RStabilis: Uma Máquina Reflexiva de Busca

Rômulo César Silva

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

RStabilis: Uma Máquina Reflexiva de Busca

Rômulo César Silva

Setembro de 1997

Banca Examinadora:

- Luiz Eduardo Buzato (Orientador)
- Maria Lúcia B. Lisboa
  Instituto de Informática - Universidade Federal do Rio Grande do Sul
- Cecília Mary Fischer Rubira
  Instituto de Computação - Unicamp
- Eliane Martins
  Instituto de Computação - Unicamp

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Silva, Rômulo César

Si38r RStabilis: uma máquina reflexiva de busca / Rômulo César Silva -- Campinas, [S.P. :s.n.], 1997.

Orientador : Luiz Eduardo Buzato

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Programação orientada a objetos (Computação). 2. Sistemas distribuídos (Computação). 3. Tolerância a falhas (Computação). I. Buzato, Luiz Eduardo. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

RStabilis: Uma Máquina Reflexiva de Busca

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rômulo César Silva e aprovada pela Banca Examinadora.

Campinas, 9 de setembro de 1997.

Luiz Eduardo Buzato
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 01 de setembro de 1997 pela Banca Examinadora composta pelos Professores Doutores

Profa. Dra. Maria Lúcia Blanck Lisbôa

Profa. Dra. Cecília Mary Fischer Rubira

Prof. Dr. Luiz Eduardo Buzato

Como faço uma escultura?
Simplesmente retiro do bloco de mármore
tudo o que não é necessário.

Michelângelo.

Agradecimentos

Eu agradeço a Deus pela força e perseverança a mim concedidas para a realização deste trabalho, que se fizeram presentes nos familiares, amigos e colegas que sempre me estimularam. Agradeço também ao meu orientador, Luiz Eduardo Buzato, pela sua paciência e dedicação. A meus pais, pelo seu apoio constante e dedicação. Aos amigos Islene e Oliva, que deram sugestões valiosas para este trabalho. A todos os professores do Instituto de Computação. E ainda, aos amigos e colegas do mestrado, em especial: Sand, Walter, Solange, Sávio, Delano e Dinalva.

Resumo

A crescente complexidade e os requisitos de novas capacidades (interoperabilidade, reutilização, escala, transparência, etc) por parte de aplicações impulsionam a pesquisa de técnicas de construção de componentes de software que atendam efetivamente a essas capacidades. Dentro deste contexto, procura-se atualmente somar técnicas já existentes de propósitos distintos, mas que combinadas permitem minimizar a complexidade de construção desses componentes. Uma das grandes dificuldades no desenvolvimento de software é a separação satisfatoria das atividades externas do sistema, relacionadas ao domínio da aplicação, das atividades internas, relacionadas à gerência da aplicação. Reflexão computacional é uma técnica que permite separar os mecanismos que controlam as atividades internas e externas da aplicação, de maneira que qualquer dos mecanismos possa ser alterado de forma autônoma e quase transparente.

Esta dissertação pesquisa a utilização de reflexão computacional em Stabilis, uma ferramenta para construção de máquinas de busca de objetos. Máquinas de busca de objetos são sistemas de meta-informações que provêem interface orientada a objetos para informações contidas em ambientes distribuídos de grande escala. RStabilis é a versão reflexiva de Stabilis, produzida através da inclusão de mecanismos de reflexão computacional em Stabilis. RStabilis permite a fácil separação dos mecanismos de controle de atividades internas e externas.

Abstract

The increasing complexity and the requirement for new capabilities (openess, reusability, scalabity, etc) in applications motivate research of technics for construction of software components that effectively provide these capabilities. In this context, there has been much research towards combining distinct software engineering technics to construct such software components. One of the main obstacles to the construction of flexible components is the separation of the mechanisms that control external activities, related to application's domain, from the mechanisms that control the internal activities of an application. Computational reflection is a software construction technic that seems to be very promising to implement separation of control mechanisms (internal from external), in a way that the internal can be modified without affecting the external, and vice-versa.

In this thesis, we research the utilization of computational reflection in Stabilis, a toolkit for construction of object engine. Object engine are metainformation systems that provide object-oriented interface to information contained in network resources of large distributed enviroments. RStabilis is the reflexive version of Stabilis, it allows a clear separation of control mechanisms in the implementation of software applications.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A crescente complexidade e os requisitos de novas capacidades (interoperabilidade, reutilização, escala, transparência, etc) por parte das aplicações impulsionam a pesquisa de técnicas de construção de componentes de software que atendam efetivamente a essas capacidades. Dentro deste contexto, atualmente procura-se somar técnicas já existentes de propósitos distintos, mas que agrupadas permitem minimizar a complexidade de construção dos componentes.

A técnica de orientação a objetos é usada cada vez mais para modelagem e construção de componentes de software pela sua facilidade de abstração. Tolerância a falhas consiste de um conjunto de técnicas que permite detectar e tratar situações anormais no comportamento dos componentes, aumentando assim a confiabilidade e disponibilidade do sistema.

Uma das grandes dificuldades no desenvolvimento de software é conseguir separar satisfatoriamente os mecanismos que controlam as atividades externas do sistema, relacionadas ao domínio da aplicação, dos mecanismos que controlam as atividades internas, relacionadas com a administração do sistema. Por exemplo, numa ferramenta para captura de diagramas de modelos orientados a objetos, as atividades externas referem-se a permitir ao usuário desenhar os modelos e fazer geração de código automática enquanto as atividades internas referem-se aos algoritmos de traçado usados para apresentar os modelos. É desejável separar essas atividades tal que os algoritmos de traçado possam ser substituídos por novos e mais eficientes, sem no entanto afetar a interface com o usuário e vice-versa.

Paralelamente às técnicas de orientação a objetos e tolerância a falhas, reflexão computacional pode ser usada na construção de componentes de software. Seu objetivo é permitir a separação entre atividades internas e externas, de maneira que as internas possam ser alteradas sem afetarem a externas. Reflexão computacional tem sido aplicada a vários domínios de aplicações, desde linguagens de programação até sistemas operacionais.

Sistemas de meta-informação permitem ao usuário representar os modelos de suas aplicações, fazer geração de código automática, além de servirem como documentação da aplicação.

Este trabalho pesquisa a utilização de reflexão computacional em um sistema de meta-informação — Stabilis [Buz94, Cal96] — visando facilitar o desenvolvimento de aplicações flexíveis. Stabilis é uma ferramenta para a construção de aplicações distribuídas orientadas a objetos tolerantes a falhas, provendo persistência e ações atômicas aos objetos. Aplicações típicas que podem ser construídas usando Stabilis são máquinas de busca de objetos. Uma máquina de busca de objetos é um sistema de meta-informação cujo propósito é prover uma interface orientada a objeto para a informação contida em ambientes distribuídos de grande escala.

A estruturação desta dissertação é a seguinte:

- Capítulo 2: **Fundamentos**. Contém os conceitos básicos sobre as técnicas de orientação a objetos, tolerância a falhas e reflexão computacional, apresentados devido ao fato do ambiente alvo usar estas três técnicas. Mostra também as principais vantagens de utilização dessas técnicas em geral.

- Capítulo 3: **Trabalhos Relacionados**. Contém um "survey" de linguagens de programação reflexivas e uso de métodos em consultas de bancos de dados. O objetivo do "survey" é coletar informações relevantes de experiências anteriores para obtenção de uma visão crítica sobre projeto/implementação de reflexão computacional e invocação de métodos em consultas visando sua aplicação no ambiente alvo. Para isto, é feita uma discussão sobre os modelos de reflexão implementados pelas linguagens e a relação entre os modelos de objetos, e o uso de métodos em consultas de bancos de dados orientados a objetos.

- Capítulo 4: **Stabilis**. Contém uma descrição detalhada da arquitetura de software do ambiente alvo, os principais elementos que compõem uma aplicação típica construída sobre Stabilis e também a semântica de invocação de métodos em Stabilis.

- Capítulo 5: **RStabilis**. RStabilis é evolução de Stabilis, que implementa um modelo de reflexão usando os recursos existentes no próprio Stabilis. Apresenta vários modelos de implementação de reflexão computacional possíveis em Stabilis. Traça comparações entre eles, mostrando suas vantangens/desvantagens e as ocasiões em que melhor se aplicam. O capítulo também apresenta um exemplo de aplicação que pode ser construída usando RStabilis, com o objetivo de demonstrar o uso do modelo de reflexão implementado por RStabilis.

- Capítulo 6: **Conclusão**. Assinala as principais contribuições desta dissertação, e trabalhos futuros, tanto de aplicação geral como específicos do ambiente alvo usado.

Capítulo 2

Fundamentos

2.1 Introdução

Orientação a objetos tem se mostrado uma técnica bastante útil no desenvolvimento de sistemas, incentivando a reutilização e melhor estruturação de seus componentes, devido a facilidade de abstração provida pelo conceito de objeto. Técnicas de tolerância a falhas têm sido usadas na construção de sistemas complexos, com o objetivo de detectar e tratar falhas, já que esses sistemas estão normalmente sujeitos a ocorrência de falhas. Ao lado dessas duas técnicas, reflexão computacional auxilia na organização interna do sistema, permitindo separar os mecanismos que controlam as atividades internas e externas, tornando-o mais flexível. Este capítulo mostra os conceitos básicos dessas três técnicas: orientação a objetos, tolerância a falhas e reflexão computacional, que formam o escopo principal do trabalho apresentado nesta dissertação. Antes porém, são apresentadas algumas definições importantes relacionadas com estruturação de componentes de software.

2.2 Estruturação de Componentes de Software

Um sistema é um conjunto de componentes de software que operam de acordo com um projeto, ou seja, uma especificação. Componentes, por sua vez, são subsistemas: sistemas menores que são integrados para oferecem um conjunto de funcionalidades [Weg90]. A interface de um sistema define os serviços providos por ele. O domínio de um sistema é a sua área de aplicação. Por exemplo, o domínio de um sistema de informações geográficas são mapas.

A adaptabilidade de um componente de software está relacionada com a facilidade de sua alteração para a adição de novos serviços que atendam a requisitos específicos ditados por novas aplicações, e com a sua facilidade de reutilização. Várias técnicas

de construção de componentes de software foram propostas com a finalidade de torná-los adaptáveis. A terminologia usada na classificação dessas técnicas de construção de componentes de software é similar à usada na construção de componentes de hardware. Os componentes podem ser implementados como caixas-pretas ou caixas-brancas. Se os componentes são projetados usando técnicas que constroem caixas-pretas, então os usuários têm acesso somente às interfaces, ficando as implementações transparentes. Se ao contrário, os componentes são projetados usando técnicas que constroem caixas-brancas, a implementação é totalmente aberta, permitindo aos usuários alterá-la de forma irrestrita.

Nota-se que cada uma dessas técnicas possui vantagens e desvantagens. As técnicas de caixas-pretas tendem a produzir componentes menos adaptáveis, já que a implementação do sistema está fechada ao usuário. As técnicas de caixas-brancas produzem componentes mais adaptáveis. Mas não oferecem proteção contra alterações indevidas, já que a implementação encontra-se totalmente aberta ao usuário. Assim, é desejável utilizar um conjunto de técnicas de construção de componentes que combinem as melhores características de ambas abordagens. Para isto, o componente produzido com essas novas técnicas têm de ser estruturado adequadamente. Sua interface deve definir os serviços e permitir alterações de forma controlada em sua implementação, conseguindo assim o encapsulamento oferecido pelas por caixas-pretas e a adaptabilidade de caixas-brancas.

2.3 Orientação a Objetos

Os conceitos de orientação a objetos têm sido empregados em vários domínios de aplicações devido à facilidade de abstração provida pelo conceito de objeto. A seguir são apresentados os fundamentos de orientação a objetos, particularmente aplicados a linguagens de programação.

2.3.1 Objetos e Classes

Um objeto é uma entidade que encapsula uma estrutura de dados e operações. As operações, denominadas métodos, determinam a interface e o comportamento do objeto. Os atributos que representam o estado interno do objeto são chamados variáveis de instância. Os métodos de um objeto compartilham seu estado, tal que mudanças feitas por um método podem ser vistas pelos métodos subseqüentes. Operações fazem acesso ao estado do objeto através de referências à variáveis de instância do objeto. Orientação a objetos é tipicamente uma técnica de construção de componentes caixas-pretas, pois objetos só podem ser manipulados através de sua interface.

Objetos que possuem o mesmo comportamento são agrupados em uma única classe. Logo, uma classe descreve um grupo de objetos com propriedades similares, isto é, atri-

butos e operações de uma coleção de objetos. Por exemplo, considere uma classe cujo nome é Pessoa, com atributos *Nome* e *Idade*, e método *incrementa_idade*, representados na figura 2.1.

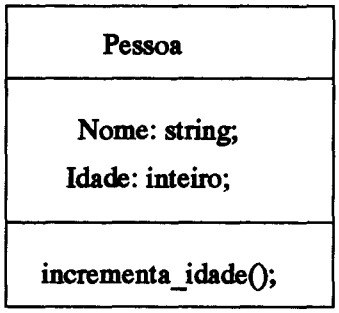

Figura 2.1: Classe Pessoa

É necessário prover um mecanismo para que os objetos possam interagir. Seguindo o modelo de computação orientada a objetos esta interação deve ser controlada, limitada à interface de método bem definida. Então, interação entre objetos é equivalente a invocação de métodos de outros objetos. Em orientação a objetos, o mecanismo de interação é conhecido como passagem de mensagem: sempre que uma operação é invocada, uma mensagem é enviada para aquele objeto com detalhes da operação requisitada [BHS]. Passagem de mensagem é conceitual, pois não necessariamente uma mensagem real é enviada para o objeto.

2.3.2 Herança

As estruturas de dados, algoritmos e interface deve ser especificada em termos dos componentes básicos do sistema. Em muitos casos, isto pode envolver esforço desnecessário devido a existência de outras classes similares no sistema. Portanto, é desejável fazer uso de classes existentes na especificação de novas classes. Uma nova classe pode ser descrita em termos de outra classe existente mas com modificações ou extensões para atingir os requerimentos da nova classe [BHS]. Herança é um mecanismo que permite reutilizar o comportamento de uma classe na definição de novas classes. Assim as classes podem ser especializadas, acrescentando novas variáveis de instância e métodos às novas classes. As classes das quais são herdados os atributos e métodos são chamadas de superclasses. As classes que herdam são chamadas de subclasses. Por exemplo, considere novamente a classe Pessoa (Figura 2.1) , pode-se ter uma subclasse de Pessoa, cujo nome é Funcionario, que acrescenta à classe Pessoa um atributo *Salario* e método *calcula_salario*. Objetos da classe Funcionario podem também utilizar o método *incrementa_idade* definido pela sua superclasse. A especificação de uma nova classe a partir de uma classe existente pode ser feita [BHS]:

- adicionando um novo comportamento: a nova classe acrescenta novos atributos e métodos além dos herdados da superclasse.
- mudando o comportamento: a nova classe redefine a implementação de métodos existentes na superclasse.
- retirando comportamento: menos comum, a nova classe é criada sem algumas funcionalidades da superclasse.

Linguagens e sistemas orientados a objetos oferecem diferentes maneiras de se fazer especialização. A mais comum é permitindo mudança e adição de comportamento.

