CALEBE: UMA MÁQUINA VIRTUAL PARALELA COM SUPORTE A LINGUAGENS MULTIPARADIGMA

Gustavo A. Kellerman[1] Jorge L. V. Barbosa[1,2]
Patrícia Kayser Vargas[1] Cláudio Fernando Resin Geyer[1]

[1] Universidade Federal do Rio Grande do Sul, Instituto de Informática
Porto Alegre, RS, Brasil
{keller,kayser,geyer}@inf.ufrgs.br

[2] Universidade Católica de Pelotas, Escola de Informática
Pelotas, RS, Brasil
barbosa@atlas.ucpel.tche.br

Resumo

Linguagens Multiparadigma têm sido estudadas como plataforma alternativa para o desenvolvimento de software, com a proposta de unir vantagens e solucionar deficiências encontradas em cada um dos paradigmas básicos de programação. Um dos problemas encontrados no seu desenvolvimento é estabelecer uma semântica que permita a unificação dos paradigmas. Uma abordagem é a utilização de construções com semântica em cada um dos paradigmas unificados, e mecanismos de integração. Outra abordagem é a utilização de uma unidade única de abstração, que suporte o estilo de programação de cada paradigma. A máquina virtual Calebe é baseada numa proposta que busca unir as vantagens de ambas as técnicas de implementação de linguagens. Para isto, possui um conjunto de operadores rico o bastante para prover modos diferenciados de computação, e integração entre esses modos numa base semântica comum. Além disso, existe suporte para concorrência e distribuição. A máquina virtual Calebe foi pensada como um *middleware* - um sistema intermediário entre a linguagem multiparadigma e o sistema distribuído onde esta é executada. Assim, pode-se escrever um compilador da linguagem multiparadigma para Calebe, a qual terá a mesma semântica em qualquer sistema, com a vantagem de que o ambiente da máquina virtual Calebe e seus operadores foram projetados para permitir a execução paralela, e o compartilhamento de recursos.

Palavras-chave: Linguagens de Programação; Sistemas Paralelos; Linguagens Multiparadigma.

1. Introdução

Com o desenvolvimento, ao longo da história de computação, de linguagens com nível crescente de abstração, surgiram diferentes paradigmas de desenvolvimento de *software*, cada um com seu próprio modelo de execução e metodologias associadas. Pode-se dizer que há pelo menos quatro paradigmas básicos: imperativo, funcional, em lógica e orientado a objetos, todos extensamente utilizados e pesquisados, além de novos paradigmas sendo desenvolvidos. Com o conhecimento acumulado durante anos no estudo e implementação de sistemas, percebeu-se que diferentes problemas possuem sua melhor solução em paradigmas diferentes, justamente por utilizarem tipos de abstrações diferentes. Assim, surgiu o interesse por parte dos pesquisadores de linguagens e de metodologias de *software* em obter um paradigma unificado, possibilitando o uso de diferentes tipos de abstrações na construção do mesmo aplicativo. Pode-se traçar como objetivos gerais a criação de uma ferramenta de desenvolvimento com a maior expressividade e clareza

possível para a elaboração do *software*, ao mesmo tempo que permitindo o uso de recursos computacionais adequados para a solução de cada problema, por este ter sido expresso com as abstrações que lhe são mais adequadas.

Em [BAR98] encontra-se uma classificação de linguagens multiparadigma em pesquisa, sendo o principal critério a utilização das unidades de abstração dos paradigmas existentes, ou a proposição de um tipo novo de abstração, capaz de substituir todos os outros. Uma máquina virtual adequada para a implementação de linguagens multiparadigma deve ser versátil o suficiente para possibilitar a tradução de diversos tipos de abstrações, ou seja, possuir uma semântica rica o bastante para acomodar a semântica das linguagens sendo implementada. Em Calebe, essa necessidade é suprida pelas operadores e tipos de dados acrescentados ao modelo *dataflow*.

Nos deparamos, atualmente, com o desenvolvimento conjunto da programação paralela e a construção de sistemas distribuídos, na confecção de diversos tipos de *clusters* e multiprocessadores, incluindo o atraente uso de redes de estações para a solução de problemas computacionais de grande porte.

Entretanto, os sistemas paralelos e distribuídos são inerentemente mais complexos que os seqüenciais, surgindo problemas como a divisão de um problema em subtarefas, a distribuição destas entre os nodos de processamento, comunicação, compartilhamento de dados e sincronização das subtarefas, e o uso de técnicas de balanceamento de carga e mobilidade. A solução desses problemas exige uma plataforma de desenvolvimento mais inteligente e robusta, capaz de abstrair o processo de paralelização dos programas, podendo o desenvolvedor se concentrar nas características e comportamentos específicos de sua aplicação.

