Já os componentes concretos realizam a implementação desses tipos de componentes, considerando os parâmetros de contextos neles existentes. O processo de instanciação de uma aplicação HASH indica apenas o tipo abstrato do componente

a ser executado, juntamente com os seus parâmetros de contexto. Ao avaliar essas informações, o motor do sistema de tipo (HTS) identifica qual o componente concreto mais adequado para instanciar.

Entretanto, a mudança de valor do parâmetro de contexto não obrigatoriamente implica na alteração do componente concreto, tendo em vista que o componente atual pode continuar a ser o mais adequado. Isso ocorre em função da hierarquização dos tipos de parâmatro. Por exemplo, os tipos SPMethod e BTMethod são especializações de SolvingMethod. Suponhamos a existência de um componente abstrato ACom que possua o parâmetro *method*. Ao desenvolver um componente concreto ImplCom do tipo ACom que não implemente especialidades de nenhum dos métodos SP ou BT, o valor deste parametro será SolvingMethod. Imaginando que para o tipo ACom esteja disponível no *Back-End* apenas a implementação ImplCom, reconfigurações sobre este componente mediante a troca do parâmetro de contexto não causarão nenhuma substituição, tendo em vista a ausência de implementações mais adequadas.

A título de exemplo, a Figura 5.11 apresenta o resultado do processo de avaliação do *script* de reconfiguração estrutural sobre o conector *app.adi_solver3D*. Como pode ser visto, este cenário não ocorre no caso específico da nossa reconfiguração. O **adi** é o único componente diretamente aninhado de ADI_Solver3D a depender do parâmtro *method*, para o qual há uma melhor implementação ao modificar de SP para BT. Assim, este deverá receber um novo componente concreto. Como consequência desta alteração, os componentes aninhados **x_solve**, **y_solve** e **z_solve** também são impactados, porém essas substituições são tratadas durante o processo de instanciação do novo **adi**.

Vencida a etapa de avaliação, o ambiente de configuração, na forma do ConfigurationManager fica apto a efetivar as reconfigurações mediante o atingimento dos estados seguros, similarmente ao que ocorre no experimento de reconfiguração anterior, porém acrescido da desconexão dos velhos componentes para conexão dos novos.

Figura 5.11: Saída obtida após o processo de avaliação da reconfiguração estrutural sobre a aplicação, do tipo adi.ADI_Solver3D.

```
(ConfigurationManager.EvaluateReconfiguration) Iniciando avaliação da
reconfiguração...
(XmlLoader.loadRequest) Iniciando carga de script de reconfiguração...
(XmlLoader.LoadRequest) Carregando as reconfigurações estruturais...
(XmlLoader.LoadRequest) Carregando as reconfigurações comportamentais...
(XmlLoader.loadRequest) Carga de script realizada com sucesso!
(ConfigurationManager.EvaluateReconfiguration) Identificando os componentes
impactos...
(StructuralReconfigurationRequest.GenerateChanges) Componente 'adi' impactado
pelo parametro 'method'.
(StructuralReconfigurationRequest.GenerateChanges) Finalizada a avaliação dos
InnerComponents!
(SecurityAnalyzer.Evaluate) Avaliando a ação 'main'
(SecurityAnalyzer.IsChangeConcrete) Parametros definem novo componente concreto
para 'adi'!
(SecurityAnalyzer.Evaluate) Definindo os estados críticos...
(ConfigurationManager.EvaluateReconfiguration) Avaliação Concluída!
```

Fonte: Próprio autor.

Capítulo 6

Considerações Finais

O crescimento e o amadurecimento da área de Computação de Alto Desempenho (CAD) tem sido ao longo das décadas fortemente guiado pelas necessidades de usuários oriundos das área das ciências e das engenharias, tanto em aplicações de interesse acadêmico quanto de interesse industrial. O envolvimento dos cientistas da computação tem como foco principal a busca por viabilizar a escalabilidade computacional e o desempenho, requisitos importantes no contexto de CAD, sendo por muitas vezes relegadas ao segundo plano questões relacionadas a requisitos não-funcionais, como a usabilidade e a manutenibilidade do *software*.