2.3.3 Tipos

Tipagem é um conceito fundamental em computação relativamente fácil de entender, porém difícil de definir precisamente. Tipos vêm da necessidade de agrupar valores para manipulação, permitindo estabelecer propriedades. Um tipo é portanto uma descrição abstrata de um grupo de entidades relacionadas [BHS]. Por exemplo, o tipo *inteiro* denota entidades que exibem propriedades similares ao conceito matemático de números inteiros. Tipos são úteis em classificação, organização, abstração, e transformação de coleções de valores. Lidam com aspectos diferentes de abstração em linguagens de programação tais como abstração sobre propriedades, composição de tipos para formar novos tipos e proteção contra operações inválidas.

Valores em linguagens de programação podem ser entidades complexas consistindo de uma estrutura particular e um conjunto de semânticas associadas. A estrutura define a representação do valor enquanto as semânticas definem o modo como o valor é interpretado. Ambas estrutura e semânticas formam as propriedades do valor. Linguagens orientadas a objetos assumem que todas propriedades serão encapsuladas dentro de um objeto. Além disso, seu encapsulamento será protegido atrás de uma interface abstrata. O termo comportamento é usado para denotar esta interface abstrata. Tipo é então tratado somente em termos de seu comportamento. Assim, dois tipos são o mesmo se eles provêem o mesmo comportamento.

A distinção entre tipos e classes corresponde essencialmente a distinção entre estrutura e comportamento [Weg90]. Os propósitos principais de tipagem é a especificação da estrutura de expressões para checagem de tipo e especificação de comportamento para desenvolvimento de programas. Checagem de tipo é utilizada como prevenção de inconsistências de tipagem numa linguagem através da eliminação de erros de tipos, onde um erro de tipo é definido como sendo a aplicação de uma operação inválida sobre um valor. Podem acontecer em passagem de parâmetros para métodos ou em atribuições quando o tipo especificado e o valor fornecido são incompatíveis. Quanto ao momento em

que se faz a checagem de tipo, ela pode ser classificada em: estática, se feita em tempo de compilação; ou dinâmica se feita em tempo de execução [BHS]. Checagem de tipo estática requer que todas variáveis e expressões estejam ligadas a um tipo particular em tempo de compilação. Devido a essa restrição, geralmente adota-se checagem dinâmica, sem contudo permitir aos erros de tipo desenvolverem inconsistências. Isto é conseguido fazendo-se uma notificação da violação de tipo quando ela ocorre. A notificação é feita através de um mecanismo de exceção, isto é, erros de tipo são tratados da mesma forma que por exemplo divisão por zero.

2.3.4 Polimorfismo

Polimorfismo é a habilidade de um valor ter mais de um tipo [BHS]. Segundo a taxonomia de técnicas polimórficas de Cardelli e Wegner [Weg90], mostrada na figura 2.2 , polimorfismo pode ser classificado em: universal e ad hoc. Ad hoc é aquele em que as técnicas polimórficas aplicam-se somente sobre um número específico de tipos, de forma não sistemática. Ao contrário, em polimorfismo universal, as técnicas polimórficas aplicam-se a um conjunto infinito de tipos, de forma sistemática.

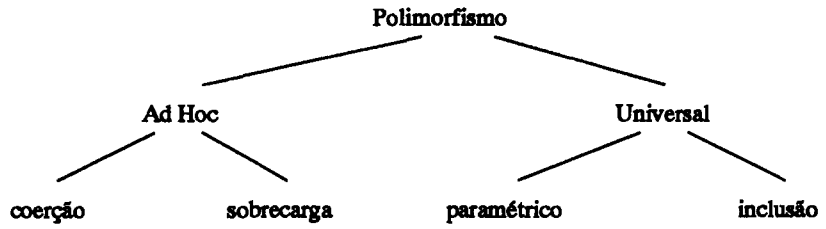

Figura 2.2: Classificação de polimorfismo segundo Cardeli e Wegner

Polimorfismo ad hoc classifica-se em:

- coerção: é uma técnica de polimorfismo que permite a conversão ou mapeamento interno entre tipos diferentes. Por exemplo, se uma função é definida sobre dois reais; e um inteiro e um real são fornecidos como parâmetros, então o inteiro será mapeado para real.

- sobrecarga: é uma técnica de polimorfismo que permite que o nome de uma função seja usado mais de uma vez com tipos diferentes de parâmetros.

Polimorfismo universal classifica-se em:

- paramétrico: uma função única (codificada uma única vez) será aplicada uniformemente sobre um intervalo de tipos. Funções paramétricas são também chamadas funções genéricas.

- inclusão: também permite uma função operar sobre um intervalo de tipos. Entretanto, este intervalo é determinado pelos relacionamentos de tipo-subtipo. Com o polimorfismo de inclusão, uma função definida sobre um tipo particular, irá também operar sobre quaisquer subtipos.

2.3.5 Delegação

Delegação [BHS] é um mecanismo para compartilhar comportamento, semelhante a herança, mas que opera diretamente entre objetos e não entre classes. Quando usa-se delegação os objetos são vistos como protótipos que delegam seus comportamentos para outros objetos. Essa relação de delegação pode ser estabelecida dinamicamente, enquanto que a relação de herança é estabelecida estaticamente. Cada objeto, também chamado "protótipo", define o seu próprio tipo. Em sistemas que utilizam delegação não há distinção entre classe e instância, pois não existe algo que defina abstratamente atributos para um objeto. Qualquer objeto pode ser definido em termos de outro. Ambos métodos e valores podem ser compartilhados quando um objeto os delega para um protótipo.

2.3.6 Metaclasses

Pode-se também considerar uma classe como um objeto que descreve outros objetos. Desta forma, metaclasse é definida como um objeto que descreve uma classe. A metaclasse possui uma única instância: a classe.

2.3.7 Classificação das Linguagens Orientadas a Objetos

A técnica de programação orientada a objetos implica que o domínio da aplicação é estruturado em termos de objetos e não de procedimentos. Entretanto, o conceito de objeto não é o mesmo para todas linguagens. Assim sendo, pode-se identificar, segundo Masini *et al.* [MNC+91], pelo menos três famílias de linguagens, cada uma enfatizando um ponto de vista particular sobre o conceito de objeto:

- Linguagens baseadas em classes: definem um objeto sob o ponto de vista estrutural, como um tipo de dados que encapsula uma estrutura e operações que podem ser aplicados a esta estrutura. Exemplos: C++, Smalltalk-80 e Java.

- Linguagens baseadas em "frames": definem um objeto, sob o ponto de vista conceitual, como uma unidade de conhecimento. Um "frame" é um agente que descreve uma situação padrão ou um objeto. "Frames" podem também ser organizados em uma hierarquia de herança, mas ao contrário de classes, cada objeto é uma representação do "frame" que o originou e um gerador de "frames" mais especializados.

Um "frame" é composto por "slots" que são descritos por facetas declarativas e procedurais. Facetas declarativas associam valores a "slots". Facetas procedurais são procedimentos ativados quando é feito um acesso ao "slot".

De maneira geral, linguagens baseadas em "frames" são aquelas que apresentam a tripla ("frame", "slot", faceta) como unidade básica de representação. Tais linguagens são projetadas principalmente para a representação do conhecimento. Exemplo: 3-KRS.

- Linguagens baseadas em agentes: definem um objeto sob o ponto de vista de uma entidade ativa e autônoma, chamada agente. A comunicação entre agentes é feita através de mensagens assíncronas. O comportamento de um agente é definido por um "script" que descreve como um agente reage a eventos provocados por outros agentes. Ao contrário de classes, agentes não são organizados hierarquicamente. Cada agente pode delegar mensagens que não consegue processar a outros agentes e desta forma, prover um mecanismo equivalente a herança (delegação). Devido ao seu aspecto dinâmico, linguagens baseadas nesse modelo são geralmente usadas para programação concorrente. Exemplo: ABCL/R2.

2.4 Tolerância a Falhas

Existem áreas onde sistemas computacionais realizam tarefas críticas tais como controle de tráfego aéreo e monitoração de pacientes hospitalares. Nesses sistemas um defeito [1] pode causar uma catástrofe. Portanto é desejável que esses sistemas sejam confiáveis, seguros, protegidos e tenham alta disponibilidade. Confiabilidade relaciona-se com a continuidade do serviço, segurança com a capacidade de evitar conseqüências catastróficas no ambiente, proteção com prevenção de acesso e/ou manipulação não autorizadas de informações, e disponibilidade com prontidão para uso.

Ocorre um defeito no sistema quando seu comportamento não é consistente com suas especificações [Jal94]. A complexidade e o grande número de componentes nos sistemas atuais reforçam a possibilidade de ocorrer falhas [2] no sistema. Há duas abordagens para solucionar esse problema: prevenção de falhas, e tolerância a falhas. O objetivo da primeira abordagem é tentar eliminar toda possibilidade de falha no sistema, aumentando assim sua confiabilidade. Já a segunda abordagem considera que dificilmente consegue-se retirar todas as possibilidades de falhas no sistema e logo deve-se ter meios de tomar ações após a detecção das falhas.

[1] Do inglês *failure*
[2] Do inglês *fault*

Um erro é aquela parte do estado do sistema que está sujeito a levá-lo a defeitos subseqüentes [Jal94]. Se existe um erro no estado do sistema, então existe uma seqüência de ações que pode ser executada e que irá levá-lo a um defeito, a menos que uma medida corretiva seja empregada. A causa de um erro é uma falha. Desde que erro é uma propriedade do estado do sistema, ele pode ser observado e avaliado. Falha, porém, não é uma propriedade do estado do sistema, e não pode ser observada facilmente. Falha é definida como algo que tem o potencial de gerar erros [Jal94]. Embora uma falha tenha o potencial de gerar erros, ela pode não gerar qualquer erro durante o período de observação. Isto é, a presença de falhas não garante que um erro irá ocorrer. Falhas podem ser caracterizadas como transientes ou permanentes. Transientes são as de duração limitada, causadas pelo mal funcionamento do sistema ou devido a alguma interferência externa. Falhas permanentes são aquelas nas quais desde que o componente falhe, ele nunca trabalha de maneira correta novamente (ou por um longo período de tempo).

Um sistema é tolerante a falhas se ele esconde a presença de falhas usando redundância. Redundância num sistema pode ser de hardware, software ou tempo. Redundância de hardware compreende os componentes de hardware adicionados ao sistema para prover tolerância a falhas. Redundância de software inclui todos os programas e instruções incorporadas para tolerância a falhas. Uma técnica comum para tolerância a falha é executar alguma instrução (ou seqüência de instruções) várias vezes. Esta técnica requer redundância de tempo, isto é, tempo extra para realizar as tarefas para tolerância a falhas. Normalmente em sistemas distribuídos são utilizados os três tipos de redundância.

Pode-se identificar quatro fases em tolerância a falhas [Jal94]: *(i)* detecção de erro, *(ii)* confinamento e avaliação dos danos, *(iii)* recuperação de erro, e *(iv)* tratamento de falha e continuação do serviço do sistema. Detecção de erro é a fase na qual a presença de uma falha é deduzida detectando-se um erro no estado de algum subsistema. Qualquer dano causado pode ser identificado e delimitado na segunda fase. Recuperação de erros é a correção do estado do sistema, evitando que novos erros ocorram devido a propagação do erro detectado. Após a recuperação do erro, pode ser feito o tratamento da falha e a continuação do serviço oferecido pelo sistema. Nesta última fase, a falha ou o componente onde ela ocorreu pode ser identificado, e o sistema tolerante a falha deve funcionar sem que este componente seja usado, ou usado de maneira diferente, tal que a falha não cause defeitos novamente.

2.4.1 Tolerância a Falhas de Software

No desenvolvimento de um projeto que inclui técnicas de tolerância a falhas pode-se dividir as falhas em [Bed97]: falhas de hardware e falhas de software. As falhas de hardware são aquelas causadas por componentes físicos no computador, e podem ser tratadas através de replicação de componentes, ou seja, redundância de componentes físicos críticos. As

falhas de software são aquelas causadas por erros lógicos deixados no software durante o seu desenvolvimento, e são difíceis de serem detectadas e tratadas, uma vez que exigem redundância de projeto e não simplesmente replicação de programas. Redundância de projeto, ou diversidade de projeto, é uma técnica que propõe implementações diferentes de componentes. Cada uma dessas implementações é chamada de variante, e são desenvolvidas com base numa mesma especificação. Dessa forma, garante-se a probabilidade de minimizar a ocorrência de erros iguais. Os resultados das variantes são submetidos a um seletor (um algoritmo de decisão) que seleciona o resultado com maior probabilidade de estar correto.

Técnicas básicas para tolerância a falhas em software são baseadas em diversidade de projeto, como por exemplo, *Blocos de Recuperação* e *N-Versões* [Bed97]. Na técnica de blocos de recuperação [Bed97], as variantes são nomeadas de alternantes e o seletor é chamado de teste de aceitação. O teste de aceitação é aplicado seqüencialmente aos resultados obtidos pelas variantes: se a primeira variante falha ao passar pelo teste de aceitação, o estado do sistema é restaurado e a segunda variante é executada; este processo continua até que todas as variantes se esgotem ou algum resultado passe pelo teste.

A técnica de programação em N-versões [Bed97] consiste na utilização de N componentes de software, com $N > 1$, independentemente projetados a partir de uma especificação comum, com o intuito de tolerar falhas de projeto. Cada componente é projetado independentemente, e se possível por grupos que não interagem entre si, fazendo uso de algoritmos distintos. Um mecanismo de votação determina um único resultado a partir dos resultados obtidos pelas N-versões, que podem ser executadas em paralelo ou seqüencialmente.

2.4.2 Tolerância a Falhas em Sistemas Concorrentes

As técnicas de *Bloco de Recuperação* e *N-Versões* são utilizadas para implementar tolerância a falhas em software para sistemas monoprocessados. Um Sistema concorrente consiste de múltiplos processos autônomos que se comunicam através de troca de mensagens. Em tais sistemas, erros gerados por um processo podem ser propagados para outros. Portanto, falhas podem se manifestar em diferentes locais. *Ações atômicas coordenadas*[3] é uma das principais técnicas para tolerância a falhas em sistemas concorrentes. Nessa técnica, uma transação atômica é usada para garantir ao sistema as seguintes propriedades:

- seriação: é possível mostrar que uma execução concorrente é equivalente a alguma ordem de execução serial
- atomicidade: uma computação termina normalmente ou é abortada

[3]Do inglês *atomic action*

- permanência de efeito: qualquer mudança de estado pode ser guardada em armazenamentos que sobrevivam às falhas de nós.

Com essas propriedades é possível restaurar o estado do sistema para um estado consistente, quando ocorre uma falha.

2.5 Reflexão Computacional

Refletir segundo o Aurélio [dHF93], significa *fazer retroceder, desviando da direção original; deixar ver, revelar; meditar, reflexionar.* Pode-se refletir não apenas sobre o mundo a volta, mas também sobre as próprias idéias, ações e experiências passadas. Essa habilidade está relacionada à flexibilidade de lidar com o mundo, adquirir novas capacidades; de reagir a circunstâncias inesperadas. A reflexão sobre as próprias ações, como um meio de adquirir a capacidade de lidar com situações novas, já era conhecida dos gregos na antiguidade, cujo templo de Delfos continha a famosa inscrição "Conhece-te a ti mesmo" [Gaa91].