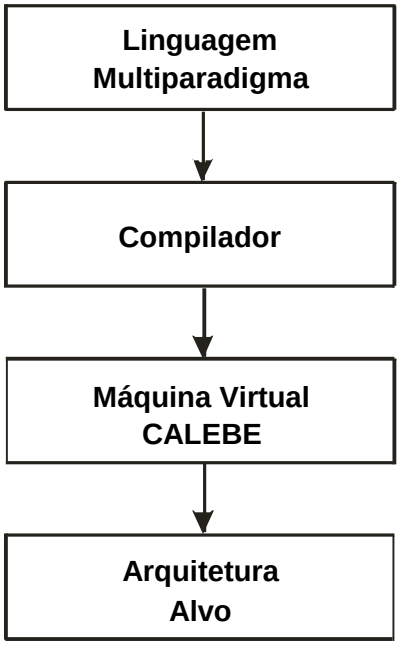

Figura 1 – Contexto de execução da máquina virtual

Assim, os objetivos do desenvolvimento de uma linguagem multiparadigma, e do uso eficiente do paralelismo, podem ser unificados: encontrar meios de construção de software adequados ao universo de problemas sendo modelado, ao mesmo tempo que encontrando uma representação computacional da solução desses problemas que utilize da melhor forma os recursos disponíveis, no caso, utilizando paralelismo entre diversas unidades de execução.

A criação de uma máquina virtual para o desenvolvimento de linguagens paradigmas em sistemas paralelos vem atender essa união de objetivos. Linguagens multiparadigma podem facilitar

a exploração automática do paralelismo, utilizando abstrações que se aproximam do universo do problema solucionado [BAR99]. Um programa descrito nessa classe de linguagens será traduzido por um compilador para o código da máquina virtual (figura 1). Enquanto que o código multiparadigma tende a ser descritivo, sem explicitar a ordem de execução e sincronização dos componentes do programa, o mesmo não ocorre com o código gerado para a máquina virtual. O compilador terá a tarefa de explicitar toda a sincronização e comunicação com o uso de operadores apropriados, além de fazer anotações referentes a granulosidade, mobilidade e replicação, permitindo um manejamento de recursos e balanceamento de cargas dependente das características do aplicativo. Esse código, no entanto, é independente de plataforma. A máquina virtual Calebe irá traduzir os operadores para cada arquitetura alvo onde esta for implementada. Além disso, ela utiliza *just in time compilation* para otimizar partes do programa para arquitetura alvo, além de lidar com trechos de código que mudam durante a execução. Em [POL99] é apresentada um dialeto de C com geração dinâmica de código. O objetivo é expandir o uso desse conceito, permitindo que determinadas abstrações de linguagens de alto nível (como funções de alta ordem, *templates*) possam ter instâncias traduzidas para código de baixo nível em tempo de execução. Essa abordagem é mais eficiente que a interpretação do significado dessas abstrações a cada utilização, especialmente se a função ou classe criada durante a execução tiver uma vida longa – o que pode ser determinado pelo compilador.

2. O Modelo Calebe

2.1. Exploração de Paralelismo num Modelo Híbrido entre Dataflow e Von Neumann

A escolha do modelo de execução *dataflow* como base para Calebe deve-se a este permitir a exploração do máximo de paralelismo existente entre as instruções de um programa, e possuir uma semântica simples, adequada para a implementação em forma de máquina, cuja tarefa é determinar os operadores prontos para execução, executá-los e direcionar suas saídas para os operadores de destino. Segundo a classificação de arquiteturas *dataflow* em [JAG95], Calebe é uma modificação do modelo estático proposto inicialmente em [DEN74]. Para contornar as deficiências deste modelo, Calebe pode executar de forma híbrida, utilizando segmentos de código Von Neumann da máquina hospedeira; utiliza alocação dinâmica de memória para resolver o problema da execução paralela de laços e funções recursivas, semelhantemente a [IAN90], e implementa direitos de acesso para sincronizar acessos concorrentes à mesma estrutura de dados.

Em *dataflow*, os operadores estão prontos para executar quando seus valores de entrada estão disponíveis, permitindo a execução paralela. Num sistema *dataflow* puro, em que toda a computação se dá através do fluxo de dados entre operadores, o programa exprime completa e exclusivamente as dependências na ordem de execução das instruções, ao contrário dos sistemas convencionais, em que são introduzidas dependências artificialmente, pela serialização do código e pela nomeação de variáveis temporárias. Essas deficiências são em parte resolvidas pelos processadores superescalares, mas apenas num espaço de busca limitado. Foram feitos estudos de arquiteturas intermediárias entre *dataflow* e Von Neumann [IAN90], com o desenvolvimento das denominadas *micro-threads*, linhas de execução invocadas e sustidas por hardware, segundo regras de sincronização. São o agrupamento em blocos de instruções Von Neumann, sincronizadas entre si de um modo *dataflow*. O elemento de sincronização é apresentado em [ARV89]: as *I-structures*, uma memória com característica *write once, read many*, em que o evento de uma escrita libera para execução todas as *micro-threads* que estavam presas após uma tentativa de leitura. Trata-se de uma versão baixo nível do modelo *blackboard* [VRA95].