Porém, o crescimento em escala e complexidade dos programas com requisitos de CAD desenvolvidos no contexto das ciências e engenharias agora exige ferramentas e linguagens que permitam menor tempo de desenvolvimento e maior facilidade de manutenção e de verificação de confiabilidade, requisitos que passaram a ser investigados dentro do contexto da pesquisa em CAD. Com a investigação sobre o uso de conceitos, técnicas e ferramentas disseminadas da engenharia de *software*, as pesquisas na área de CAD passaram a não mais restringirem-se a mera busca por menor tempo de execução em suas aplicações.

Ferramentas e linguagens mais acessíveis passam a ser demandadas por uma classe de novos usuários de CAD que desejam usufruir dos benefícios dos experimentos e simulações computacionais, antes apenas restrito a cientistas e engenheiros extremamente familiarizados com as complexas tecnologias empregadas.

Com esse novo contexto em vista, grandes avanços foram alcançados. Sabemos que o acesso e compreensão da infraestrutura básica de *hardware* e *software* ainda não pode ser considerada trivial, frente a outras áreas que alcançaram maior popularidade na computação. Entretanto, os esforços dos últimos anos,

notadamente no que tange o desenvolvimento de novas linguagens paralelas, já apresenta resultados animadores. O programa *High Productivity Computing Systems* da DARPA [26], agência norte-americana de projetos avançados de defesa, é um excelente representante das iniciativas para a simplificação de meios de programação paralela através do estímulo ao projeto de novas linguagens de programação [44].

Em consonância com tais iniciativas, este trabalho discutiu a aplicação de técnicas de engenharia de *software* no contexto do desenvolvimento voltado a CAD, especialmente no que diz respeito ao uso de componentes e conceitos de arquitetura de *software*. Os *modelos de componentes*, consolidados e largamente utilizados em aplicações corporativas, tiveram os seus conceitos adaptados com sucesso para CAD através de *frameworks* computacionais e plataformas de componentes baseados nos modelos CCA e Fractal. Assim, as vantagens do uso de componentes, notadamente quanto ao encapsulamento e reuso, mostraram-se viáveis para uso em CAD através desses trabalhos. Esta dissertação esteve especialmente interessado em problemas relacionados ao projeto de plataformas de componentes onde estes são intrinsecamente paralelos, segundo a definição do modelo *Hash*, notadamente o HPE (*Hash Programming Environment*).

Dentro desse contexto, identificamos neste trabalho que, além da necessidade de se definir infraestruturas de execução de aplicações paralelas baseadas em componentes, há ainda uma demanda associada ao suporte a reconfiguração dinâmica, uma vez que, com frequência, o tempo de execução de aplicações de CAD atinge a escala de horas ou dias. A possibilidade de realizar ajustes e correções sem desprezar o processamento já realizado apresenta grande relevância, principalmente durante desenvolvimentos experimentais para validação dos *softwares* desenvolvidos, antes de sua implantação para uso em produção.

Segundo nossa investigação, acreditamos que o atendimento aos requisitos associados a reconfiguração dinâmica passa pela construção de um ambiente de execução que possa ser programado por uma *linguagem de descrição de arquitetura* capaz de descrever *conectores exógenos* hábeis a orquestrar a execução de um conjunto de componentes paralelos para atingir uma certa finalidade. Resolvemos então desenvolver uma solução que atendesse aos requisitos da plataforma HPE.

O desenvolvimento de conectores exógenos permite a coordenação de ações de componentes independentes, potencialmente reutilizáveis, para a resolução de problemas computacionais, preservando o baixo acoplamento e possibilitando a composição hierárquica.

Diante disso, foram propostas extensões à linguagem de descrição de arquitetura HCL (HASH *Configuration Language*), que a tornam capaz de especificar configurações e reconfigurações de componentes sobre o HPE. Além da linguagem, desenvolvemos um arcabouço capaz de realizar a interpretação das ações descritas por essa linguagem, como também por seus comandos de reconfiguração. Dessa forma, componentes podem ser completamente descritos, computacionalmente, em termos das construções do HCL, sem necessidade de dependência a uma linguagem de programação hospedeira e tornando possível a intervenção em tempo de execução sobre a arquitetura do *software* baseado em componentes. Isso nos motivou a estudar e classificar as formas de reconfiguração suportadas pelo HPE.