Por analogia, em computação, um sistema reflexivo é um sistema computacional capaz de refletir sobre o seu próprio comportamento [Mae87, Smi85]. Smith [Smi85] fazendo esta analogia enunciou sua "hipótese de reflexão": *assim como sistemas computacionais podem ser construídos para "raciocinar" sobre um mundo externo através da manipulação de representações daquele mundo, também podem ser construídos para "raciocinar" sobre si mesmos manipulando representações de suas próprias operações e estruturas.* Percebe-se então que um sistema computacional é reflexivo se ele "conhece" a si próprio. Além disso há uma relação de causalidade entre o domínio do sistema e as estruturas que representam seu comportamento, pois uma alteração em um deles causa um efeito correspondente no outro. As estruturas que representam o comportamento do sistema são chamadas de auto-representação [Mae87]. Elas permitem que o sistema tenha uma representação precisa de si mesmo, além de estar sempre de acordo com o estado e a computação do sistema. Através de alterações na auto-representação é possível alterar o comportamento do sistema.

Quando utiliza-se reflexão computacional como uma técnica de construção de componentes de software, têm-se como resultado caixas-brancas, já que o comportamento dos componentes pode ser alterado. Portanto, sistemas reflexivos são mais flexíveis que sistemas não reflexivos.

2.5.1 Uso de Reflexão Computacional

O objetivo primordial de reflexão não é auxiliar nas atividades relacionadas com o domínio externo do sistema, mas sim contribuir na organização interna do sistema e na sua interface com o mundo externo [Mae87]. Desta forma, reflexão torna-se útil para a

estruturação das atividades administrativas do sistema — tais como estatísticas de performance, depuração de erros, otimizações e políticas de segurança — porque podem ser mudadas de acordo com novas necessidades, sem no entanto interferir na computação do mundo externo. Quando usada desta forma, reflexão computacional é uma técnica bastante útil para a construção de sistemas, por torná-los mais flexíveis. Reflexão é então uma técnica de abstração, permite separar os mecanismos que controlam as atividades internas dos mecanismo que controlam atividades externas do sistema, mantendo a transparência das atividades internas, porém tornando-as passíveis de alteração.

2.5.2 Arquiteturas Reflexivas

Numa arquitetura reflexiva, o sistema é visto como sendo constituído de duas partes: a parte reflexiva e a parte externa. A parte externa é responsável pela computação referente ao domínio externo, enquanto a parte reflexiva refere-se a computação das informações sobre o próprio sistema.

Embora reflexão possa ser empregada em diversos tipos de sistema, tem sido utilizada mais freqüentemente em linguagens de programação. Uma linguagem de programação tem uma arquitetura reflexiva se ela reconhece reflexão como um conceito fundamental de programação e provê meios de manipular computação reflexiva explicitamente [Mae87]. Reflexão já foi introduzida em linguagens de vários paradigmas. Algumas linguagens lógicas adotam o conceito de meta-teoria, ou seja, uma teoria sobre outra teoria. Linguagens funcionais reflexivas introduzem o conceito de funções reflexivas, que são funções que executam ao nível do interpretador manipulando dados representando o código a ser executado. Smalltalk-80 [Gol83] introduziu o conceito de metaclasse, que é uma classe que descreve outra classe.

Uma característica comum às linguagens reflexivas é que geralmente a representação da interpretação da linguagem é também usada para executar a linguagem. Linguagens assim construídas são chamadas de metacirculares. Metacircularidade torna a interpretação da linguagem constituída de níveis, sendo o número de níveis virtualmente infinito, onde o nível superior interpreta o nível imediatamente inferior. Geralmente as linguagens reflexivas interrompem a metacircularidade, criando um nível cujo metanível seja ele próprio. A razão pela qual as linguagens reflexivas utilizam metacircularidade é que isto facilita a implementação da relação de causalidade necessária em sistemas reflexivos. Então a auto-representação é exatamente o processo de interpretação metacircular que está executando no sistema. A utilização de metacircularidade para a implementação de reflexão exige que a linguagem trate de maneira uniforme dados e construções da própria linguagem, ou seja, programas também devem ser vistos como estruturas da linguagem [Mae87]. A consistência entre a auto-representação e o próprio sistema é automaticamente garantida porque a auto-representação é realmente usada para implementar o sistema.

2.5.3 Classificação dos Modelos de Reflexão

Segundo Ferber [Fer89], pode-se classificar os modelos de reflexão das linguagens orientadas a objetos em:

- modelo de *metaclasse*: baseado na equivalência entre meta-objeto e a classe de um objeto. Logo, um meta-objeto é compartilhado por todos os objetos que são instâncias da mesma classe.
- modelo de *meta-objeto*: onde há uma relação de 1-1 entre objetos e meta-objetos. Meta-objetos são instâncias de uma classe pré-definida, por exemplo META-OBJECT, ou instâncias de uma subclasse da classe pré-definida.
- modelo de *metacomunicação*: baseado na materialização[4] de mensagens.

O modelo de *metaclasse* é o menos flexível, já que todos objetos de uma mesma classe compartilham o mesmo meta-objeto. Esse modelo é útil quando o comportamento do meta-objeto é o mesmo para todos os objetos.

O modelo de *meta-objeto* permite que o meta-objeto guarde informações específicas sobre o objeto. Embora todos os meta-objetos tenham o mesmo comportamento, as informações contidas neles variam para cada objeto.

O modelo de *metacomunicação* pressupõe a existência de uma classe pré-definida no sistema, por exemplo MESSAGE, tal que toda invocação de método torna-se uma instância desta classe. A classe MESSAGE possui todas as informações necessárias para interpretar mensagens. Então, quando um objeto invoca um método, é criada uma instância de MESSAGE. Logo mensagens são também objetos. Este modelo é o mais flexível. Em contra-partida tende a ser mais ineficiente pois o sistema passa a maior parte do tempo executando no metanível.

2.6 Resumo

Orientação a objetos tem se revelado uma técnica bastante útil para o desenvolvimento de sistemas, devido a facilidade de abstração provida pelo conceito de objeto. Ao seu lado, tolerância a falhas aumenta a confiabilidade dos sistemas, permitindo tratar falhas que ocorram durante o funcionamento do sistema. Orientação a objetos é útil para a modelagem e estruturação dos componentes do sistema enquanto tolerância a falhas permite a operação do sistema quando os componentes falham. Reflexão computacional apresenta-se como uma técnica para auxiliar na organização interna do sistema, permitindo a separação entre atividades internas e externas. Neste capítulo foi apresentado os

[4]Do inglês *reification*

principais conceitos dessas três técnicas, seu uso e sua importância no desenvolvimento de sistemas.

Capítulo 3

Trabalhos Relacionados

3.1 Introdução

Este capítulo apresenta os principais trabalhos relacionados com o tema desta dissertação, divididos em linguagens de programação, sistemas operacionais e bancos de dados para fins didáticos. Nos tópicos Linguagens de Programação e Sistemas Operacionais enfatiza-se os aspectos de reflexão computacional enquanto em bancos de dados considera-se questões relacionadas com chamadas de métodos em consultas. A versão inicial de Stabilis não tratava invocações de métodos em consultas. Para implementar uma versão reflexiva de Stabilis é necessário alterar sua linguagem de consulta, de forma que métodos possam ser chamados dentro de consultas. Essa a razão para colocar-se também bancos de dados — no que se refere principalmente a chamada de métodos — como trabalhos relacionados.

3.2 Linguagens de Programação

Nesta seção são apresentadas diversas linguagens de programação reflexivas, mostrando os detalhes principais de implementação dos seus respectivos modelos de reflexão, e em seguida faz-se uma discussão desses modelos.

3.2.1 Open C++ 1.2

Open C++ 1.2 [CM93, Chi93] é uma extensão de C++ que permite o uso de reflexão. A linguagem pode refletir no acesso a atributos e em chamada de métodos. A linguagem adota o modelo de meta-objetos, uma vez que cada objeto tem um meta-objeto associado que controla o acesso a variáveis e chamadas de métodos. Em Open C++ existem dois tipos de objetos quanto à reflexão: reflexivos e não reflexivos. Um objeto reflexivo é

controlado por seu meta-objeto, que geralmente altera sua implementação. Um objeto não reflexivo é um objeto C++ normal. Métodos e atributos reflexivos são declarados através de pragmas no nível base [CM93], como mostra a Figura 3.1

```
class X {
    ...
  public:
    //MOP reflect:
        a();
        b();
   };

//MOP reflect class X : Y

class Y : public MetaObj {
    ...
  public:
    ...
    void Meta_MethodCall(Id m_id,ArgPac& args, ArgPac& reply)
     {
        ...
        Meta_HandleMethodCall(m_id,args,reply);
     }
   };

main()
  {
     X x1;                                          // objeto normal
     ref_X x2;                                      // objeto reflexivo
  }
```

Figura 3.1: Exemplo de um programa em Open C++ 1.2

Os métodos *a()* e *b()* da classe X são definidos como reflexivos pela diretiva *MOP reflect:*. A classe X está associada a sua metaclasse Y através da diretiva *MOP reflect class*. Em Open C++, toda metaclasse é uma classe C++ que herda e redefine alguns métodos de uma classe pré-definida: a classe MetaObj. Entre estes métodos destacam-se: *Meta_MethodCall*, sobrecarregado pela metaclasse para estender o método reflexivo e *Meta_HandleMethodCall*, herdado para executar o método do nível base definido pelo programador. Antes da compilação, Open C++, encontrando uma classe com métodos reflexivos, como X da Figura 3.1, cria uma subclasse com prefixo refl_, indicando que será

a classe dos objetos reflexivos — refl_X para o exemplo mostrado na Figura 3.1. Objetos C++ normais são criados como instâncias de X, e objetos reflexivos são criados como instâncias de refl_X.

Arquitetura do Metanível

O protocolo de meta-objetos de Open C++ baseia-se na interceptação de mensagens. Quando um método reflexivo é chamado, sua execução é desviada para o metanível, que pode então manipulá-la. Os métodos *Meta_MethodCall* e *Meta_HandleMethodCall* são fundamentais neste processo.

A Figura 3.2 ilustra o funcionamento do protocolo de meta-objetos de Open C++. Quando um método reflexivo é invocado, essa chamada é interceptada pelo meta-objeto — instância da metaclasse Y(Figura 3.2) — que passa a controlar sua execução. O método *Meta_MethodCall* é então invocado. Em seu código, o programador deve incluir as extensões desejadas além de invocar *Meta_HandleMethodCall*, que executa o método do nível base. Após a execução de *Meta_MethodCall*, o resultado é retornado para o usuário. Isto implica que, quando um método reflexivo é chamado, o meta-objeto precisa distingüir: qual método está sendo invocado, seus argumentos e onde armazenar seu resultado. Durante a invocação do método reflexivo, essas informações estão disponíveis em objetos da classe ArgPac. Essa classe comporta-se como uma pilha e implementa operações *push* e *pop* para cada tipo básico de C++ (int, char*, etc.).

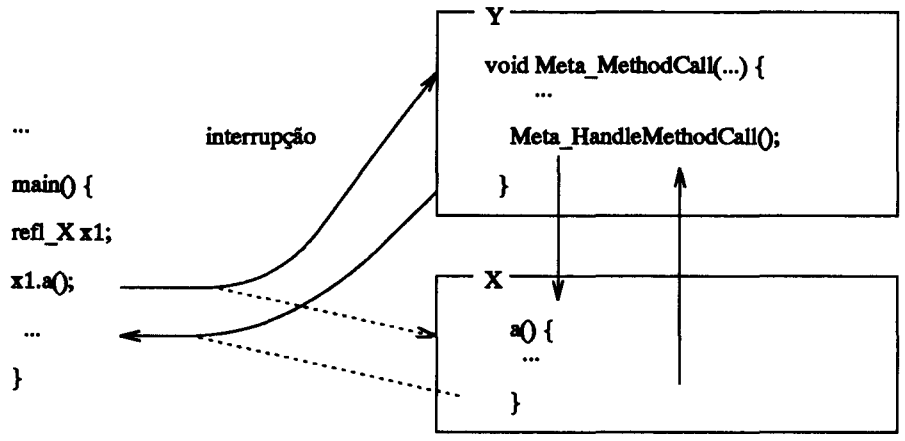

Figura 3.2: Funcionamento do protocolo de meta-objetos de Open C++1.2

Uma hierarquia em Open C++ para reflexão envolve basicamente quatro classes, como mostrado na Figura 3.3.

- X, criada pelo programador no nível base

- refl_X, subclasse de X gerada por Open C++. Os objetos reflexivos são instâncias de refl_X
- Y, metaclasse de X e cujas instâncias são meta-objetos.
- MetaObj, classe raiz da hierarquia de metaclasses;

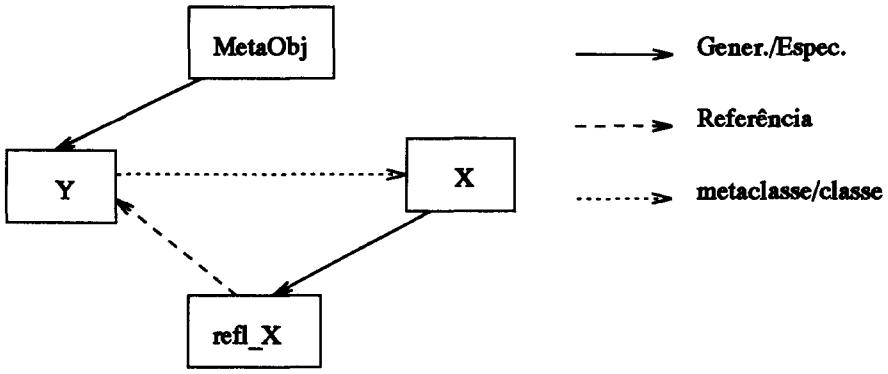

Figura 3.3: Hierarquia de metaclasses em Open C++1.2

Em Open C++ 1.2, o protocolo de meta-objetos foi incorporado através de um pré-processador que incorpora à implementação do programador um mecanismo para interceptação de mensagens. Uma metaclasse é na verdade uma classe C++ ordinária, havendo apenas uma referência da classe gerada automaticamente por C++ (refl_X na Figura 3.3) para metaclasse (Y na Figura 3.3). Não existe uma relação de instanciação entre classe e metaclasse. A metaclasse não tem acesso às variáveis definidas na classe, o que limita a aplicação. Para solucionar este problema, tem-se que romper o encapsulamento, fazendo com que atributos definidos na classe sejam passados como parâmetro para os métodos da própria classe, a fim de que sejam manipulados na metaclasse.

3.2.2 Open C++ 2.0

Como a versão 1.2, Open C++ 2.0 [Chi96] é também baseada em C++, e portanto uma linguagem baseada em classes. As estruturas da linguagem são as mesmas de C++, exceto as acrescentadas pelo programador usando programação no metanível.

Arquitetura do metanível

O compilador Open C++ 2.0 consiste de três estágios [Chi96]: pré-processador, tradutor de fonte Open C++ para fonte C++, e compilador C++. O protocolo de meta-objetos de Open C++ é uma interface que controla o tradutor do segundo estágio. Ele permite especificar como um código Open C++ estendido é traduzido em código C++ regular. Para

ligar-se as classes do nível base com classes do metanível, Open C++ introduz uma nova sintaxe para declaração de metaclasse. A metaclasse default é Class, mas o programador pode mudá-la usando: *metaclass class-name: metaclass-name [(meta-arguments)];*. Isto declara a metaclasse para uma classe. *meta-arguments* é uma seqüência de identificadores, nomes de tipos, literais ou expressões C++. Open C++ permite extensões como: novos modificadores de tipo, especificadores de acesso, entre outros.

No metanível, os programas do nível base são representados por objetos de algumas classes pré-definidas (e/ou suas subclasses que o programador define). Esses objetos são chamados meta-objetos porque eles são meta-representações dos programas. Tradução de fonte Open C++ para C++ é implementada pela manipulação destes meta-objetos. Os meta-objetos mais importantes são:

- Ptree: representam uma árvore de "parse" do programa, implementada como uma lista ligada
- Class: representam definições de classe e controlam tradução fonte do programa.