Essa idéia é usada no modelo Calebe para diminuir o *overhead* de sincronização entre os operadores. Realiza-se a compilação de trechos de código cuja estrutura não se modifica em tempo

de execução. A ligação deste código com as primitivas da máquina virtual caracteriza o funcionamento como uma biblioteca de *runtime*. O aumento de eficiência do código compilado geralmente entra em conflito com a portabilidade. Para mantê-la, o modelo prevê um compilador de um subconjunto da linguagem da máquina virtual, que deve ser portado com esta para cada arquitetura hospedeira. Assim, no momento da carga do programa, trechos anotados serão compilados, aumentando a granulosidade dos operadores a serem executados. Devido ao modelo permitir a modificação de estruturas em tempo de execução, é impossível determinar se um trecho de código marcado como passível de compilação não precisará ser lido como código Calebe e recompilado posteriormente, motivo pelo qual a máquina tem de manter as duas versões do trecho de programa na memória, e traz a necessidade do tradutor da linguagem de alto nível para Calebe utilize heurísticas eficientes para marcar trechos de códigos para compilação ou execução interpretada. Não serão tratadas aqui as restrições impostas sobre o compilador anexo à máquina virtual, mas são tais que ele possa executar rapidamente, gerando código seqüencial livre de primitivas de sincronização e comunicação. Ou seja, é muito mais simples escrevê-lo e portá-lo para um subconjunto da linguagem Calebe do que fazer o mesmo com um compilador para uma linguagem multiparadigma paralela.

2.2. Operadores e Conexões

As unidades básicas Calebe são operadores *dataflow* e conexões entre estes, pelas quais trafegam valores em um único sentido, e referências, em ambos os sentidos.

Além de operadores que realizam funções aritméticas e lógicas, o modelo inclui operadores estruturais, outros ligados à chamada de funções e sincronização, operadores responsáveis pela manipulação de referências, e ainda os que são responsáveis por agregar informação de contexto a operadores, conexões ou dados.

Os tipos de dados podem ser divididos em duas classes disjuntas: valores e referências. Os primeiros correspondem aos tipos primitivos de uma linguagem de programação convencional, como inteiros, *booleans* e caracteres. As referências fazem papel de apontadores, e só podem ser criadas e manipuladas por operadores especiais. Estas não constituem um endereço de memória física, mas um endereço lógico com relação aos operadores estruturais, chamados de domínios.

2.3. Domínios, modos de endereçamento e direitos de acesso

Um domínio é um mapeamento de endereços lógicos em endereços físicos de operadores, incluindo outros domínios. Há dois tipos de mapeamento, considerando se o domínio possui o operador ou apenas tem acesso a ele. O primeiro tipo é uma relação de posse, e o segundo uma relação de acesso. Os domínios de um sistema formam uma árvore para a relação de posse, e um grafo conexo para a relação de acesso, pois toda relação de posse é implicitamente uma relação de acesso. No endereçamento absoluto, o endereço de um operador é um lista de seus endereços lógicos, obtidos percorrendo-se a relação de posse a partir do domínio raiz. No endereçamento relativo, um operador é localizado percorrendo-se as relações de acesso e de posse a partir de um domínio qualquer do grafo.

Uma conexão é formada por dois endereços lógicos, uma de cada operador ligado por ela. Se ela possui endereços absolutos, a movimentação de qualquer dos domínios abaixo de um dos operadores na árvore irá provocar a mudança do endereço. Se possui endereços relativos, o que importa são as mudanças nos domínios intermediários, que, em geral, serão em número menor. Nota-se que o endereçamento absoluto tem custo maior, mas o endereçamento relativo requer a existência de uma relação de acesso entre os domínios dos operadores envolvidos, o que nem sempre é desejável, devido aos direitos de acesso.

Cada mapeamento em um domínio possui permissões associadas, que limitam as operações que podem ser realizadas por uma referência que penetre no domínio por aquela conexão (um mapeamento de endereços constitui um tipo especial de conexão, utilizando endereços físicos, e pode ser percorrida apenas por referências). Os direitos indicam se a referência pode alterar algo no domínio, se pode obter informações sobre os operadores do domínio, se pode multiplicar-se dentro do domínio e se poderá acessar outros domínios que estão conectados ao primeiro. Além disso, a relação de posse habilita a faculdade de ajustar permissões no sentido do domínio pai para o domínio filho, enquanto que a relação de acesso sempre desabilita essa possibilidade. Toda vez que uma referência necessita resolver um mapeamento para se movimentar no grafo, ela só perde direitos, não podendo recebê-los de volta.

2.4. Referências e Manipuladores

As referências podem apontar para operadores ou conexões, tendo como atributos seus direitos e um endereço lógico, seja absoluto ou relativo. O endereço de uma conexão é o endereço de seu operador mais sua posição dentro da lista de conexões que o operador possui.

Os operadores que manipulam referências podem: criar uma referência, ler atributos do objeto a qual se refere, criar um operador junto a uma conexão livre, criar conexões livres em operadores com número variável de conexões, como os domínios, destruir operadores, destruir conexões, movimentar referências entre as conexões e operadores, duplicá-las e destruí-las. Para as tarefas de criação e destruição de operadores, são necessárias referências a um tipo especial de operador, que denota uma região de memória livre.

2.5. Slots e instanciação de funções

Slots são o mecanismo Calebe para a sincronização quando do envio e recebimento de dados agrupados. Desse modo, servem para a chamada e retorno de métodos ou funções, além de acesso sincronizado a estruturas de dados compostas. Um *slot* pode ser considerado um agrupamento de conexões, cada uma com seu tipo de dados e sentido, possuindo, portanto, um tipo que é o resultado da composição dos tipos de suas conexões e o sentido destas. A ligação de dois *slots* complementares permite a troca de um pacote de dados entre o grupo de conexões de saída de um e o grupo de entrada de outro, e vice-versa.