De acordo com nossos experimentos, foi possível atestar a viabilidade das soluções propostas e implementadas na execução deste trabalho.

Apesar do caráter de prototipação, onde a otimização da implementação do interpretador de protocolos em relação ao desempenho e uso da memória não é o foco principal, os resultados mostraram sobrecarga considerada aceitável no processo de interpretação, para instâncias realísticas de problemas, de forma que, quando utilizado com prudência, os conectores podem serem utilizado até mesmo em cenários de produção, oferecendo maior flexibilidade aos usuários da aplicação ao custo de penalidades toleráveis de desempenho, que poderiam ser, por exemplo, compensadas com a aceleração da execução devido a reconfigurações apropriadas, afetando os algoritmos utilizados de acordo com o contexto da execução.

Os ensaios de reconfiguração, com estudo de caso estrutural e comportamental, também se mostraram satisfatórios para os requisitos apresentados. A simplicidade do mecanismo utilizado é um dos principais diferencias da solução, o qual não poderia existir sem a premissa de segregação de dados e procedimentos defendidos pelo modelo HASH e implementada pelo HPE.

6.1 Confrontação dos Resultados perante os Objetivos

Nos parágrafos a seguir, confrontamos os objetivos anunciados para esse trabalho de pesquisa com os resultados encontrados pela sua execução, visando esclarecer as limitações dos objetivos alcançados e identificar possibilidades de trabalhos futuros.

6.1.1 Proposta e implementação de uma linguagem de descrição de arquitetura que forneça suporte à configuração e à reconfiguração dinâmica de conectores exógenos para orquestração de componentes paralelos, sobre a plataforma HPE

Os resultados obtidos a partir do uso do arcabouço de interpretação e reconfiguração se mostraram satisfatórios dentro do que se esperava de seu comportamento.

Entendemos que este trabalho de dissertação conseguiu atender aos objetivos lançados, uma vez que apresentou as extensões necessárias à HCL para a especificação de conectores exógenos, como também a infraestutura básica para a sua execução e reconfiguração.

Essas questões, entretanto, não indicam a ausência de limitações neste trabalho.

Os experimentos foram realizados mediante a utilização da HCL em forma de linguagem intermediária, sendo necessária adaptação do atual compilador para reconhecer os novos construtores da linguagem, como também efetuar a correta geração no novo formato de linguagem intermediária.

A herança funcional, como também o protocolo de validação não foram explorados durante a realização dos experimentos. A validação dos mecanismos não foi exercitada por este trabalho.

6.1.2 Avaliar qualitativamente os mecanismos que permitem a inclusão de reconfiguração dinâmica em modelos de componentes em geral

No Capítulo 2 discutimos os mecanismos capazes de suportar a realização de reconfigurações em sistemas de *software*. Buscamos trazer a tona os artifícios já utilizados para tal fim, além da apresentar suas vantagens e desvantagens.

Entretanto, parte das discussões foram limitadas a apontamentos teóricos, pela ausência de informações detalhadas do funcionamento das aplicações apresentadas pelas publicações.

Com um pouco mais de ousadia, futuras discussões sobre o assunto podem desenvolver pequenas provas de conceito para fazer juízo de valor a partir de experimentos práticos.

6.1.3 Identificar e descrever os principais modos de reconfiguração estática e dinâmica de componentes paralelos na plataforma HPE

Definimos com precisão os tipos de reconfiguração dinâmica suportados pelo HPE, trabalho realizado sob o amparo de uma metodologia de classificação já existente, porém adaptada às particularidades do modelo HASH.