Meta-objetos Ptree possuem métodos sobrecarregáveis que permitem ao programador obter o significado sintático do texto do programa, representado na árvore de "parse", e também modificá-lo. Assim, dado um meta-objeto Ptree, pode-se saber se o texto representa uma função, um comando *while*, uma classe, etc...

Os meta-objetos do tipo Class implementam o protocolo de meta-objetos. A classe de um meta-objeto é selecionada pela declaração `metaclass` no nível base. Desde que um meta-objeto Class é uma metarepresentação de uma classe, programadores podem acessar informações da definição da classe através do meta-objeto.

Open C++ 2.0 é uma linguagem metacircular, isto é, os programas do metanível são também programas Open C++. Logo, metaclasses também têm suas metaclasses. A metaclasse para Class e suas subclasses é Metaclass. Programadores não têm de declarar explicitamente a metaclasse para suas metaclasses porque elas herdam a metaclasse de Class. Desde que Metaclass é também uma subclasse de Class, sua metaclasse é a própria Metaclass. Com isto, Open C++ 2.0 interrompe a metacircularidade. Os relacionamentos entre Class e MetaClass estão representados na Figura 3.4

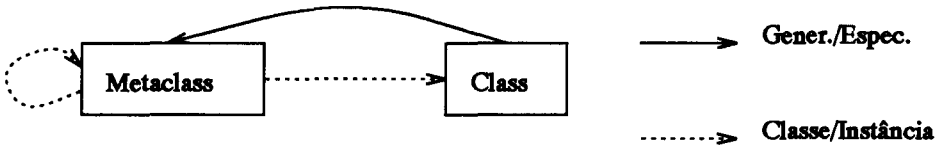

Figura 3.4: Relacionamento de Instanciação de Open C++2.0

Open C++ permite um alto grau de flexibilidade à aplicação, porém seu protocolo de meta-objetos é bastante complexo, portanto difícil de ser utilizado.

3.2.3 CLOS

CLOS (Common Lisp Object System) [DeM93, KdRB92] é uma extensão orientada a objetos de Common Lisp, baseada nos conceitos de classes, funções genéricas, métodos e herança múltipla. A linguagem adota o modelo de metaclasse, uma vez que todos objetos de uma mesma classe compartilham um único meta-objeto. A linguagem CLOS consiste de cinco blocos de construção básicos:

- classes
- "slots"
- funções genéricas
- métodos
- combinação de métodos

Classes contêm a descrição da estrutura e comportamento de instâncias. "Slots" são equivalentes a atributos em outras linguagens orientadas a objetos. Métodos são partes de código, associados a uma função genérica. Funções genéricas consistem de zero ou mais métodos que definem comportamentos diferentes para instâncias de classes diferentes. Quando uma função genérica é invocada, ela transfere o controle para o método apropriado, baseando-se na classe passada como parâmetro durante a chamada. Combinação de métodos permite ao programador estruturar e aumentar a execução de um método, chamado principal, através das primitivas: *:before*, *:after* e *around*. A primitiva *:before* especifica os métodos que devem ser executados antes do método principal. Analogamente, *:after* especifica os métodos executados após o método principal enquanto *:around* especifica os que executam antes e depois do principal.

Arquitetura do Metanível

Em CLOS, todos objetos são instâncias de classes com comportamento definido por métodos associados a suas classes. CLOS é metacircular: seu comportamento é caracterizado por um conjunto de classes, instâncias e métodos CLOS. A representação física de uma instância é controlada pela sua metaclasse, que determina as estruturas de armazenamento usadas. Então, programando-se no metanível é possível definir representações alternativas para as instâncias de uma classe, como por exemplo instâncias persistentes. Ao analisar CLOS é conveniente separar os aspectos estáticos dos dinâmicos. A parte estática de CLOS compõe a arquitetura do metanível e descreve os componentes do sistema e as estruturas dos blocos de construção da linguagem (classes, "slots", funções

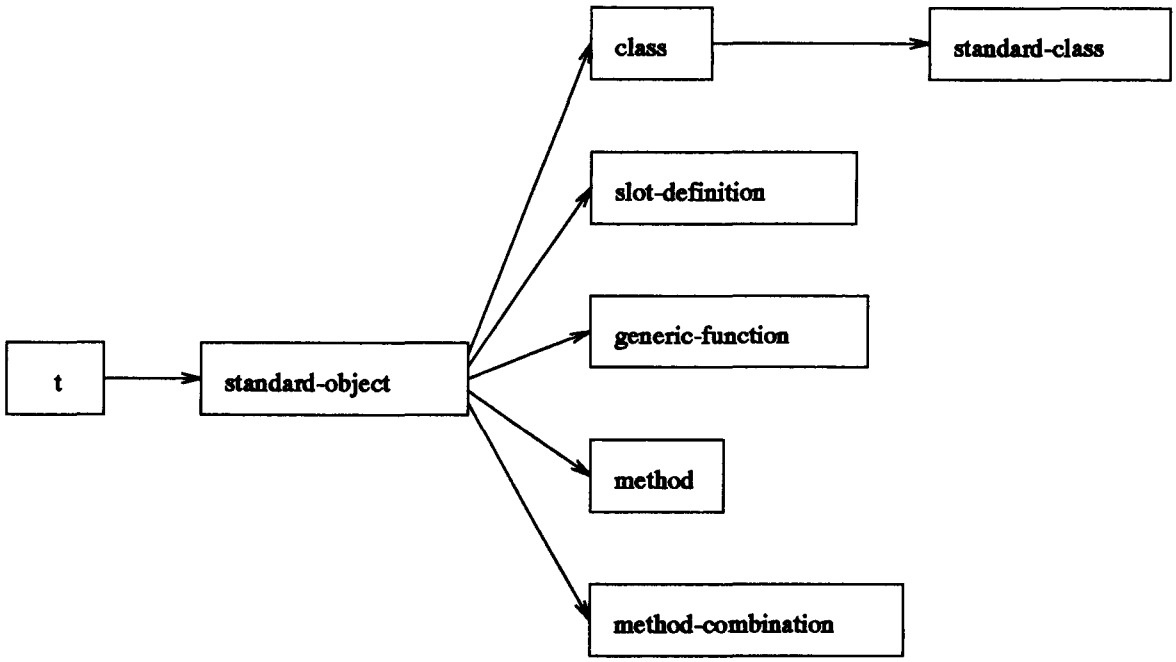

Figura 3.5: Hierarquia de classes em CLOS

genéricas, métodos e combinação de métodos). A hierarquia de classes que implementa tais elementos está apresentada na Figura 3.5.

Na Figura 3.5 nota-se que a classe **standard-object** é superclasse de todas as classes que representam os blocos de construção da linguagem, isto é, **standard-class**, **slot-definition**, **generic-function**, **method** e **method-combination**. Instâncias de **standard-class** são meta-objetos das classes definidas pelo programador. A macro *defclass* especifica as informações que definem uma classe e que podem ser obtidas posteriormente, fazendo-se acesso a esses meta-objetos. Instâncias da classe **slot-definition** representam "slots" e guardam informações como: nome, tipo e iniciação de "slots". A classe **generic-function** representa funções genéricas. Suas instâncias armazenam informações obtidas durante a definição de funções genéricas através da macro **defgeneric**. Analogamente, instâncias da classe **method** guardam as informações associadas a métodos obtidas durante a execução da macro **defmethod**. A classe **method-combination** representa combinações de métodos.

A parte dinâmica de CLOS é descrita em termos de protocolos que manipulam os blocos de construção obtendo um determinado comportamento para a linguagem. Os protocolos descrevem as principais atividades de funções genéricas e também como devem ser invocadas para implementar um padrão de comportamento da linguagem. Portanto, extensões e/ou modificações da parte dinâmica são implementadas definindo-se novos métodos em funções genéricas já existentes [Pae93]. Os principais protocolos de CLOS são:

1. Protocolos de iniciação de meta-objetos, onde classes, métodos e funções genéricas são iniciados usando a função pré-definida *make-instance*;

2. Protocolo de finalização de classe, que computa informações herdadas por superclasses;

3. Protocolo de instanciação de estruturas, responsável pela implementação do comportamento das funções que fazem acesso a "slots".

4. Protocolo de chamadas de funções genéricas. A cada função genérica está associado um discriminador. Sempre que uma função genérica é invocada o discriminador é invocado para determinar o seu comportamento. Ele deve procurar e executar os métodos apropriados, bem como retornar os resultados.

5. Protocolo para manter um dependente. Como classes e funções podem ser modificadas, é necessário que haja uma forma de alterar as informações referentes a esses meta-objetos. O protocolo de manutenção têm esse propósito, fornecendo uma maneira de registrar um objeto que deve ser notificado sempre que uma classe e/ou função genérica é modificada. Esse objeto é chamado de dependente.

Os conceitos de orientação a objetos incorporados em CLOS são compatíveis com o padrão Common Lisp já existentes. Isto forçou algumas decisões em torno do protocolo de meta-objetos, por exemplo distinção entre funções genéricas e funções ordinárias, além de métodos não serem encapsulados em classes. CLOS não é eficiente por ser um ambiente totalmente interpretado.

3.2.4 Smalltalk-80

Smalltalk-80 [Gol83] é uma linguagem orientada a objetos baseada em classe usando herança simples. O programa encontra-se distribuído em classes na hierarquia de classes do próprio Smalltalk-80, cuja raiz é a classe Object.

Smalltalk-80 não é apenas uma linguagem, mas também um ambiente de programação, provendo visões textuais entre o nível da linguagem e o nível do programador. Essas visões incluem ferramentas introspectivas e navegacionais tais como o inspetor de objetos e o "browser" de classes.

Arquitetura do Metanível

A implementação de Smalltalk-80 [Coi93] é organizada como um programa orientado a objetos. Os objetos representando os componentes ou blocos de construção primários

da linguagem são objetos do metanível, e portanto meta-objetos. Os meta-objetos em Smalltalk-80 são classes e métodos.

A arquitetura do metanível é construída a partir das classes que descrevem estes meta-objetos e suas hierarquias associadas. Smalltalk-80 trata classes como objetos que são instâncias de outras classes: suas metaclasses. Metaclasses são parcialmente escondidas do usuário, e acessíveis somente pelo envio de mensagens. Por exemplo, considere uma classe A. Smalltalk-80 denota sua metaclasse por A class. Metaclasses auxiliam na criação e iniciação de instâncias e de variáveis de classes.

Metaclasses também podem ser usadas para estender a estrutura de uma classe padrão. Podem ser modificadas adicionando novas variáveis de instância à definição "default" provida automaticamente pelo sistema. As extensões de uma classe padrão são limitadas devido a correspondência 1-1 entre classe e metaclasse. Parte da hierarquia de Smalltalk-80, compreendendo os meta-objetos, é mostrada na Figura 3.6.

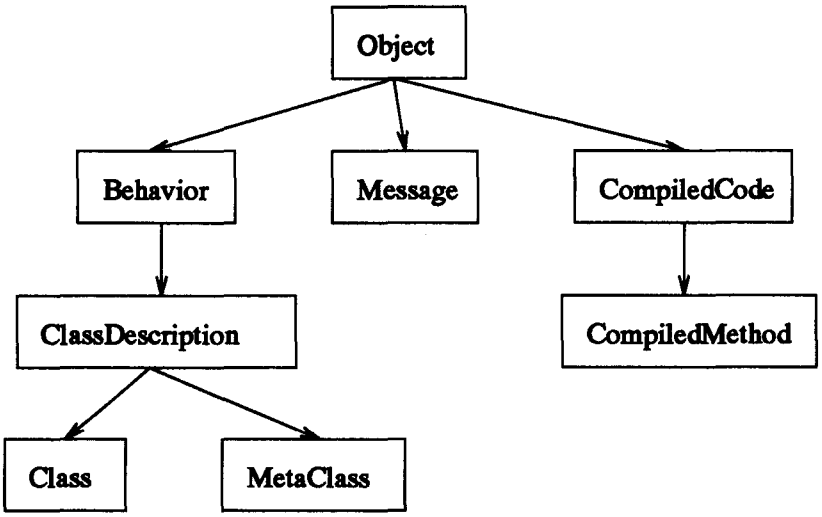

Figura 3.6: Hierarquia de classes de Smalltalk-80

Pode-se então reconhecer uma classe padrão (instância de Class) e sua metaclasse privada (instância de Metaclass).

Desde que metaclasse são automaticamente geradas por Smalltalk-80, a herança entre elas deve obedecer a uma regra padrão. A hierarquias de herança no metanível espelham as hierarquias de classes que elas implementam,isto é, se uma classe C_1 é subclasse de C_2, então a metaclasse de C será subclasse da metaclasse de C_2.

Para ser completo, o modelo de Smalltalk-80 especifica o estado de uma metaclasse como um objeto regular. Toda metaclasse é automaticamente definida como uma instância da classe Metaclass, sendo ela própria instância da metaclasse Metaclass class. Para fixar o número de metaníveis, evitando um número indefinido de metaclasses, uma circularidade

é introduzida definindo-se **Metaclass class** como uma instância de **Metaclass**, como mostra a Figura 3.7.

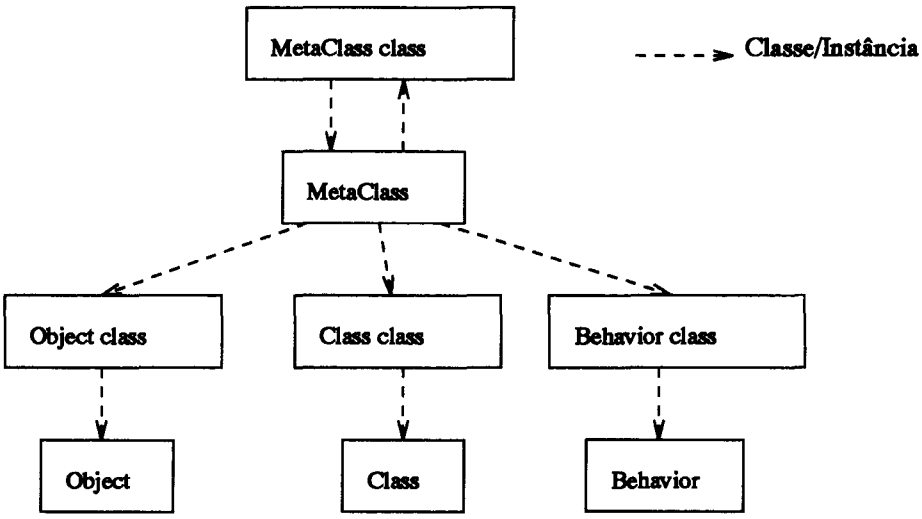

Figura 3.7: Relacionamento de Instanciação de Smalltalk-80

O estudo das classes mostradas nas figuras 3.6 e 3.7 revela que todos os métodos que criam objetos, classes e metaclasses, sempre fazem uso de dois métodos primitivos (*new* e *new:*), contidos em **Behavior**, sendo que o método definido em **Metaclass** sobrepõe o definido em **Behavior** para garantir a correspondência 1-1 entre uma classe e sua metaclasse privada. Smalltalk-80 portanto implementa um modelo de reflexão baseado em metaclasse, pois todos objetos estão associados a um único meta-objeto.

As limitações deste modelo vêm da generalidade do problema que ele tenta resolver (o estado de classes) e a solução provida (metaclasses escondidas parcialmente). O modelo é muito restritivo, pois o número de níveis de metaclasse é constante. Além disso, classes diferentes não podem compartilhar uma mesma metaclasse. Então, mecanismos comuns implementados no metanível não podem ser extraídos e implementados por uma metaclasse compartilhada.