Um caso relevante é o de um domínio com um único *slot*. Este se comportará como um único operador *dataflow*, consumindo suas entradas quando da ativação do *slot*, e retornando o resultado de sua operação em sua saída. Se este for anotado como um domínio instanciável, a ativação do *slot* irá alocar espaço de memória dinamicamente para as conexões do domínio, permitindo a execução paralela de mais de uma instância do mesmo, e recursão. Os valores comunicados entre os operadores do domínio instanciado serão armazenados no bloco, e a existência de dois valores no mesmo bloco, do mesmo operador, irão disparar a execução deste. O processo é inspirado em [IAN90], onde o uso de *tags* do modelo *dataflow* dinâmico clássico [ARV82] é substituído pelo armazenamento em pilha. O bloco será liberado quando ocorrer o retorno de valores por parte do *slot*.

O processo pode ser generalizado para domínios quaisquer, em que se define um ou mais *slots* com a propriedade de ser um instanciador. A principal dificuldade nesse caso é garantir que não haverá comunicações pendentes em outros *slots*, quando ocorrer o retorno de valores no *slot* que provocou a criação de uma nova instância. Isso é importante quando da definição de métodos recursivos em objetos, pois estes terão obrigatoriamente conexões com a estrutura de dados que contém o estado do objeto.

2.6. Sincronização de acessos a dados e funções

Há dois modos de se efetuar sincronização de acessos a dados ou funções, dependendo se este acesso se dará através de referências ou de *slots*. O primeiro caso caracteriza a manipulação direta dos operadores de um domínio, e o segundo, o acesso a uma interface padronizada, seja de um serviço ou de uma estrutura de dados compartilhada.

Um operador, conexão ou valor pode comportar apenas uma referência de cada vez, impedindo conflitos (*races*) causados por acessos concorrentes. O mesmo pode ser estendido a uma estrutura de dados complexa, em forma de árvore ou grafo, pois, a medida que uma referência segura caminha, ela bloqueia os operadores e conexões para remoção, garantindo que o caminho não será alterado. Dessa forma, tem-se duas possibilidades: a referência encontra o objeto que necessita ser modificado e obtém acesso exclusivo a ele, ou não o encontra, porque outro processo modificou a estrutura antes. É eliminada a possibilidade de um processo manipular a folha de uma árvore, por exemplo, e ao mesmo tempo outro processo remover um dos nodos dessa árvore, gerando inconsistência entre as informações que cada um dos processos possui a respeito da estrutura. Outras possibilidades relacionadas ao uso de direitos e conexões fazem parte do modelo, como uma referência de escrita que possui acesso único a um domínio, bloqueando para qualquer outra referência, ou permitindo apenas referências de leitura.

A sincronização através de *slots* é mais segura, pois realiza o encapsulamento do dado ou serviço a que se pretende o acesso. Deve-se utilizar *slots* em todos os casos em que o encapsulamento for possível, e referências apenas quando é necessário manipular ou mesmo construir um estrutura. Quando vários processos possuem conexão a um mesmo *slot*, há duas alternativas: a função disponibilizada pelo *slot* é instanciável, e existirão tantas cópias desta quanto necessário pelos clientes, ou ela não é instanciável, e apenas um cliente poderá ter acesso de cada vez. Nesses casos, um operador será responsável por garantir o acesso de um cliente de cada vez ao *slot*, realizando um mapeamento de 1:N entre o serviço e os clientes. Calebe possui dois operadores desse tipo, um com prioridades fixas e outro com prioridade circular dos clientes. O comportamento dos operadores pode ser modificado agregando um contexto a eles, para casos onde são necessárias outras políticas de sincronização.

2.7. Resposta a Eventos através de Contextos e Extensão do Modelo Calebe

Um Contexto é responsável por ligar anotações a um operador, uma conexão ou a um valor. Podem ser de dois tipos: predefinidas em Calebe, ou extensões. São implementadas através de conexões. Portanto, como os domínios, os contextos são uma lista de tamanho variável de conexões, mas, os primeiros tem o papel de aplicar estrutura ao sistema de operadores *dataflow*, e os contextos modificam o comportamento do objeto ao qual estão associados. Os domínios são um mapeamento entre endereços lógicos e físicos, e os contextos são um mapeamento entre eventos e conexões com tratadores de eventos.

O tratamento de erros em Calebe é feito através da sinalização de eventos predefinidos, identificado o contexto do ambiente em que ocorreram. Um erro de divisão por zero, por exemplo, é sinalizado ao contexto do operador de divisão, se este existir e a conexão ligada a este evento for válida, caso contrário o evento é sinalizado ao domínio que possui o operador, se este também não estiver preparado, ao seu domínio pai, e assim por diante. Da mesma forma, os contextos permitem modificar a resposta de uma entidade, estabelecendo a resposta a eventos como a tentativa de destruição através de uma referência, ou a inclusão de uma nova conexão dentro de um domínio.

Além do tratamento de erros e ações, também está definida em cada contexto uma conexão com um serviço de resolução de nomes. Este é fundamental para o estabelecimento do ambiente em que um módulo ou aplicativo é carregado dentro de um sistema Calebe. A ligação dinâmica é

realizada com consultas ao servidor de nomes, requisitando a conexão com *slots* de outros módulos responsáveis pela função ou pelo dado requisitado. É a única forma de um programa realizar a conexão com outro programa, se o caminho entre os dois tiver de passar através do domínio raiz, pois qualquer conexão que descer pela árvore até a raiz perde todos os seus direitos.