Como resultado, formalizamos três tipos de reconfiguração:

- A ***reconfiguração comportamental***, que pode ser aplicada sobre componentes de configuração a fim de alterar o comportamento de uma ação ou condicional de uma unidade de um componente;
- A ***reconfiguração estrutural explícita***, atuante também sobre os componentes de configuração, porém com a finalidade de lhes adicionar novos componentes aninhados, de modo a gerar a inserção de novas fatias às suas unidades. Imaginamos que seu uso com frequência ocorra associado à reconfiguração anterior; e
- A ***reconfiguração estrutural implícita***, que pode ser aplicada sobre componentes de configuração, porém podendo repercutir em qualquer dos tipos de componentes transitivamente aninhados. Essa reconfiguração atualmente é realizada mediante a alteração de parâmetros de contexto do sistema de tipos HTS, suportado pelo HPE, resultando na readequação dos componentes ao novo contexto apresentado.

Quanto a reconfiguração estática, defendemos o uso da herança funcional a qual pode se associar com qualquer uma das reconfigurações acima definidas, dando, desta forma, o mesmo tratamento e expressividade das dinâmicas.

6.1.4 Disponibilizar uma infraestrutura de execução de aplicações baseadas em conectores exógenos, desenvolvidas com a ADL proposta

Já citamos há pouco, na seção que descreve o objetivo geral, os resultados satisfatórios obtidos com o ambiente de interpretação e reconfiguração, para os quais entendemos que já haja suficiente exposição. Nesta seção, acreditamos ser de maior relevância relatar as suas limitações.

O protótipo desenvolvido deve ser encarado como tal, tendo em vista a baixa maturidade da atual implementação para dar suporte a execução em ambientes críticos em produção.

Os experimentos de interpretação apontam a necessidade de maior rigor na gestão de memória, tendo em vista o crescimento de seu consumo a cada nova iteração. Além de revisar os pontos de criação de objetos, melhorias também podem ser alcançadas a partir de configurações e usos explícitos de funções do *garbage collector*;

Também é necessário revisar a arquitetura de *software* empregada pelo protótipo, buscando explorar métodos mais eficientes de implementação. Essa tarefa inclusive pode ser guiada pela identificação de possíveis gargalos durante a execução da aplicação BT, tendo em vista a exorbitante discrepância dos resultados obtidos entre os cenários BT.W.9 e BT.S.9, a partir dos quais podemos supor a existência de possíveis otimizações em função do forte contraste frente aos outros experimentos realizados com a aplicação SP.

6.2 Contribuições

Em qualquer nível de desenvolvimento, motivado pela economia de tempo e custo, o reuso de *software* é desejável. Naturalmente, os benefícios são mais acentuados quando se trata de componentes de grossa granularidade, capazes de entregar grandes funcionalidades e significativas porções de código-fonte. Quando modelados de forma devida, esses componentes se tornam adequados para a programação de grande escala na forma de aplicações escritas como conectores exógenos.

Programar nesta escala traz o benefício da simplicidade ao manipular grandes componentes que representam entidades e aspectos do domínio do problema. Porém, este benefício só é alcançado quando a linguagem de programação segue os mesmos preceitos, disponibilizando operadores e construções coerentes com o tipo de programação demandada. Linguagens de propósito geral, conhecidas pela versatilidade e grande expressividade, passam a ter nestas mesmas característica os principais empecilhos para o seu uso na configuração de macro componentes. A versatilidade e expressividade destas linguagens estão apoiadas na baixa granularidade de suas instruções. Indubitavelmente, é possível realizar todas as operações de coordenação e configuração a partir de uma dessas linguagens, tendo porém que utilizar um conjunto de chamadas. Porém, na medida em que operações conceitualmente atômicas para o modelo tem de ser desenvolvidas mediante várias instruções, este excesso induz ao erro, dificulta o desenvolvimento, como também a legibilidade das aplicações.

A disponibilização de uma nova ADL, com extensões que permitam a

HCL especificar o comportamento das ações, contribui para a simplificação do desenvolvimento de grandes aplicações construídas por composição. Com isso, além dos benefícios associados ao reuso, passamos a permitir de forma prática no âmbito do HPE, a possibilidade de existência de dois perfis de programadores, onde o primeiro, com perfil mais técnico, atua no desenvolvimento em pequena escala, na confecção dos componentes nativos, enquanto que o segundo, detentor de maior conhecimento da área fim, faz uso desses componentes de forma simplificada através da construção de conectores descritos em HCL. Essa divisão de papéis favorece a popularização da programação paralela e distribuída, normalmente associada a complexidade de seu uso.