3.2.5 RKL1

RKL1 [TT95] é uma linguagem de programação lógica paralela reflexiva baseada em KL1 [MNC+91]. KL1 foi projetada para representação do conhecimento sobre classes de objetos importantes do mundo real bem como os relacionamentos entre eles, afim de automatizar o entendimento de linguagem natural. Sua semântica formal pode ser descrita usando lógica de primeira ordem. Em KL1, pragmas especificam o nó destino sobre o qual o objetivo será executado e a prioridade de execução num nó.

Arquitetura do Metanível

Na arquitetura do metanível de RKL1 há dois conceitos importantes: metacorpo que é usado para descrever o código do metanível em RKL1, e Grupo de metaprocessos (MPG), que realiza o gerenciamento dos recursos entre os múltiplos objetivos [TT95]. A Figura 3.8 mostra um programa que define um meta-interpretador para KL1:

```
exec(true):- true | true.
exec((P,Q)):- true | exec(P), exec(Q).
exec(P):- not_sys(P) | reduce(P,Body), exec(Body).
exec(P):- sys(P) | P.
```

Figura 3.8: Um Meta-interpretador para KL-1

Este meta-interpretador é similar a um para Prolog. *sys(P)* e *not_sys(P)* são predicados pré-definidos que testam se P é um predicado "built-in" ou não. *reduce(P,Body)* procura por cláusulas cuja cabeça unifica com P, e seleciona uma que satisfaça sua guarda. Então instancia *Body* para a parte do corpo da cláusula selecionada. Modificando *reduce/2*, *sys/1*, *not_sys/1* e *exec/1* pode-se aumentar o meta-interpretador. Por razões de eficiência, só se pode modificar o corpo das cláusulas. Na definição do meta-interpretador anterior, a invocação do corpo do nível base é representada por *exec(Body)*. Substituindo *exec(Body)* por outro executor, computação reflexiva pode ser efetuada. Por exemplo, a cláusula seguinte corresponde à terceira cláusula do meta-interpretador que substitui *exec(Body)* por *distrib(Body)*.

```
exec(P):- not_sys(P) | reduce(P,Body), distrib(Body).

distrib(Body):- current_node(_,NN), random(NN,Node),
                    exec(Body)@node(Node).
```

Figura 3.9: Exemplo de programa em RKL1

Em RKL1, um executor definido pelo usuário para a parte do corpo (nível base) é descrito num metacorpo. Alguns predicados que abstraem uma execução da parte do corpo são usados no metacorpo. Por exemplo, *$execbody* é uma abstração para *exec(Body)*. Usando *$execbody* , o executor é descrito como se segue:

```
distrib:- current_node(_,NN), math:random(NN,Node),
          $execbody@node(Node).
```

Este metacorpo *distrib/0* distribui a execução de corpo do nível base para um nó determinado de forma aleatória. *current_node(_,NN)* é a invocação de um predicado "built-in" que instancia NN para o número de nós. *@node(Node)* é um pragma "built-in" de KL1 que especifica o nó para execução de um objetivo.

Em RKL1, pode-se especificar metacorpos diferentes para cada cláusula do nível base, como mostrado na Figura 3.10.

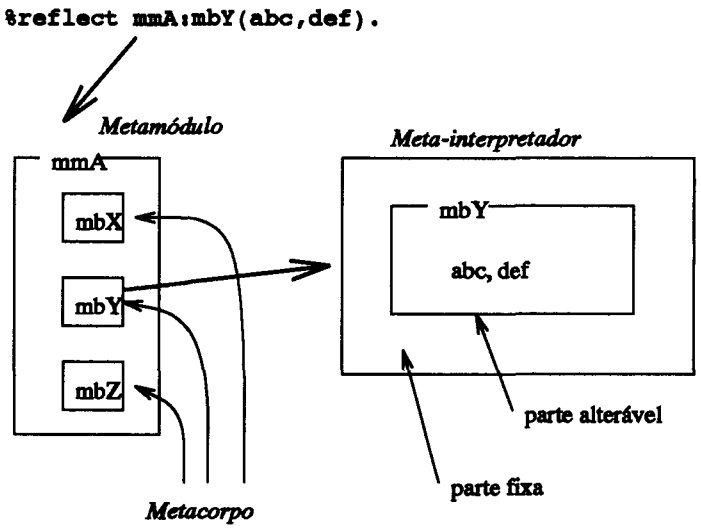

Figura 3.10: Arquitetura do metanível de RKL1

O programador insere a notação `%reflect` *MetaModulo: Metacorpo(Parametros...).* antes da cláusula que ele deseja mudar seu executor. *Metamodulo* encapsula metacorpos. *Parametros* são argumentos para a execução do metacorpo. A Figura 3.10 ilustra a relação entre o nível base e o metanível.

Quando um metacorpo é especificado numa cláusula do nível base, o corpo da cláusula será executado sob o interpretador (executor) adequado. O novo executor invoca os objetivos do metanível, e em seguida os objetivos do nível base ou outros objetivos ao invés dos originais.

O MPG é responsável pelo gerenciamento de recursos compartilhados entre os objetos. Consiste dos seguintes componentes:

- *Group-Controller*: gerencia recursos compartilhados. Sua descrição é feita num metamódulo. Ele deve ser invocado antes do uso do recurso.
- *Group-Stream*: é um canal de comunicação entre o *Group-Controller* e os metaprocessos.

- *Meta-Process.* Metaprocessos são execuções de metacorpos que enviam requisições do recurso para o *Group-Controller* através do *Group-Stream.*

Observando a arquitetura do metanível de RKL1, nota-se que o modelo implementa a cardinalidade 1(nível base) - N (metanível).

3.2.6 3-KRS

3-KRS [Mae87] é uma extensão reflexiva de KRS — uma linguagem usada para a representação do conhecimento. KRS é uma linguagem baseada em frames, segundo a classificação de Masini *et al.* [MNC+91].

Arquitetura do Metanível

3-KRS segue o modelo de meta-objetos. Cada objeto na linguagem tem seu próprio meta-objeto. Este, por sua vez, tem uma referência para o objeto a ele associado. Portanto não existe uma relação de instanciação entre eles. Todas entidades da linguagem (instâncias, métodos, variáveis de instância, mensagens e meta-objetos) são objetos.

Meta-objetos guardam informações sobre a implementação e interpretação de objetos. Assim, o interpretador está dividido em blocos que representam como certos tipos de objetos são implementados. A hierarquia de metaclasses da linguagem está ilustrada na Figura 3.11.

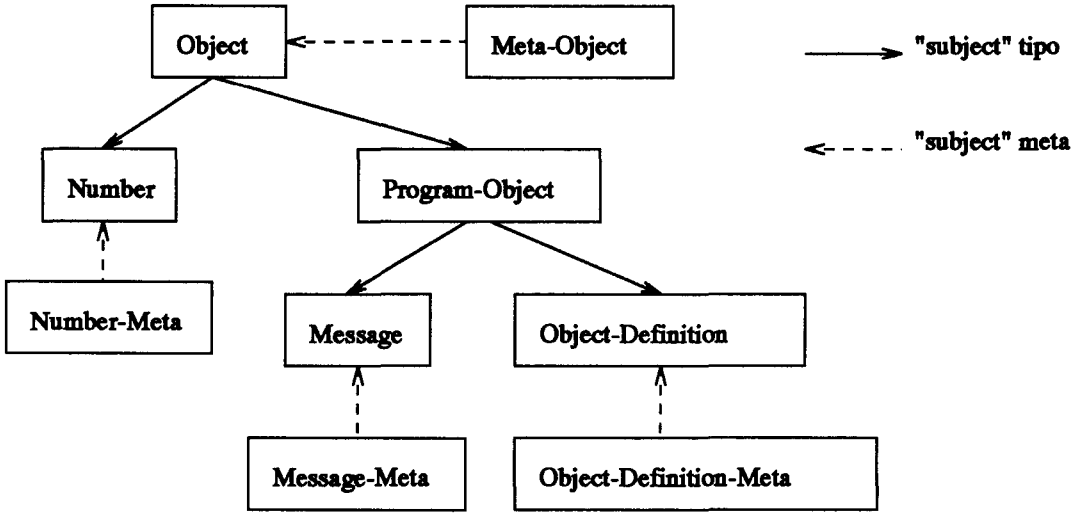

Figura 3.11: Hierarquia de classes e metaclasses de 3-KRS

Meta-Object é a raiz da hierarquia de metaclasses, ou seja, os demais meta-objetos da Figura 3.11 adicionam ou especializam seu comportamento. Por exemplo, Message-Meta

guarda informações sobre objetos que representam mensagens. Ela adiciona a **Meta-Object** variáveis de instância que representam o método a ser executado e o objeto que deve receber os resultados. 3-KRS fornece uma biblioteca de meta-objetos para imprimir e depurar programas. O programador precisa simplesmente associá-la aos objetos de sua aplicação.

3.2.7 Moostrap

Moostrap [MC93] é uma linguagem reflexiva cujo projeto foi motivado pelo estudo de reflexão em linguagens baseadas em agentes. Um agente é considerado como uma entidade inteligente que se comunica com outras entidades inteligentes para resolver algum problema [MNC+91]. Cada agente é também, por sua vez, uma comunidade de agentes elementares. Neste contexto, informação e comportamento estão distribuídos entre agentes que executarão a tarefa de forma independente e concorrentemente. Como classes, um agente encapsula dados locais (variáveis de instâncias) e um comportamento que define as ações que ele deve tomar de acordo com um determinado evento. Ao contrário de classes, este comportamento não é definido por métodos, mas por um único *script* que filtra as mensagens recebidas e ativa a parte apropriada do comportamento. Apenas um tipo de evento pode ocorrer: envio de mensagem. Uma mensagem contém um "continuation", ou seja, o agente a quem devem ser transmitidos os resultados obtidos pela execução da mensagem. O "continuation" pode ser o próprio agente que enviou a mensagem ou um novo agente criado especialmente para este propósito. No primeiro caso, o envio de mensagem tem a mesma semântica de linguagens baseadas em classe. No segundo caso, tem-se um tipo de comunicação assíncrona, e o agente que enviou a mensagem não precisa esperar pela resposta e pode, portanto, continuar executando suas atividades.

A idéia inicial do projeto era desenvolver o conceito de reflexão no contexto da linguagem Self [MC93]. Em Self, a estrutura de objetos pode ser consultada através de objetos chamados *mirrors*. Esses objetos são criados explicitamente para fornecer uma interface entre um protótipo e sua representação. Entretanto, Self não pode ser considerada reflexiva uma vez que *mirrors* não são objetos reflexivos reais, mas sim, um meio de ter acesso a informações sobre objetos através de primitivas da linguagem, mas não modificá-las. Assim, os projetistas envolvidos nesse trabalho optaram por desenvolver uma nova linguagem: Moostrap. Foi inspirada em Self, com um pequeno conjunto de primitivas. Protótipos (agentes) são compostos por "slots". Existem três tipos de "slots": "slots" de dados ou imutáveis (onde o valor do "slot" é computado em tempo de criação e não pode mais ser alterado); "slots" de atribuição ou mutáveis (onde o valor é usado para iniciar o "slot", podendo ser mudado posteriormente) e "slot" de função ou método (corresponde a métodos em outras linguagens orientadas a objetos).

Envio de mensagem tem a seguinte sintaxe: *(<receiver> <selector> <arg1>...<argn>)*,

onde *receiver* corresponde ao objeto onde o método será aplicado; *selector* corresponde ao nome do método e <*arg1*>...<*argn*> correspondem aos seus parâmetros. Para estender ou modificar protótipos existem duas primitivas: *-addSharedSlots* e *-removeSlots* que adicionam e removem "slots" a protótipos, respectivamente.

Arquitetura do Metanível

Moostrap implementa um modelo de reflexão baseado em meta-objetos. Cada objeto está associado a um meta-objeto que mantém as regras de acesso a seus "slots". Mensagens são materializadas em duas fases: *(i)*a busca do "slot" correspondente ao seletor (nome do método) e, *(ii)* em seguida sua aplicação. Toda mensagem *(receiver selector arg1 ... argn)* é divida em duas novas mensagens:

```
((meta-object(receiver) lookup receiver selector)
          apply receiver array-of(arg1...argn))
```

Todo meta-objeto implementa um método chamado *lookup* que faz a busca de "slots". Esse método tem como argumentos o objeto sobre o qual a mensagem será aplicada, e seu seletor (*lookup(rec sel)*). Meta-objetos são descritos por outros meta-objetos. Essa recursão é interrompida pela introdução de objetos que são seus próprios meta-objetos. Existe um meta-objeto básico, chamado *basic-meta-object*, ao qual todos objetos do sistema estão associados inicialmente. Esse meta-objeto define um método *lookup(rec sel)* que retorna o "slot" desejado ou gera um erro, caso não o encontre.

Fase de Busca do Seletor

O exemplo a seguir mostra como ocorre a fase de busca do seletor em um objeto `p`, controlado por um meta-objeto `bmo` onde deseja-se executar o método `x`.

```
((p -get-meta-object) lookup p ''x'')
      ((bmo -get-meta-object) lookup bmo ''lookup'')
      ((bmo -lookup ''lookup'')
          apply bmo selector)
```

A fase de busca do seletor é primeiramente delegada ao meta-objeto de `p`. A primitiva `-get-meta-object` retorna o meta-objeto associado a p (`bmo`). De acordo com a decomposição de mensagem, uma nova mensagem `lookup` é enviada ao meta-objeto de `bmo`, e assim recursivamente até se alcançar um meta-objeto circular, onde o "slot" para `lookup` é diretamente extraído usando-se a primitiva (`-lookup`) e então aplicado ao objeto.

Fase de Aplicação do Método

O resultado da fase de busca do seletor é um "slot" capaz de entender uma mensagem `apply`. Entretanto, cada "slot" está associado a um objeto chamado `slot-executant`. Esse objeto implementa um método `apply`, tendo como argumentos um array composto do objeto em que a mensagem será aplicada e seus parâmetros. Sempre que uma mensagem (`aSlot selector arg1...argn`) é executada, ela é redefinida da seguinte forma: (`slot-executant(aSlot) selector aSlot array-of(arg1...argn)`). Existem três tipos de `slot-executant`, um para cada tipo de "slot":

- `data-slot-executant`. A chamada de seu método `apply` retorna o conteúdo do "slot";
- `method-slot-executant`. A chamada de seu método `apply` executa o método armazenado no "slot";
- `assign-slot-executant`. A chamada de seu método `apply` atualiza o conteúdo do "slot".

Moostrap é uma linguagem flexível, não só por usar um mecanismo de delegação, mas também devido à arquitetura do seu metanível. A fase de aplicação para um "slot" pode ser modificada facilmente definindo-se um novo `slot-executant` (contendo um novo método `apply`) e associando-o com o "slot" correspondente. Como isto pode ser feito para os três tipos de "slots", a linguagem permite refletir no acesso a variáveis (leitura e escrita) e chamadas de métodos. Porém, uma desvantagem é que não há maneira de criar objetos não reflexivos. Logo, toda invocação de métodos será desviada para o metanível, onde a mensagem será decomposta nas fases de busca e aplicação, fazendo o sistema operar grande parte do tempo no metanível.