O uso dos eventos definidos pela extensão de um contexto ocorre com um operador especial que atua sobre uma referência ao objeto que possui o contexto. Esse operador toma a referência, o descritor do evento, e uma referência que será passada como parâmetro, e envia o parâmetro através da conexão do contexto identificada pelo descritor. Essa é uma das formas de extensão de Calebe com o uso de contextos, mas não a única.

O modo mais poderoso de extensão do modelo se dá pela sobrecarga de ações predefinidas, pois, nesse caso, a identificação de eventos é realizada de forma automática pela máquina. Pode-se utilizar um domínio com uma dada estrutura de dados como uma representação de um outro domínio. Transformações na representação geram eventos, os quais, através do contexto associado, modificam a estrutura do domínio escravo. Esse mecanismo permite a criação de diversos níveis de abstração, em que as modificações num dos níveis superiores repercutem nos inferiores, de forma transparente. Dessa forma, um compilador implementado sobre Calebe irá herdar a capacidade do modelo de interagir com o código gerado em tempo de execução, podendo funcionar como compilador ou interpretador, quando necessário. Essa discussão será retomada adiante, no tópico sobre as vantagens do modelo para implementação de linguagens multiparadigma.

Quando um contexto está isolado, a regra para o tratamento de um evento não definido é aquela já citada, em que é utilizado o contexto do domínio mas próximo associado. Entretanto, essa característica pode ser modificada, utilizando-se outro contexto como tratador de eventos pai. O mecanismo é semelhante à herança numa hierarquia de classes: quando o evento está definido no contexto presente, ele será utilizado, senão será utilizado o contexto pai. Assim, um domínio mantido por uma aplicação dentro de uma árvore de domínios que representa a relação de posse, mas não a relação de significado, pode ter seu contexto herdado de uma outra árvore que representa o tipo de cada domínio. Esse mecanismo será exemplificado com uma possível implementação de linguagem orientada a objetos, no tópico correspondente.

3. Compondo abstrações para implementação de paradigmas

3.1. Modos de Composição

Um dos requisitos do modelo Calebe é o suporte à implementação das semânticas dos diversos paradigmas, caracterizadas por unidades de abstrações básicas, como objetos, funções, cláusulas lógicas e comandos. A abordagem adotada é a utilização dos operadores Calebe como formas de abstração básicas capazes de compor uma ampla variedade de abstrações de alto nível. Dependendo do estilo da linguagem multiparadigma, podem ser utilizadas várias unidades diferentes, beneficiando-se de uma base comum que facilita seu funcionamento conjunto, ou utiliza-se uma única unidade de abstração, beneficiando-se da riqueza e flexibilidade dos operadores Calebe, resultando numa implementação simples de um conceito com grande densidade de expressão, como *streams* de G [PLA91], tarefas de Oz [MUL95], e entes de Holoparadigma [BAR99].

Para alcançar esse objetivo, devem ser utilizados diferentes modos de composição de abstrações, todos baseados nos quatro grupos de operadores propostos em Calebe: *slots*, domínios, manipuladores de referências, e contextos.

1) **Encapsulamento** – o agrupamento de conexões em *slots* permite o encapsulamento de um serviço, de forma semelhante a um procedimento, uma função, um método ou uma cláusula lógica. O usuário do serviço precisa conhecer apenas a interface do mesmo, sem se preocupar com a implementação.
2) **Agrupamento de valores** –os *slots* realizam o agrupamento de valores, permitindo a composição de estruturas de dados complexas.
3) **Composição** – a relação de posse entre domínios determina a estrutura do sistema, definindo uma relação de controle entre programas e seus componentes. Pode-se identificar cada domínio com um componente, e utilizar níveis arbitrários de composição.
4) **Compartilhamento** – a relação de acesso entre domínios permite que um domínio compartilhe um componente de software com outro domínio que não o possui, mantendo os direitos de uso adequados para a lógica da aplicação.
5) **Tratamento de exceções e eventos** – os contextos dão significado aos eventos gerados pela execução da máquina Calebe, auxiliando na separação entre diversos aspectos de um componente de software. O critério mais simples de classificação é a atividade normal de um componente e a atividade excepcional.
6) **Reuso de comportamentos** – a relação hierárquica entre contextos fornece meios para a separação de aspectos de uma forma mais sofisticada, onde são separados a estrutura dos componentes de seu comportamento. Em conjunto com a abstração de compartilhamento, a hierarquia de contextos permite a criação de componentes com diversos tratadores de eventos separados de acordo com os aspectos de seu comportamento, sem depender dos tratadores utilizados no domínio que o possui, nem quais são utilizados em seus subcomponentes.
7) **Manipulação de estruturas como valores** – uma referência é um valor, e permite a operação com operadores, conexões e conjuntos destes como se fossem valores. Essa forma de abstração abre um componente de software à manipulação externa, ao contrário do encapsulamento. Entretanto, a modificação direta da estrutura de um componente é o modo mais poderoso de modificar estruturas de dados complexas e que possuem subcomponentes encapsulados. Também é o meio para criação de novos componentes, e para a realização de computação sobre componentes, ou metacomputação, através da obtenção e alteração de propriedades dos componentes.
8) **Computação em diferentes níveis de abstração** – o uso de contextos permite a sobrecarga de eventos como a atuação de uma referência sobre uma conexão. Este contexto pode determinar que a conexão significa o processo de transformação de um valor em um nível de abstração inferior, e a destruição da conexão significa a destruição do processo, a qual o contexto da conexão efetuar utilizando referências, antes dele próprio ser eliminado. Essa ação é transparente para a computação realizada no nível mais elevado, cujo objetivo era apenas alterar uma estrutura de dados, que poderia simbolizar uma função de alta ordem, por exemplo. A resolução do valor da função de alta ordem (uma função que pode ser executada diretamente pela máquina) seria simplificada. Esta seria realizada por metacomputação sobre a estrutura que a simboliza, e não sobre os operadores diretamente executados pela máquina Calebe, os quais podem ser em número elevado. A quantidade de níveis de abstração depende unicamente da estrutura do sistema implementado, sendo a computação em cada um deles traduzida para ações no nível inferior, até que se atinja a execução de operadores básicos Calebe.