Com a reconfiguração dinâmica, suprimos uma importante lacuna na plataforma HPE. O amadurecimento das funcionalidades defendidas pela plataforma, a credencia a tornar-se uma alternativa na execução de aplicações concorrentes, permitindo a disseminação do uso de componentes intrinsecamente paralelos, principal diferencial da plataforma. O uso conjunto da nova ADL com a reconfiguração dinâmica oferece as ferramentas necessárias para a prototipação e o desenvolvimento experimental, de forma a criar mais um meio que contribua para cientistas e pesquisadores desfrutarem de máquinas de alto desempenho, pois apesar de serem especialistas nas suas áreas fins, possuem baixas habilidades de programação concorrente e distribuída.

Este trabalho contribui ainda como insumo para a definição e implementação de reconfiguração dinâmica no contexto de outras plataformas de componentes. A forma modular do arcabouço permite a sua inserção em outras plataformas mediante a necessidade de pouca alteração, desde que esta seja aderente ao modelo HASH.

6.3 Trabalhos Futuros

Ao longo da realização desta dissertação, trabalhos de extensão ou de aprofundamento se mostraram relevantes dentro do contexto estudado, os quais citamos abaixo como sugestão para trabalhos futuros:

- Desenvolvimento de um compilador *source-to-source* que permita a tradução da linguagem HCL para a linguagem intermediária do arcabouço de execução, ambas especificadas por este trabalho;
- Realização de testes de estresse, mediante a utilização de componentes interpretados em vários níveis de hierarquização de forma a identificar os níveis

de composição máximos até o surgimento de sobrecargas proibitivas;

- Criação de um mecanismo de transferência de estado, de forma permitir que componentes de estruturas de dados e outros que mantenham informações de execução possam participar dos processos de reconfiguração;
- Demonstração de prova formal de corretude do mecanismo de reconfiguração dinâmica proposto, alinhado com métodos mais eficazes de detecção e prevenção de *deadlocks*.

Referências Bibliográficas

[1] The common component architecture forum. Disponível em: <http://http://www.cca-forum.org>, 2012.

[2] The ruby programming language. Disponível em: <http://www.ruby-lang.org>, 2012.

[3] Allan, B. A., and Armstrong, R. Ccaffeine framework: Composing and debugging applications interactively and running them statically. Tech. rep., Compframe 2005, June 2005.

[4] 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 & Experience 14*, 5 (2002), 323–345.

[5] Araújo, G. A., Carvalho, Jr, F. H., and Corrêa, R. C. Implementing endogenous and exogenous connectors with the common component architecture. In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing* (New York, NY, USA, 2009), CBHPC '09, ACM, pp. 12:1–12:4.

[6] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 1999), HPDC '99, IEEE Computer Society, pp. 115–124.

[7] 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 & Experience 18*, 2 (February 2006), 215–229.

[8] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., V., V., and Weeratunga, S. The nas parallel benchmarks. Tech. Rep. RNR-91-02, NASA Ames Research Center, Moffett Field, CA, 1991.

[9] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. The nas parallel benchmarks - summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1991), Supercomputing '91, ACM, pp. 158–165.

[10] Batista, T., Gomes, A. T., Coulson, G., Chavez, C., and Garcia, A. On the interplay of aspects and dynamic reconfiguration in a specification-to-deployment environment. In *Proceedings of the 2nd European conference on Software Architecture* (Berlin, Heidelberg, 2008), ECSA '08, Springer-Verlag, pp. 314–317.

[11] Batista, T., and Rodriguez, N. Dynamic reconfiguration of component-based applications. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems* (Washington, DC, USA, 2000), IEEE Computer Society, pp. 32–.

[12] 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*, 1 (2009), 5–24.

[13] Bertrand, F., and Bramley, R. Dca: a distributed cca framework based on mpi. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments.* (Los Alamitos, CA, USA, April 2004), IEEE Computer Society, pp. 80 – 89.