3.2.8 ABCL/R2

ABCL/R2 [MWY91] é descendente direta de ABCL/R, uma linguagem baseada em agentes, cujo propósito geral é concorrência e orientação a objetos. ABCL/R2 é baseada numa arquitetura de grupo híbrida [MWY91], que combina características de ABCL/R (baseada numa arquitetura individual) e ACT/R (baseada numa arquitetura "group-wide"). Permite a existência de grupos de objetos heterogêneos e grupos de recursos compartilhados, metagrupos e torres reflexivas de grupo/indivíduo, além de objetos não materializáveis (não reflexivos).

Arquitetura do metanível

ABCL/R2 coloca no metanível encapsulamento de controle de recursos compartilhados dentro de grupos de objetos, tais como comunicação e armazenamento. Os recursos são representados no metanível como objetos e são compartilhados entre os objetos do nível base num grupo. Ao mesmo tempo, cada objeto tem sua torre reflexiva, permitindo que meta-operações sejam adaptadas por objeto do nível base.

Os recursos são representados como grupo de objetos do núcleo no metanível. Grupos podem ser criados dinamicamente. Seu processo de criação é definido de forma metacircular. Como resultado, há uma torre de meta-objetos (torre individual) para cada objeto, e uma torre de meta-grupos (torre de grupo) para cada grupo. A torre individual determina a estrutura do objeto, incluindo seu "script" (métodos). A torre de grupo determina o comportamento do grupo, incluindo a computação (avaliação) do "script" dos membros do grupo, e é formada pelo grupo do núcleo. O grupo do núcleo é constituído pelos seguintes objetos [MWY91]:

- *Gerenciador de grupo*: representa e gerencia o grupo. A identidade de um grupo é dada pelo objeto gerenciador do grupo. Então, qualquer mensagem enviada para o grupo é realmente passada para o gerenciador de grupo.

- *Gerador de meta-objeto*: cria um meta-objeto no metanível. Um meta-objeto criado pelo gerador de meta-objeto padrão contém: um objeto fila de mensagens, um conjunto de scripts (métodos), um conjunto de variáveis de estado (variáveis de instância), referências para o avaliador e para o gerenciador de grupo, e o modo de execução. A recepção de uma mensagem `M` por um objeto do nível base corresponde a seu meta-objeto recebendo uma mensagem `[:message M R S]`. O meta-objeto então coloca a mensagem `M` no objeto fila de mensagens que ele possui. O meta-objeto também procura por um "script" que unifique com a mensagem no topo da fila, e se o encontra, envia uma mensagem para o avaliador executar o "script".

- *Avaliador*: é o recurso computacional compartilhado pelos membros (nível base) de um grupo. Seu papel é avaliar os "scripts" dos objetos membros. Seu comportamento é tipicamente como se segue: ele recebe uma mensagem para avaliação de uma expressão `[:do Exp Env Id Gid Eval]`, com destino de "reply" designado para um objeto de continuação C. Na recepção da mensagem, ele avalia a expressão `Exp` sob o ambiente `Env` e então envia o resultado para C. Os argumentos `Id` e `Gid` são mapeados para referências das pseudovariáveis `Me` e `Group`, que denotam o próprio objeto e seu grupo, respectivamente.

A torre individual é criada por uma técnica chamada progressão dinâmica de reflexividade. Para manter a relação de causalidade, apenas um objeto pode estar ativo na torre

em um dado instante. Quando o avaliador executa uma instrução de envio de mensagem de um meta-objeto, ele tenta enviá-la para o meta-objeto do objeto destino no mesmo nível. Se os níveis dos meta-objetos são diferentes, a mensagem é redirecionada pela torre. Como este redirecionamento é uma forma simples de delegação, tais objetos podem ser implementados usando-se objetos leves, diminuindo assim o "overhead" de comunicação.

A torre de grupo é criada através de uma técnica chamada criação preguiçosa [1]. Todo objeto, exceto o gerenciador de grupo, é criado com uma referência para o gerenciador de grupo do qual ele é membro. O gerenciador de grupo é criado inicialmente sem uma referência para seu gerenciador de grupo (o gerenciador de seu metagrupo). Esse é criado depois, quando for referenciado. Durante a criação do gerenciador de grupo, os outros objetos do grupo do núcleo não estão criados. Isto só ocorre quando eles são referenciados via gerenciador de grupo. Ao mesmo tempo, o gerenciador de grupo referencia seu próprio "group ID" causando a criação do metagrupo.

Em ABCL/R2, programadores podem criar objetos que executam mais eficientemente comparados a um objeto padrão com a perda das capacidades reflexivas baseada em indivíduo. Isto é indicado na criação do objeto.

3.2.9 Discussão

As diferenças entre as arquiteturas de metanível das linguagens aqui descritas têm origem na finalidade do projeto das linguagens. Smalltalk-80 foi projetada com a finalidade de resolver o problema de armazenamento de informações e estado de classes. Para solucionar o problema, seus projetistas optaram por uma arquitetura limitada onde metaclasses são parcialmente escondidas do usuário. CLOS, porém, foi projetada para ser uma linguagem aberta, e portanto o seu protocolo de meta-objetos já estava previsto no projeto da linguagem. Isto fez com que a própria linguagem fosse criada a partir da arquitetura proposta para o protocolo de meta-objetos, de maneira recursiva. Em Open C++1.2, o protocolo de meta-objetos foi incorporado através de um pré-processador. Logo, não existe uma integração forte entre a linguagem e o protocolo.

ABCL/R2 e RKL1 foram projetadas para aplicações distribuídas, sendo o gerenciamento de recursos do sistema implementados no metanível. Para tais aplicações, tanto o modelo de meta-objetos quanto o de metaclasse são restritivos. No modelo de meta-objetos, cada objeto controla apenas a computação do seu objeto correspondente. Portanto, a coordenação implícita de grupos torna-se difícil. Por exemplo, em RKL1 é difícil compartilhar informações sobre os recursos compartilhados porque cada metacorpo é executado individualmente para obter eficiência. A limitação do modelo de metaclasse é que a metaclasse perde a identidade de objetos no metanível. ABCL/R2 adotou um modelo

[1] Do inglês lazy

híbrido onde um objeto pode ser controlado por um meta-objeto e um grupo de objetos é controlado por grupos de meta-objetos.

Linguagem	1	2	3	4	5	6	7
Open C++ 1.2	classe	meta-objeto	1-1		√		
Open C++ 2.0	classe	metaclasse	N-1			√	
CLOS	classe	metaclasse	N-1				
Smalltalk-80	classe	metaclasse	N-1			√	
Moostrap	agente	meta-objeto	1-1				
ABCL/R2	agente	meta-objeto	1-1	√	√		√
RKL1	"frame"	metaclasse	1-N		√		√
3-KRS	"frame"	meta-objeto	1-1	√			

Tabela 3.1: Linguagens Reflexivas

1. classificação da linguagem
2. modelo de reflexão
3. correspondência meta-objeto/objeto
4. criação de meta-objetos sob demanda
5. objetos não materializáveis
6. hierarquia de metanível espelha hierarquia de herança
7. permite implementação de distribuição

Outra diferença entre as linguagens aqui apresentadas é quanto ao número de metaníveis e a cardinalidade da associação objeto/meta-objeto. Smalltalk-80 possui apenas um nível, pois o programador não pode criar uma metaclasse diretamente, a qual teria uma metametaclasse. Porém, Open C++ 1.2 e 2.0 permitem vários metaníveis, justamente porque o programador cria uma metaclasse diretamente, podendo associar a ela uma metametaclasse. Quanto a cardinalidade da associação objeto/meta-objeto, CLOS e Smalltalk-80 adotam N-1 enquanto Open C++1.2 e 2.0, ABCL/R2 adotam 1-1.

Outra questão que pode-se levantar é se a hierarquia do metanível deve ou não espelhar a hierarquia do nível base. Para compreender melhor esta questão considere classes A, B, MetaA e MetaB, sendo que B é subclasse de A, MetaA é metaclasse de A e MetaB é

metaclasse de B. A questão é: MetaB deve ser subclasse MetaA? Em Smalltalk-80 isto ocorre porque as metaclasses são criadas automaticamente, de forma implícita, garantindo assim a herança entre as metaclasses. Já Open C++2.0 estabelece uma ordem para a seleção da metaclasse: primeiro é verificado a existência de declaração explícita através da sintaxe `metaclass`. Quando a declaração não aparece, a metaclasse para as superclasses é selecionada, sendo que se elas possuem metaclasses diferentes, o compilador acusará um erro.

Em Open C++1.2, ABCL/R2 e RKL1, os objetos podem não ser reflexivos, apesar da indicação de uma metaclasse em sua classe ou um metacorpo em um corpo. Entretanto, em Open C++2.0 os objetos cujas classes contenham declaração de metaclasses serão sempre reflexivos.

3.3 Sistemas Operacionais

Sistemas operacionais são inerentemente reflexivos, pois inspecionam os processos e a estrutura de dados mantida pelo kernel. Entretanto a utilização de reflexão em sistemas operacionais é limitada desde que existem mecanismos providos pelo kernel cujo comportamento é de difícil alteração. A seguir é apresentado um sistema operacional que possui capacidades reflexivas: Apertos.

Apertos

Apertos [Yok92] é um sistema operacional orientado a objetos desenvolvido por Sony Computer Science Laboratory, no Japão. Provê as características requeridas por um sistema operacional distribuído e serviços para um ambiente aberto de computação móvel.

O "framework" usado para estruturar o sistema operacional é baseado em reflexão, onde é feita a separação das hierarquias de objetos e meta-objetos. A camada de meta-objetos define a semântica de certos métodos dos objetos. A associação entre objetos e meta-objetos muda com o decorrer do tempo. Isto reflete no comportamento dos objetos. Migração é o mecanismo usado para mudar o grupo de meta-objetos associados a um objeto. Apertos define um meta-espaço como sendo um grupo de meta-objetos que define um conjunto de protocolos do sistema operacional. Podem existir interseções entre meta-espaços. Objetos ao migrarem de meta-espaço mudam suas propriedades, por exemplo, um objeto ao migrar para um meta-espaço que fornece persistência, torna-se persistente. A camada de meta-objetos é vista como uma máquina virtual que pode ser adaptada às necessidades do grupo de objetos que ela gerencia.

3.4 Bancos de Dados

Nesta seção, encontram-se apresentados, em linhas gerais, os modelos de alguns sistemas de banco de dados, principalmente quanto ao uso de métodos em consultas. Isto porque foram implementadas chamadas de métodos em consultas para permitir o uso de reflexão computacional em consultas.

3.4.1 OSCAR

Como em sistemas relacionais convencionais, OSCAR [GH93] tem um dicionário de dados descrevendo as classes, tipos, e hierarquia definidos no esquema. A interface de um método em seu modelo de banco de dados orientado a objetos é uma informação a nível de esquema enquanto a implementação de um método para uma classe específica C é o valor de um atributo do meta-objeto C. Desde que o usuário está apto a consultar não apenas objetos concretos, mas também tipos de objetos, os tipos de objetos devem ser objetos de outro nível (meta). O metanível deve ser definido no mesmo modelo de objetos como um esquema normal para utilizar a mesma linguagem de consulta. OSCAR, ao introduzir meta-informação, não faz separação de esquema e instâncias completamente, mas trata os esquemas como instâncias de outro nível. O metanível de OSCAR consiste de duas visões: *(i)* a visão do sistema consistindo do dicionário de dados; e *(ii)* a visão da aplicação, contendo os tipos padrões de objetos da aplicação, definidos pelo usuário.

A visão do sistema consiste das classes **Classes**, **Attributes**, **Meta_Objects** e **Methods**, dentre outras. Após a iniciação do sistema, a classe **Meta_Objects** contém todas metaclasses e atributos das metaclasses da visão do sistema. Se o usuário cria uma nova classe abstrata com alguns atributos e métodos, então é feita a criação de um objeto na classe **Meta_Objects** e sua inserção na classe **Classes**. Também é criado um objeto na classe **Meta_Objects** e feita sua inserção na classe **Attributes** e naquelas classes **X_Attributes** que correspondam ao construtor do tipo X.

OSCAR representa a implementação dos métodos através de um tipo abstrato de dados que encapsula uma árvore de "parse". Desde que métodos são codificados numa sintaxe similar a C++, incorporando operações de atualização e/ou consulta ao sistema OSCAR. Os nós da árvore de "parse" representam expressões C++ de atualização/consulta. Portanto, métodos em OSCAR são interpretados, funcionando como macros.

3.4.2 O_2

O modelo de dados de O_2 [LRV92] estabelece que as entidades de um banco de dados orientado a objetos são objetos. Cada objeto é identificado por um par *(id,valor)*, onde *id* é único e sempre referencia o objeto. Uma classe é caracterizada pelo tipo de suas instâncias

e por um conjunto de métodos. Valores, ao contrário de objetos não têm identidade e são sempre manipulados por operadores, ao passo que objetos são manipulados por métodos. Métodos em O_2 podem ser codificados em CLOS ou CO_2 e têm seus códigos binários armazenados no sistema. Quando um método precisa ser executado para resolver uma consulta, o código binário correspondente é carregado e executado.

3.4.3 POSTGRES

POSTGRES [LAR] é um sistema de gerenciamento de banco de dados estensível. Seu modelo de dados é baseado na idéia de estender o modelo relacional com mecanismos gerais que podem ser usados para simularem uma variedade de construções de modelagem de dados semânticos. Os mecanismos incluem tipos abstratos de dados, dados do tipo procedimento e regras. Esses mecanismos podem ser usados para implementarem uma hierarquia de objetos compartilhada para uma linguagem de programação orientada a objetos. Em POSTGRES, um banco de dados é composto por uma coleção de relações que contém tuplas que representam entidades do mundo real ou relacionamentos. Uma relação tem atributos de tipos fixos que representam propriedades das entidades e relacionamentos, e uma chave primária. Tipos dos atributos podem ser atômicos ou estruturados. O usuário pode definir procedimentos, escritos em uma linguagem de programação convencional, que serão usados para implementar operadores de tipos abstratos de dados. Um procedimento é definido para o sistema pela especificação dos nomes e tipos dos argumentos, o tipo de retorno, a linguagem em que ele está escrito e onde os códigos fonte e objeto estão armazenados. POSTGRES armazena a informação sobre um procedimento nos catálogos do sistema e dinamicamente carrega o código objeto quando ele é invocado dentro de uma consulta.

3.4.4 Discussão

Alguns modelos de bancos de dados orientados a objetos fazem a distinção entre valores e objetos [LRV92]. Para eles, valores não possuem um identificador único como objetos possuem. Essa distinção influi na semântica de execução de métodos em consultas. Quando o modelo considera que o resultado de uma consulta é sempre valores, então métodos retornam valores e não objetos. Quando o modelo considera o oposto, isto é, consultas sempre retornam objetos, depara-se com a seguinte questão: retorno de um método, cujo tipo é primário (inteiro, booleano, etc.) deve ser um objeto? Considerando que a resposta é sim, então o sistema deve criar automaticamente um objeto (persistente) para o retorno de métodos cujo resultado são, por exemplo, valores de atributos. Ou seja, valores de tipos primários também devem ser considerados objetos. Considerando que o

retorno de tipo primário para um método não é um objeto, então a linguagem de consulta perde a uniformidade, pois nem tudo que ela manipula é objeto.