3.2. Exemplo de uma Implementação de Orientação a Objetos em Calebe

A figura 2 exemplifica a estrutura de um programa orientado a objetos implementado em Calebe. Não é objetivo desta seção discorrer sobre as decisões de projeto envolvidas. A abordagem apresentada não é a mais simples nem a mais sofisticada, apenas ilustra o uso de formas de abstrações do modelo na composição dos conceitos de orientação a objetos. Projetistas de linguagens implementadas sobre Calebe podem fazer escolhas diferentes, dependendo de seus objetivos.

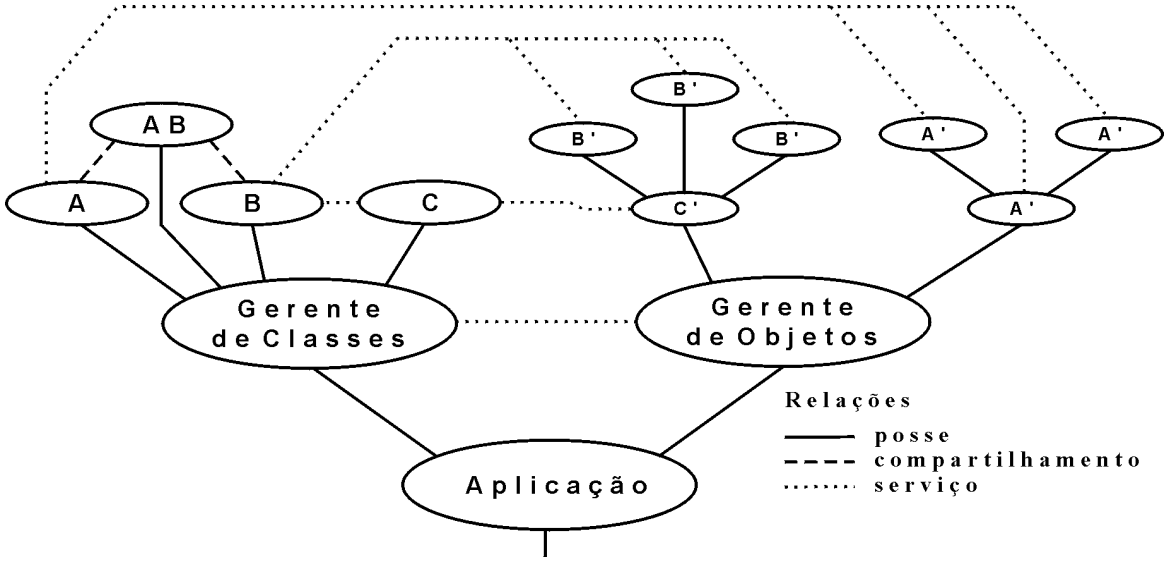

Figura 2 – Representação de uma hierarquia de classes e de um conjunto de objetos compondo uma aplicação.

Cada elipse na figura mostra um domínio. As linhas sólidas representam a relação de posse, que compõe uma árvore, cuja raiz não está representada. Assim, a aplicação tem total controle sobre as classes e objetos que a contém, enquanto que o Gerente de Classes possui sobre as classes, e assim por diante. Nota-se que cada unidade de abstração, seja uma classe, um objeto, ou mesmo um aplicativo, é representada por seu domínio, o qual serve de delimitador, e permite o uso dos direitos de acesso.

Cada objeto guarda o seu estado interno, e uma conexão com sua classe correspondente, a qual implementa seus métodos. O objeto tem uma conexão de prestação de serviços com sua classe, através de *slots*, o que garante o encapsulamento da classe, e a implementação dos métodos do objeto como serviços. Observa-se isso através das diversas linhas pontilhadas ligando cada objeto A', B' e C', às classes A, B e C, respectivamente.

O exemplo acima possui duas classes A e B, com o mesmo conteúdo, mas uma representa objetos não relacionados, e a outra uma árvore de objetos. O comportamento comum de ambos os objetos se encontra na classe ancestral AB. A classe C serve apenas para organizar uma coleção de objetos B. Assim, a classe C utiliza serviços da classe B, como a criação e destruição de objetos.