[14] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. The fractal component model and its support in java: Experiences with

auto-adaptive and reconfigurable systems. *Software Practice & Experience 36*, 11-12 (September 2006), 1257–1284.

[15] Bruneton, E., Coupaye, T., and Stefani, J. *The Fractal Component Model*, Fevereiro 2004. Versão 2.0-3.

[16] Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. Towards a taxonomy of software change: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice 17*, 5 (September 2005), 309–332.

[17] Carvalho Junior, F. H., and Correa, R. C. The Design of a CCA Framework with Distribution, Parallelism, and Recursive Composition. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing (GRID'2010) - 2010 Workshop on Component-Based High Performance Computing (CBHPC'10)* (Brussels, Belgium, 2010), pp. 339–348.

[18] Carvalho Junior, F. H., Correa, R. C., Lins, R., Silva, J. C., and Araújo, G. A. High Level Service Connectors for Components-Based High Performance Computing. In *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing* (Gramado, RS, Brazil, October 2007), SBAC-PAD 2007, pp. 237–244.

[19] Carvalho-Junior, F. H., and Lins, R. D. An institutional theory for #-components. *Electronic Notes in Theoretical Computer Science (ENTCS) 195* (January 2008), 113–132.

[20] Carvalho Junior, F. H., Lins, R. D., Corrêa, R. C., and Araújo, G. A. Towards an architecture for component-oriented parallel programming. *Concurrency and Computation: Practice & Experience 19*, 5 (April 2007), 697–719.

[21] Crnkovic, I., Sentilles, S., Vulgarakis, A., and Chaudron, M. A classification framework for software component models. *IEEE Transactions on Software Engineering 37*, 5 (September-October 2011), 593 –615.

[22] Danelutto, M., Fragopoulou, P., Getov, V., Malawski, M., Gubaula, T., Kasztelnik, M., Bartynski, T., Bubak, M., Baude, F., and Henrio, L. High-level scripting approach for building component-based

applications on the grid. In *Making Grids Work*. Springer US, 2008, pp. 309–321.

[23] de Araújo, G. A. *Uma Plataforma para Aplicações Científicas de Alto Desempenho Usando Conectores*. PhD thesis, Universidade Federal do Ceará, 2010.

[24] de 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.

[25] de Rezende, C. A. Avaliação de desempenho de uma plataforma de componentes paralelos. Master's thesis, Universidade Federal do Ceará, 2011.

[26] Dongarra, J., Graybill, R., Harrod, W., Lucas, R., Lusk, E., Luszczek, P., Mcmahon, J., Snavely, A., Vetter, J., Yelick, K., Alam, S., Campbell, R., Carrington, L., Chen, T.-Y., Khalili, O., Meredith, J., and Tikir, M. Darpa's hpcs program: History, models, tools, languages. In *Advances in COMPUTERS High Performance Computing*, M. V. Zelkowitz, Ed., vol. 72 of *Advances in Computers*. Elsevier, 2008, pp. 1 – 100.

[27] Englander, R. *Developing Java beans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[28] Frumkin, M., Schultz, M., Jin, H., and Yan, J. Implementation of the nas parallel benchmarks in java. Tech. Rep. NAS-02-009, NASA Ames Research Center, Moffett Field, CA, 2002.

[29] Heineman, G. T., and Councill, W. T., Eds. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[30] Hoare, C. A. R. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[31] Hovland, P., Keahey, K., McInnes, L. C., Norris, B., Diachin, L. F., and Raghavan, P. A quality of service approach for high-performance numerical components. In *Proceedings of the Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference* (Toulouse, France, June 2003).

[32] Kang, P., Selvarasu, N. K., Ramakrishnan, N., Ribbens, C. J., Tafti, D. K., and Varadarajan, S. Modular, fine-grained adaptation of parallel programs. In *Proceedings of the 9th International Conference on Computational Science: Part I* (Berlin, Heidelberg, 2009), ICCS '09, Springer-Verlag, pp. 269–279.