3.5 Resumo

Reflexão computacional tem sido aplicada a vários domínios de aplicações, desde sistemas operacionais, sistemas distribuídos em geral, a linguagens de programação. Neste capítulo foram apresentadas algumas arquiteturas reflexivas de linguagens orientadas a objetos, mostrando suas diferenças quanto à organização do metanível, à forma de comunicação entre objetos e meta-objetos, e à relação entre as hierarquias de herança e metaclasses. O estudo comparativo permite concluir que ao projetar arquiteturas reflexivas para linguagens orientadas a objetos deve-se considerar principalmente os seguintes critérios: *(i)* cardinalidade da associação objeto/meta-objeto; *(ii)* objetos não materializáveis (não reflexivos) e *(iii)* relação entre as hierarquias de herança e metaclasses. Foi apresentada também a arquitetura do sistema operacional Apertos, que utiliza reflexão computacional para implementação de migração de objetos.

Alguns modelos de banco de dados orientados a objetos foram mostrados, enfatizando o uso de métodos em consultas. Discutiu-se também as implicações do uso de métodos em consultas para os modelos de banco de dados em geral.

Capítulo 4

Stabilis

4.1 Introdução

Stabilis [Buz94, Cal96] é uma ferramenta para construção de aplicações distribuídas orientadas a objetos tolerantes a falhas, implementado sobre Arjuna [SP91], que fornece mecanismos para distribuição e tolerância a falhas através de um modelo baseado em objetos e ações atômicas. Um objeto Arjuna pode ser uma entidade persistente, ou seja, continua a existir após a aplicação que o criou terminar sua execução. Como os objetos são manipulados apenas dentro de ações atômicas, garante-se sua integridade, e portanto do sistema na presença de falhas de hardware, tais como falha e parada de estações, e perdas de mensagens na rede. Nas seções seguintes é apresentada a descrição da arquitetura de Arjuna e Stabilis.

4.2 Arjuna

Arjuna [SP91] é um sistema de programação orientado a objetos que provê um conjunto de ferramentas para construção de aplicações distribuídas tolerantes a falhas utilizando o modelo de computação baseado em objetos e ações atômicas. Neste modelo, um programa consiste de objetos interagindo, onde cada interação acontece dentro de um ação atômica. Uma ação atômica é uma abstração de programação que garante seriação, atomicidade de falha e permanência de efeito de uma execução. Seriação garante que a execução de programas concorrentes está livre de interferências e portanto equivale a alguma execução serial. Atomicidade de falha garante que se uma computação termina, então produz os resultados desejados; ou se ela aborta, então não produz nenhum resultado. Se ocorre alguma falha, o uso de recuperação de erro desfaz o resultado produzido. Permanência de

efeito garante que qualquer mudança produzida é gravada em memória estável [1]. Arjuna suporta ações atômicas aninhadas para estruturação de aplicações. Objetos em Arjuna podem ser atômicos ou atômicos persistentes. Operações sobre os objetos são invocadas sobre o controle de ações atômicas. Permitindo que objetos sejam persistentes e manipulados somente dentro de uma ação atômica, garante-se a integridade dos objetos e portanto a integridade do sistema na presença de falhas. Arjuna provê para as aplicações: armazenamento de objeto (gerenciamento de persistência), ações atômicas aninhadas (gerenciamento de transações distribuídas), acesso a objetos remotos (invocação de método remoto através de RPC); todos de forma transparente. Esse é o modelo de objetos e ações atômicas de Arjuna.

Arjuna é implementado como uma biblioteca de classes C++ e o programador utiliza-o através do mecanismo de herança [Cal96]. Os principais módulos que compõem Arjuna são: *RPC*, *Name Server*, *Object Store*, *Atomic Action* e *Replication*. O módulo *RPC* é usado para invocar operações sobre objetos persistentes. O módulo de *Name Server* mantém informação de identificação e localização sobre objetos persistentes. *Object Store* encapsula a representação estável de objetos persistentes. O módulo *Atomic Action* é a interface ao nível de aplicação. Estes módulos estão descritos a seguir.

4.2.1 *RPC*

Arjuna [SP91, Buz94] adota o modelo cliente/servidor [Tan] para fazer acesso aos objetos persistentes. Um servidor gerencia o estado de um objeto, define e executa operações que são exportadas para os clientes. Clientes invocam essas operações para manipular o estado do objeto guardado pelo servidor. A invocação de uma operação sobre um objeto persistente é implementada como uma chamada de procedimento remoto (RPC) [Tan].

4.2.2 *Name Server*

Nomes são úteis para vários propósitos em sistemas distribuídos; um deles refere-se a objetos. Para nomear e encontrar objetos num sistema distribuído, as funções "naming" e "binding" são geralmente providas através de um servidor de nomes. A função "naming" mapeia um nome de objeto fornecido pelo usuário para um identificador único de objeto já atribuído ao objeto. Já a função "binding" mapeia o identificador único de um objeto à sua localização.

[1] Do inglês *stable storage*

4.2.3 *Object Store*

Este módulo provê um serviço de acesso ao estado passivo de objetos persistentes. A representação estável do estado passivo de um objeto persistente, geralmente armazenada em disco, tem que ser independente da máquina para permitir sua transmissão entre armazenamento estável e volátil, e também sua transmissão como uma mensagem. A classe **ObjectState** implementa tal representação, provendo operações para empacotamento e desempacotamento do estado de um objeto persistente para/de uma instância de **ObjectState**.

4.2.4 *Atomic Action*

Provê a interface de programação de Arjuna [SP91]. Os mecanismos necessários para controlar concorrência, recuperação e ações atômicas são implementados neste módulo. Para escrever uma aplicação que esteja de acordo com o modelo de computação de objetos e ações, o programador declara instâncias da classe **AtomicAction** em seu programa. As métodos providos por esta classe (*begin()*, *end()* e *abort()*) são usados para organizar as ações atômicas. Os objetos controlados por ações atômicas são instâncias de classes Arjuna ou definidas pelo usuário derivadas da classe **LockManager** de Arjuna.

4.2.5 *Replication*

Arjuna implementa replicação passiva e ativa. Replicação passiva é "default". Mecanismos de transparência de replicação escondem do usuário o número e lugar das cópias dos recursos ou serviços. Normalmente, usuários de um serviço replicado não devem ser informados que existem múltiplas cópias de um serviço. Para eles, os serviços replicados são identificados individualmente quando eles requisitam operações para serem executadas. Embora requisições de operações possam ser executadas concorrentemente por todas cópias do serviço replicado, transparência de replicação garante que os usuários do serviço somente recebem de volta um único conjunto de resultados. Na interface de programação de serviços replicados cada recurso replicado é representado por um "replicagroup". Em Arjuna, grupos são gerenciados através do servidor de nomes com o auxílio de um banco de dados especializado chamado *GroupView*. Ambos servidor de nomes e o banco de dados *GroupView* mantêm informação pertencente a objetos replicados. Para replicar um objeto, um programador tem que criar um grupo de réplica fornecendo como entrada a informação referente às réplicas em ambos subsistemas.

4.3 Arquitetura de Stabilis

Stabilis [Buz94, Cal96] é uma ferramenta para construção de aplicações distribuídas orientadas a objetos tolerantes a falhas. Aplicações típicas que podem ser construídas usando Stabilis são máquinas de busca de objetos. Uma máquina de busca de objetos[2] é um sistema de meta-informação cujo propósito é prover uma interface orientada a objeto para informação contida em ambientes distribuídos de grande escala. Assim, uma máquina de busca mantém objetos que contêm informação extraída dos recursos de rede, ou seja, objetos são instâncias de classes que modelam informações contidas em recursos de rede. Uma máquina de busca [Cal96] é composta pelos seguintes componentes básicos: classe (um tipo abstrato de dados que define um conjunto de atributos, um conjunto de relacionamentos com outras classes e um conjunto de métodos), objeto (uma instância de uma classe) e um esquema (um conjunto de classes que modela alguma informação). O objeto aqui é uma unidade atômica, e é uma unidade de concorrência, replicação e "caching". Um esquema designa indiretamente um conjunto de objetos que é o conjunto de todas as instâncias das classes pertencentes ao esquema. Classes e esquema podem ser considerados componentes conceituais, porém a máquina de busca de objetos deve mantê-los representados fisicamente pelas seguintes razões:

- Documentação. Informação sobre classes auxiliam os usuários a navegar através da informação. Além disso, esta informação é útil no desenvolvimento de ferramentas que usam a máquina de busca.
- Resolução de consulta. Informação sobre classes permite a um interpretador de consulta analisar expressões de consultas e resolvê-las.
- Gerenciamento de Software. Informação sobre classes permitem geração de código automática.
- Administração. Administradores de sistemas precisam gravar definições de classes e esquemas para atualização e/ou reutilização futura.

O serviço mais requisitado a máquinas de busca é resolução de consultas. Os componentes da máquina de busca usados para este propósito são índices [3]. Stabilis [Cal96] mantém dois índices:

- Índice de Atributo: faz a relação entre valores de atributos e referências de objetos. Um índice de atributo referencia a todos objetos que são instâncias da classe a qual o atributo pertence.

[2] Do inglês *object engine*

[3] Assim chamados para acompanhar a nomenclatura usada em sistemas de bancos de dados

- Índice de Relacionamento: faz a relação entre referências de objetos. Assim uma referência de objeto é mapeada para outra referência de objeto quando ambas correspondem a objetos conectados entre si.

Um programa Stabilis [Buz94] pode ser visto como sendo a comunicação entre duas partes distintas: C++ e o interpretador de Stabilis. Uma aplicação Stabilis é constituída de dois programas: o programa esquema e o programa consulta. O programa esquema especifica a relações estáticas do modelo de objetos da aplicação enquanto o programa consulta especifica criação e/ou recuperação de objetos persistentes, que são instâncias das classes da aplicação. Para descrever a arquitetura de Stabilis, deve-se definir o que são objetos, classes e relacionamentos para o mesmo, como é feita a construção de um programa esquema/consulta e mostrar a organização do espaço de objetos. A seguir são descritos todos esses aspectos da arquitetura de Stabilis.

4.3.1 Modelo de Objetos de Stabilis

Um objeto em Stabilis é definido por uma identidade, um estado e uma interface [Cal96]. A identidade é uma identificação única que permite o objeto ser referenciado de forma não ambígua. Ela é representada por um nome único. O estado é uma estrutura contendo as propriedades do objetos é representado por atributos e relacionamentos. A interface contém as operações que podem ser aplicadas ao objeto, representada pelos métodos.

Uma classe em Stabilis é um tipo abstrato de dados que define uma estrutura de estado de objetos e a interface correspondente. Portanto, a classe faz especificações de atributos, relacionamentos e métodos dos objetos. A notação gráfica usada para designar uma classe é um retângulo onde coloca-se o nome da classe, os nomes dos atributos e métodos. Os nomes dos atributos e métodos são opcionais. Instâncias são indicadas por meio de uma elipse, onde se coloca o estado do objeto (valor de seus atributos). A Figura 4.1 mostra a classe Lista que têm atributo inteiro *dado* e métodos *cons()*, *car()* e *cdr()*. Uma aplicação Stabilis deve especificar as classes e seus relacionamentos no programa esquema. O conjunto de classes e relacionamentos de uma aplicação chama-se modelo estrutural, ou seja, um modelo estrutural especifica as relações estáticas do programa distribuído a ser implementado.

Em Stabilis são possíveis os seguintes relacionamentos entre classes [Buz94]:

- Generalização/Especialização.

  Ocorre quando uma subclasse herda atributos e comportamento de outra classe: sua superclasse. Uma classe sem superclasses é uma classe raiz. Uma classe que não tem subclasses é uma classe folha. Ocorre herança múltipla quando uma classe tem

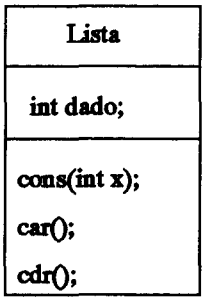

Figura 4.1: Classe Lista

mais de uma superclasse. A notação usada para este relacionamento é uma seta com início na superclasse e fim na subclasse. A Figura 4.2 mostra um relacionamento de generalização/especialização onde a classe Lista é subclasse de Vetor e superclasse de Pilha e Fila. O projetista da hierarquia de classes deve garantir que a especificação de métodos permaneça consistente. O requisito de consistência pode ser expresso em termos de invariantes, onde em cada método herdado por uma subclasse, as pré-condições sobre o método na subclasse podem ser mais fracas que as pré-condições sobre o mesmo método na classe base, e as pós-condições sobre o método na subclasse devem ser mais restritivas que as pós-condições na classe base. Além destes invariantes de métodos, também deve ser atendido o invariante de classe: a especificação de uma classe deve incluir, como um subconjunto, a especificação de cada uma das superclasses.

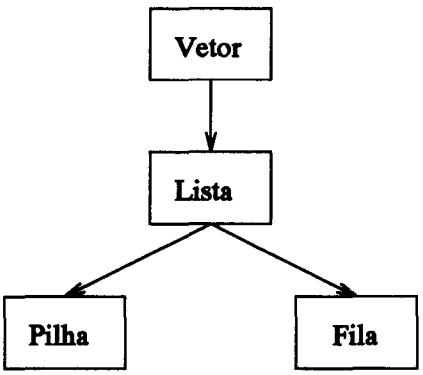

Figura 4.2: Generalização/Especialização

Atendendo a esses requisitos, ao construir uma hierarquia de classes, as subclasses serão subtipos compatíveis com suas superclasses e a substituição polimórfica de subclasses por superclasses será possível. Esse tipo de relacionamento de Generalização/Especialização é chamado *herança estrita.*

- Associação.

Uma associação é uma descrição de uma conexão conceitual entre duas classes. O propósito de uma classe numa associação é especificar o seu papel na associação. A multiplicidade de uma classe numa associação especifica como muitas instâncias dessa classe relacionam-se com uma instância da classe associada. A notação para associação é um traço ligando as duas classes. A Figura 4.3 mostra que há uma associação entre a classe **Empresa** e a classe **Empregado**. No exemplo uma empresa está associada a 0 ou mais empregados enquanto um empregado está associado a uma única empresa.

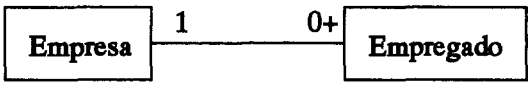

Figura 4.3: Associação

- Agregação.

  Uma agregação é uma associação onde uma classe tem um relacionamento "todo-parte" ou "parte-de" com outra classe. As classes no papel de "parte-de"são denominadas classes componentes e a classe no papel de "todo" é denominada classe agregada. Instâncias de uma classe componentes são objetos componentes, e instâncias de uma classe agregada são objetos agregados. A existência de um objeto componente pode depender da existência do objeto agregado do qual ele é parte. Neste caso a agregação é dita *forte*, e este relacionamento é denotado por uma seta tracejada partindo da classe agregada para a classe componente. Quando componentes e agregados têm existência independente, então a agregação é dita *fraca*; sua notação é uma seta pontilhada. No exemplo da Figura 4.4 tem-se: Caso (a) a classe **Maquina** é uma agregação fraca de **Peca**, isto porque as peças existem independentemente da máquina. No caso (b), **Class** é uma agregação forte de **Attribute**, pois a existência de uma instância de **Attribute** depende da existência de uma instância de **Class**.

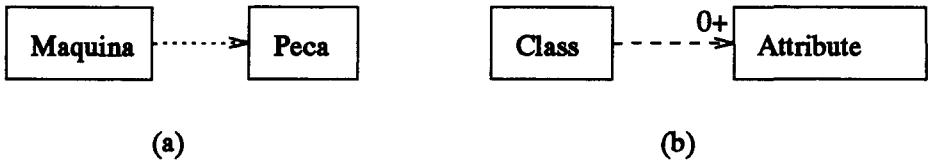

Figura 4.4: Agregação

- Instanciação.