Nota-se que a relação entre uma classe e sua ancestral é de compartilhamento. A estrutura interna da classe AB é visível para as classes A e B, mas outras classes e objetos que utilizam métodos da mesma só têm acesso a uma parte de sua interface externa. Todas as classes são administradas pelo Gerenciador de Classes, e a hierarquia de classes cresce como um grafo de compartilhamentos entre as classes pertencentes ao Gerenciador, permitindo herança múltipla. A decisão de conflitos e a orientação da relação de herança nesse grafo é determinada pelo uso de direitos de acesso nas conexões de compartilhamento.

O Gerenciador de Objetos e o Gerenciador de Classes trabalham em conjunto. A conexão entre eles é de solicitação de serviços. Quando a aplicação necessita a criação de um novo objeto, o Gerenciador de Objetos necessita obter a estrutura deste junto ao Gerenciador de Classes, e conectar o novo objeto à sua classe. Caso a estrutura de classes seja modificada em tempo de execução, a propagação dessa mudança deve ser sincronizada entre o Gerenciador de Classes, capaz de determinar as relações de dependências, e o Gerenciador de Objetos, capaz de atuar sobre os objetos ativos e traduzir as relações de dependências numa ordenação lógica das alterações a serem realizadas.

Isso permite o uso de metacomputação em tempo de execução, não apenas como forma de obter informações sobre os objetos, mas de alterar seu comportamento. A obtenção de informações se dá de três formas:

- uso de métodos que retornam informações armazenadas nos objetos ou nas classes;
- uso de referências para percorrer a estrutura de objetos e classes, e obter informações diretamente sobre os operadores e conexões que os compõe, inclusive a que está armazenada em seus contextos;
- uso de referências para acessar conexões customizadas de seus contextos.

A árvore de contextos é tal que o contexto de cada objeto é formado primeiramente pelo seu próprio, depois pelo contexto de sua classe, e depois pelo contexto do Gerenciador de Objetos. O contexto de cada classe é formado pelo seu contexto, por um ordenamento de prioridades entre suas classes ancestrais, e pelo Gerenciador de Classes. Utilizando-se diferentes níveis de abstração em orientação a objetos, pode-se ter objetos que encapsulam toda uma hierarquia de classes ou uma aplicação, e classes de hierarquias ou de aplicações.

4. Exploração de paralelismo e sistemas distribuídos

4.1. Implementação de Calebe em Sistemas Paralelos e Distribuídos

Assim como os operadores Calebe prestam-se a composição de abstrações, eles também são combinados para implementar diversas formas de particionamento, sincronização e comunicação entre unidades de abstração.

A característica chave para a implementação de paralelismo em Calebe é o uso de domínios para a composição de espaços de endereçamento estruturados. Como estes são o dispositivo básico de particionamento dos componentes de software, também o são para balanceamento de carga, e compartilhamento de dados.

Assim, a abstração de compartilhamento de dados ou processos é implementada através da troca de mensagens, em dois modos diferentes, dependendo se o espaço lógico de endereçamento de ambas é unificado ou não.

No caso de espaços de endereçamento disjuntos, duas máquinas Calebe irão possuir um domínio compartilhado, onde irão registrar todas as interfaces de seus processos capazes de se comunicarem. Modificações nesse domínio seguem um protocolo de consistência seqüencial, mas o modelo de comunicação é a troca de valores *dataflow*, ou seja, é transparente para os processos envolvidos. Caso seja necessário um gerenciador do sistema, responsável por organizar aspectos como balanceamento de carga e replicação, ele deve ser replicado em cada máquina, e se comunicar com os demais gerenciadores através de uma interface para cada um.

No caso de um espaço de endereçamento unificado, o domínio raiz das máquinas é compartilhado, e as diversas máquinas são efetivamente uma só. Utiliza-se um protocolo de consistência seqüencial para realizar modificações no domínio raiz, que é replicado em todas as

máquinas, mas os demais domínios utilizam os mecanismos de sincronização do modelo, como se executassem numa mesma máquina.

Numa máquina com endereçamento unificado, existe uma única cópia de cada domínio no sistema, a não ser que este seja explicitamente replicado. Para isto, o endereço físico permite a identificação da máquina em que o mesmo se encontra. No caso de um domínio replicado, os operadores do mesmo possuem o mesmo endereço lógico em cada máquina, mas endereços físicos diferentes. Desse modo, a localização de um operador é transparente ao sistema, ao não ser no caso em que é necessário conhecê-la, como no processamento dos gerenciadores de balanceamento e de replicação que forem utilizados.

Um operador pode ser identificado com respeito a máquina em que se encontra, utilizando-se o endereçamento absoluto acrescido da numeração da máquina virtual na rede. Esse mecanismo cria um domínio abstrato, situado abaixo dos domínios raiz de cada máquina, com a função única de identificar a localização de cada máquina real.

4.2. Implementação da Máquina Virtual Calebe numa LAN

A implementação da máquina virtual prevê a exploração de paralelismo em multiprocessadores com o uso de *threads*, todas atuando sobre o grafo de operadores na memória principal, e atuando como uma única máquina Calebe. Nesse caso, cada uma possui um espaço de endereçamento próprio, identificado por seu domínio raiz.

Calebe foi designado para ser implementado diretamente sobre uma rede capaz de troca de mensagens ponto-a-ponto, por ser essa a alternativa mais simples e mais portável. Implementações específicas poderão fazer uso de outras abstrações, como memória distribuída compartilhada, para atender a objetivos específicos, como desempenho ou custos de projeto. Entretanto, o uso de uma interface de troca de mensagens atende os requisitos de portabilidade do projeto.