[33] Kell, S. Rethinking software connectors. In *Proceedings of the International workshop on Synthesis and analysis of component connectors: in conjunction with the 6th ESEC/FSE joint meeting* (New York, NY, USA, 2007), SYANCO '07, ACM, pp. 1–12.

[34] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C. V., Maeda, C., and Mendhekar, A. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)* (Berlin, November 1997), vol. 1241 of *Lecture Notes in Computer Science*, Springer, p. 220–242.

[35] Kim, D. K., Tilevich, E., and Ribbens, C. J. Dynamic software updates for parallel high-performance applications. *Concurrency and Computation: Practice & Experience 23*, 4 (March 2011), 415–434.

[36] Kleinoder, J., and Golm, M. Metajava: an efficient run-time meta architecture for java/sup tm/. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)* (Washington, DC, USA, 1996), IWOOOS '96, IEEE Computer Society, pp. 54–.

[37] Kramer, J., and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng. 16*, 11 (Nov. 1990), 1293–1306.

[38] Kurzyniec, D., and Others. Towards Self-Organizing Distributed Computing Frameworks: The H2O Approach. *Parallel Processing Letters 13*, 2 (2003), 273–290.

[39] Lau, K.-K., Velasco Elizondo, P., and Wang, Z. Exogenous connectors for software components. In *Proceedings of the 8th international conference on Component-Based Software Engineering* (Berlin, Heidelberg, 2005), CBSE'05, Springer-Verlag, pp. 90–106.

[40] Lau, K.-K., and Wang, Z. Software component models. *IEEE Transactions on Software Engineering 33*, 10 (October 2007), 709–724.

[41] Léger, M., Ledoux, T., and Coupaye, T. Reliable dynamic reconfigurations in the fractal component model. In *Proceedings of the 6th international workshop on Adaptive and reflective middleware: held at the ACM/IFIP/USENIX International Middleware Conference* (New York, NY, USA, 2007), ARM '07, ACM, pp. 3:1–3:6.

[42] Liang, S. *Java Native Interface: Programmer's Guide and Reference*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[43] Luckham, D. C., and Vera, J. An event-based architecture definition language. *IEEE Transactions on Software Engineering 21*, 9 (September 1995), 717–734.

[44] Lusk, E., and Yelick, K. Languages for High-Productivity Computing - The DARPA HPCS Language Support. *Parallel Processing Letters*, 1 (2007), 89–102.

[45] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Software Engineering — ESEC '95*, W. Schäfer and P. Botella, Eds., vol. 989 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1995, pp. 137–153.

[46] Magee, J., and Kramer, J. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes 21*, 6 (October 1996), 3–14.

[47] Malawski, M., Bartynski, T., Ciepiela, E., Kocot, J., Pelczar, P., and Bubak, M. An ADL-based support for CCA components on the Grid. In *Proceeding of the CoreGRID Workshop on Grid Systems, Tools and Environments in Conjunction with GRIDS@work: CoreGRID Conference, Grid Plugtests and Contest* (Sophia-Antipolis, France, December 2006).

[48] May, M. D., Taylor, R. J. B., and Whitby-Strevens, C. Epl: an experimental language for distributed computing. In *Proceeding of the Trends and Applications: Distributed Processing* (May 1978), National Bureau of Standards, pp. 69–71.

[49] McKinley, P., Sadjadi, S., Kasten, E., and Cheng, B. Composing adaptive software. *Computer 37*, 7 (July 2004), 56 – 64.

[50] Medvidovic, N., Dashofy, E. M., and Taylor, R. N. Moving architectural description from under the technology lamppost. *Information and Software Technology 49*, 1 (January 2007), 12–31.

[51] Medvidovic, N., and Taylor, R. N. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering 26*, 1 (January 2000), 70–93.

[52] Mehta, N. R., Medvidovic, N., and Phadke, S. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering* (New York, NY, USA, 2000), ICSE '00, ACM, pp. 178–187.

[53] Meyer, B. *Object-Oriented Software Construction*, 2 ed. Interactive Software Engineering Inc. (ISE), 1997.

[54] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, June 1995. Revision 2.0.

[55] Object Management Group. *OMG Unified Modeling Language*, November 2007.

[56] Perry, D. E., and Wolf, A. L. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes 17*, 4 (October 1992), 40–52.

[57] Pessemier, N., Seinturier, L., Duchien, L., and Coupaye, T. A component-based and aspect-oriented model for software evolution. *International Journal of Computer Applications in Technology 31*, 1/2 (March 2008), 94–105.

[58] Potter, K. Methods for presenting statistical information: The box plot. *Hans Hagen, Andreas Kerren, and Peter Dannenmann (Eds.), Visualization of Large and Unstructured Data Sets, GI-Edition Lecture Notes in Informatics (LNI) S-4* (2006), 97–106.

[59] Pressman, R. S. *Software Engineering: A Practitioner's Approach*, 5th ed. McGraw-Hill Higher Education, 2001.

[60] Sessions, R. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[61] SGS-THOMSON Microelectronics Limited. *Occam 2.1 Reference Manual*, May 1995. Document number: 72 occ 45 03.

[62] Shaw, M. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *Selected papers from the Workshop on Studies of Software Design* (London, UK, 1996), ICSE '93, Springer-Verlag, pp. 17–32.

[63] Shaw, M., and Garlan, D. Characteristics of higher-level languages for software architecture. Tech. Rep. CMU-CS-94-210, Carnegie Mellon University, School of Computer Science, December 1994.

[64] Shaw, M., and Garlan, D. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[65] Sipser, M. *Introdução à Teoria da Computação: Tradução da 2ª edição norte-americana (trad. Ruy José Guerra Barreto de Queiroz)*. Thomson Learning, São Paulo, 2007.

[66] Smith, J. E., and Nair, R. The architecture of virtual machines. *Computer 38*, 5 (May 2005), 32–38.

[67] Soria, C. C., Irshaid, N. A., Benedí, J. P., Cubel, J. A. C., and Salavert, I. R. Dynamic reconfiguration of software architectures through aspects. In *Proceeding of the 1st European Conference on Software Architecture (ECSA'07)* (Heidelberg, September 2007), vol. 4758 of *Lecture Notes on Computer Science*, Springer, pp. 279–283.

[68] Wang, A. J. A., and Qian, K. *Component Oriented Programming*. John Wiley & Sons, Inc., March 2005. ISBN 0-471-64446-3.

[69] Wegdam, M. *Dynamic Reconfiguration and Load Distribution in Component Middleware*. PhD thesis, Centre for Telematics and Information Technology, University of Twente, 2003.

[70] Young, A., and Magee, J. A flexible approach to evolution of reconfigurable systems. In *International Workshop on Configurable Distributed Systems* (March 1992), pp. 152 –163.

Apêndice A

Configuração main do conector impl.sp.solve.connector.SolverImpl

action *main*
begin
    SEQ
    begin
        START : $forward.begin$
        $lhs.begin$
        $read\_buffer\_forward.begin$
        $write\_buffer\_forward.begin$
        $lhs.go$
        $lhs.advance$
        $forward.go$
        $forward.advance$

        $!forward.finished?$ SEQ [REPEAT = $true$]
        begin
            $write\_buffer\_forward.go$
            $write\_buffer\_forward.advance$
            $read\_buffer\_forward.advance$
            $shift\_forward.initiate\_send$
            $shift\_forward.initiate\_recv$
            $lhs.go$
            $lhs.advance$
            $shift\_forward.go$
            $read\_buffer\_forward.go$
            $forward.go$
            $forward.advance$

```
            end

            backward.begin
            matvecproduct.begin
            read_buffer_backward.begin
            write_buffer_backward.begin
            backward.init
            backward.go
            backward.advance

            !backward.finished? SEQ [REPEAT = true]
            begin
                write_buffer_backward.go
                write_buffer_backward.advance
                read_buffer_backward.advance
                shift_backward.initiate_send
                shift_backward.initiate_recv
                matvecproduct.go
                matvecproduct.advance
                shift_backward.go
                read_buffer_backward.go
                backward.go
                backward.advance
            end

            END : matvecproduct.go
        end
    end
```