5. Conclusões

Foi apresentada uma máquina virtual capaz de explorar paralelismo, de executar de maneira distribuída e dar suporte a semântica de diversos paradigmas. Esta se baseia num conjunto de operadores, capazes de executar concorrentemente, atuando dentro de espaços de endereçamento, responsáveis pela distribuição tanto de dados quanto computação.

A proposta da máquina virtual Calebe é demonstrar a possibilidade da abstração do hardware paralelo e criação de um ambiente comum para execução de programas escritos em diversas linguagens e estilos diferentes. Esse projeto de universalidade tem como desafios a variedade de arquiteturas existente e os problemas encontrados com a integração de estruturas com semânticas diferentes.

Embora o modelo da máquina virtual seja definido a partir de seus operadores e estruturas de dados, os quais foram apresentados neste artigo, Calebe deve ser estendido de forma a se tornar um ambiente para o desenvolvimento de linguagens multiparadigma e exploração de paralelismo. Para isso, é necessário definir quais anotações são fornecidas pelo compilador e como a máquina irá interpretá-las em tempo de execução. Deste modo, um dos trabalhos futuros consiste no aprimoramento do conjunto de operadores escolhidos – um processo de amadurecimento que envolve interação entre o projeto da máquina virtual e de compiladores para ela. Este processo se dará com a implementação de linguagens e dos respectivos compiladores sobre Calebe.

Para se tornar um ambiente completo, também deve ser definido como diferentes estratégias de balanceamento e mobilidade serão implementadas e escolhidas pelo compilador e/ou pela máquina. Considerando-se, por exemplo, a grande variedade de estratégias de balanceamento

[JAC99] e mobilidade [ROY97] disponíveis, observa-se que esta é uma área de pesquisa ativa em processamento paralelo, e ainda não há um consenso a respeito das melhores estratégias a serem utilizadas. Estes e outros componentes do ambiente de execução deverão ser adaptáveis à arquitetura alvo, às características da aplicação, e às novas descobertas na área. Assim, como complemento do desafio de construir uma máquina virtual para linguagens multiparadigmas e sistemas paralelos, surge a necessidade de uma máquina virtual capaz de evoluir com seu ambiente. Essa necessidade está no âmago da integração de diferentes tecnologias em desenvolvimento, e ainda não foi solucionada.

6. Bibliografia

[ARV82] ARVIND; GOSTELOW K. P. **The U-Interpreter**. *Computer*, 15(2), February 1982.

[ARV89] ARVIND; NIKHIL Rishiyur S.; PINGALI Keshav K. **I-Structures: Data Structures for Parallel Computing**. *ACM Transactions on Programming Languages and Systems*, v. 11, n. 4, October 1989, pp. 598-632.

[BAR98] BARBOSA, Jorge L. V.; GEYER, Cláudio F. R. **Taxonomia Multiparadigma**. In: Congreso Argentino de Ciencias de la computación, 4, 1998, Neuquén. *Proceedings...* Neuquén: Universidad Nacional del Comahue, octubre 1998. 1162p. p.1162 (poster).

[BAR99] BARBOSA, Jorge L. V. **Princípios do Holoparadigma**. Porto Alegre: CPGCC da UFRGS, 1999. 75p. (Trabalho Individual, 748).

[DEN74] DENNIS, J. B. **First Version of a Data Flow Procedure Language**. In *Lecture Notes in Computer Science*, volume 19, pages 362-376. Springer-Verlag, 1974.

[IAN90] IANNUCCI, Robert A. Parallel Machines: **Parallel Machine Languages: The Emergence of Hybrid Dataflow Computer Architectures**. Kluwer Academic Publishers, 1990.

[JAC99] Jacob, Josep C.; Lee, Soo-Young. **Task Spreading and Shrinking on Multiprocessor Systems and Networks of Workstations**. *IEEE Transactions on Parallel and Distributed Systems*, New York, v.10, n.10, October 1999, p. 1082-1101.

[JAG95] JAGANNATHAN, R.; **Dataflow Models**. Computer Science Laboratory, SRI International, 1995.

[MUL95] MULLER, Martin; MULLER, Tobias; ROY, Peter V. **Multiparadigm Programming in Oz**. In: SMITH, Donald; RIDOUX, Olivier; ROY, Peter V. (eds.). *Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor of Prolog*. Portland, Oregon, December 1995.

[PLA91] PLACER, John. **The Multiparadigm Language G**. *Computer Languages*, v.16, n.3/4, p.235-258, 1991.

[POL99] POLLETO; Massimiliano; HSIEH, Wilson. et al **'C and tcc: A Language and Compiler for Dynamic Code Generation**. *ACM Transactions on Programming Languages and Systems*, v. 21, n. 2, March 1999, pp. 324-369.

[ROY97] ROY, Peter Van; HARIDI, Seif; BRAND, Per; et al. **Mobile Objects in Distributed Oz**. *ACM Transactions on Programming Languages and Systems*, v. 19, n. 5, September 1997, 804-851.

[VRA95] VRANES, Sanja; STANOJEVIC, Mladen. **Integrating Multiple Paradigms within the Blackboard Framework**. *IEEE Transactions on Software Engineering*, New York, v.21, n.3, March 1995, p.244-262.