  O relacionamento de instanciação relaciona uma classe a suas instâncias. A representação explícita deste relacionamento é útil quando classe e instância são ambas

manipuladas como objetos. A notação para instanciação é uma seta tracejada bidirecional ligando a classe e sua instância, conforme mostrado na Figura 4.5.

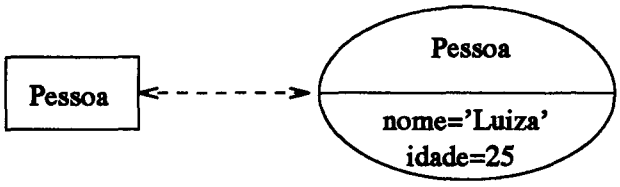

Figura 4.5: Instanciação

Pode-se fazer distinção entre instâncias diretas e indiretas de uma classe. Uma instância direta de uma classe C consiste dos atributos, relacionamentos e métodos definidos e herdados por C. Uma instância indireta de C consiste dos atributos, relacionamentos e métodos definidos e herdados por uma classe que, direta ou indiretamente herda de C. Define-se a extensão de C, denotada por *ext(C)*, como o conjunto de todas as instâncias diretas de C; e extensão fecho de C, denotada por *ext*(C)*, como a união da extensão de C e todas as extensões de todas as classes que herdam de C. Estas definições são importantes quando trata-se da aplicabilidade de métodos sobre objetos. Isto porque métodos definidos ou herdados por C são aplicáveis sobre *ext*(C)*.

O modelo estrutural de Stabilis, apresentado na Figura 4.6, explicita que uma classe é uma agregação de atributos, métodos e relacionamentos. A classe **Relationship** é uma classe abstrata que representa um relacionamento entre duas classes, que pode ser de Generalização/Especialização, agregação ou associação. A classe **Model** descreve o modelo de uma aplicação como sendo uma agregação fraca de submodelos e classes. Todo modelo possui uma única classe raiz — a raiz da hierarquia das classes que compõem o modelo. As classes de Stabilis, na verdade são metaclasses, pois descrevem as classes que compõem uma aplicação. Portanto, o modelo estrutural de Stabilis é um metamodelo que descreve modelos de aplicações. Esse metamodelo está especificado num programa esquema especial, em termos dele mesmo, logo é metacircular. A circularidade do modelo é quebrada pela classe **Class**, que define um construtor especial que instancia ela própria.

Todas as classes do modelo estrutural de Stabilis (Figura 4.6) são subclasses da classe **Object**, cuja interface é mostrada na Figura 4.7. **Object** é subclasse de **LockManager**, uma classe de Arjuna cujos objetos podem ser controlados por ações atômicas. O atributo *cache_uid* é o identificador único do objeto; *assign_expression* é uma estrutura que permite recuperar, criar e atualizar objetos através de consultas; *relationships* guarda os relacionamentos do objeto com outros objetos e *attributes* guarda os atributos do objeto.

O programador pode agrupar os objetos de maneira que as consultas possam ser resolvidas considerando apenas os objetos que pertencem ao grupo formado. O atributo

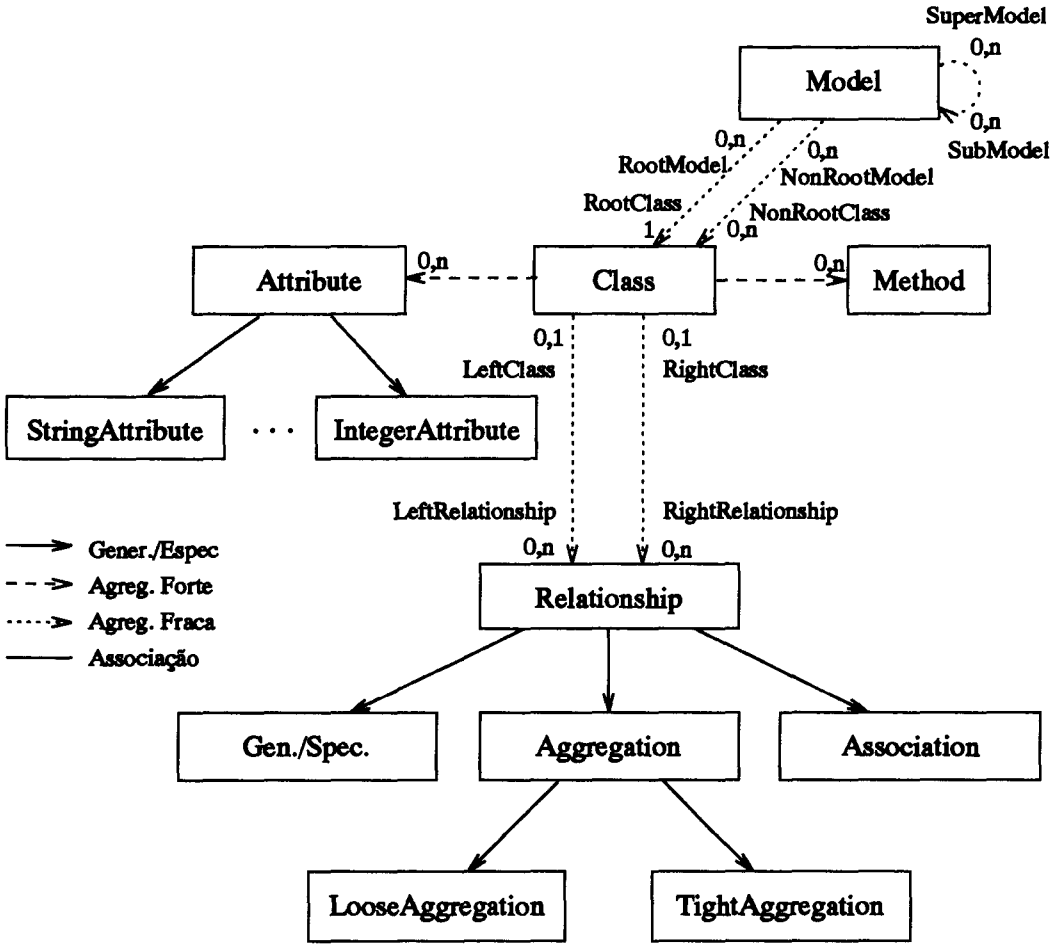

Figura 4.6: Modelo de Objetos de Stabilis

view representa o grupo ao qual o objeto pertence. Os construtores de **Object** recebem como parâmetros uma "string" (uma expressão de consulta), o modo da consulta e uma estrutura que permite armazenar o histórico das operações realizadas. O modo de uma consulta indica se ela a cria ou recupera um objeto. Existem três modos de consulta:

1. *Birth*: indica a criação de um novo objeto.
2. *Reincarnation*: recupera um objeto do banco de dados, de acordo com a expressão especificada em *sexpr* (construtor de **Object** na Figura 4.7).
3. *Provide*: combinação de *Reincarnation* e *Birth*. Se nenhum objeto for encontrado como especificado em *sexpr* então um novo objeto é criado.

O propósito do método *handle_message()* é permitir manipulação de invocações de métodos em consultas; *put()* é utilizado para atualizar os valores dos atributos de um objeto persistente; *relate()* e *unrelate()* relaciona/desrelaciona um objeto a outro especificando o papel do objeto relacionado; *is_a()* permite testar se um objeto pertence a uma determinada classe. A classe **Object** não é representada no esquema que especifica o meta-modelo.

4.3.2 Programa Esquema/Consulta

Em Stabilis, modelos estruturais descrevem o programa distribuído implementado pelo usuário e são especificados através de um programa esquema. Os objetos que representam as classes, atributos e relacionamentos do modelo estrutural da aplicação são instâncias das classes de Stabilis. Assim, uma associação é uma instância da classe **Association**, um atributo é uma instância da classe **Attribute**, etc... Todas as classes que compõem um programa esquema devem ser definidas a priori. Além disso, esquemas em Stabilis são conservativos, ou seja, Stabilis não permite evolução de esquema.

A Figura 4.8 apresenta um trecho de um programa esquema, que cria uma classe de nome **Pessoa** com atributos *nome* e *idade*. A "string" contendo a expressão passada para os construtores das classes de Stabilis é interpretada tal que os objetos: classe **Pessoa**, atributos *nome* e *idade* são criados de forma persistente.

Um programa consulta cria e/ou recupera objetos, que são instâncias de classes do modelo estrutural especificado no programa esquema. Assim, o programa esquema da aplicação que cria os objetos que representam as classes, atributos e relacionamentos — *nome*, *idade* e **Pessoa** no exemplo da Figura 4.8 — equivale a um programa consulta feito ao modelo estrutural de Stabilis. A Figura 4.9 apresenta o exemplo de um programa consulta feita ao modelo estrutural especificado pelo programa esquema da Figura 4.8.

```
class Object : public LockManager
{
 private:

  String object_name;
  Uid cache_uid;
  Expression* assign_expression;

 protected:

  RelationshipTable relationships;
  AttributeTable attributes;
  View* view;

  Object(const String& sexpr, View*, const Birth, OpHistory*);
  Object(const String& sexpr, View*, const Reincarnation, OpHistory*);
  Object(const String& sexpr, View*, const Provide, OpHistory*);

  OpHistory* make_permanent();
  OpHistory* make_volatile(const LockMode mode = READ);

 public:

  virtual OpHistory* handle_message(String signature, List<Argument>* args,
                                    Result*& reply_obj);
  OpHistory* put(String sexpr);
  OpHistory* relate(String related_object_role, Object* related_object);
  OpHistory* unrelate(String related_object_role, Object* related_object);
  Boolean is_a(String class_name);
  virtual ~Object();
};
```

Figura 4.7: Classe Object

```
OpHistory* oph = new OpHistory;
Class* m_Pessoa = new Class("Class(name='Pessoa')",...,BIRTH, oph);
StringAttribute* a_nome = new
StringAttribute("StringAttribute(name='nome')",..., BIRTH, oph);
IntegerAttribute* a_idade = new
IntegerAttribute("IntegerAttribute(name='idade')",..., BIRTH, oph);
m_Pessoa→relate("Attribute",a_nome);
m_Pessoa→relate("Attribute",a_idade);
Method* mth_descendentes = new Method("Method(signature='descendentes()'
&& return_type='Pessoas*')",...,BIRTH, oph);
Method* mth_pais = new Method("Method(signature='pais()' &&
return_type='Pessoas*')",...,BIRTH, oph);
m_Pessoa→relate("Method",mth_descendentes);
m_Pessoa→relate("Method",mth_pais);
```

Figura 4.8: Programa Esquema

```
OpHistory* oph = new OpHistory;
ObjectSet idosos("Pessoa(idade > 60)",...,oph);
for(i = 1; i ≤ idosos.card(); i++)
  {
    Pessoa* p = idosos.retrieve();
    cout ≪ p→descendentes() ≪ nl;
    idosos.forward();
  }

Pessoa m("Pessoa(nome=='Maria')",...,REINCARNATION, oph);
cout ≪ m.pais() ≪ nl;
```

Figura 4.9: Programa Consulta

No exemplo da Figura 4.9, *idosos* é uma variável que armazena um conjunto de objetos, instâncias da classe **Pessoa** com idade maior que 60 anos. O programa imprime na saída padrão os descendentes de cada pessoa que faz parte do conjunto *idosos*. Em seguida, é recuperado um instância da classe **Pessoa** que têm o atributo *nome* igual a `'Maria'` e impresso os pais.

4.3.3 Organização do Espaço de Objetos

Uma aplicação pode conter um grande número de modelos, cada um formado por várias classes, sendo que cada classe pode ter uma grande quantidade de instâncias. Portanto é necessário organizar o espaço de objetos de tal forma que as consultas possam ser resolvidas mais eficientemente. Com esta finalidade, Stabilis permite ao programador definir visões e contextos, que dividem o espaço de objetos. Visões estão intimamente relacionadas com índices. 'E mostrado a seguir a relação entre índices, visões e contextos.

Índices

Arjuna atribui uma identificação única (UID) a cada objeto persistente. Como objetos Stabilis são também objetos Arjuna, por herança, então eles também possuem uma identificação única. Como discutido anteriormente, máquina de busca de objetos mantém dois tipos de índices: de atributo e de relacionamento. Em Stabilis índices de atributos são implementados pelas classes **StringAttributeIndex** e **IntegerAttributeIndex**, e índices de relacionamentos são implementados pela classe **RelationshipIndex**. Instâncias de **StringAttributeIndex** e **IntegerAttributeIndex** contém uma tabela (instaciada de um "template" C++) que mapeiam valores a uma instância da classe **Pips**, conforme mostrado na Figura 4.10.

Um "pip" contém as informações estritamente essenciais à localização de um objeto. Estas informações são: a UID do objeto, a classe mais específica do objeto e o nó onde o objeto reside. Um valor é mapeado para um "pip" se o objeto referenciado pelo "pip" tem o atributo correspondente com aquele valor. Uma instância de **RelationshipIndex** tem os nomes da classe local e da classe relacionada, o papel relacionado e uma tabela que mapeia UIDs de instâncias da classes relacionada a "pips" de instâncias da classe local. A Figura 4.11 ilustra isto.

Visões

Uma visão em Stabilis é uma porção de um modelo estrutural de objetos, definida pela seleção de um subconjunto de classes [Cal96]. Além das classes selecionadas,tal subconjunto inclui a hierarquia de classes relacionadas definidas no modelo estrutural, para

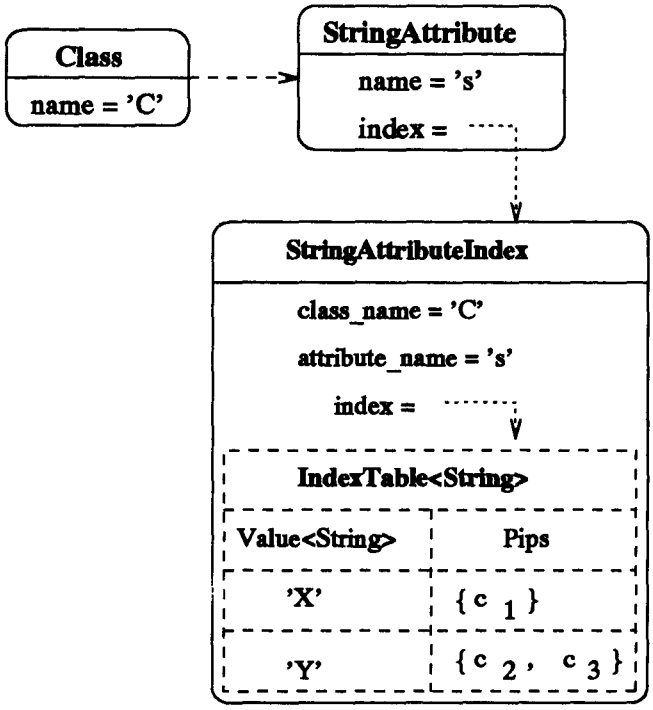

Figura 4.10: Índice de Atributo

garantir que o princípio do encapsulamento seja mantido, isto é, a seleção de determinada classe implica na seleção de todas as superclasses e todas as classes relacionadas, recursivamente. Um modelo estrutural pode ter qualquer número de visões associadas, podendo ocorrer interseções entre elas.

Uma visão define o escopo de uma consulta, ou seja, uma consulta é resolvida em uma visão ao invés de no modelo estrutural inteiro. Conseqüentemente, somente a informação de objetos que são instâncias das classes da visão são acessíveis. Mais especificamente, visões servem para os seguintes propósitos: segurança, performance, confiabilidade e administração. Uma visão é uma instância da classe View, subclasse de Object e portanto um objeto persistente. O construtor de View recebe como parâmetros o nome da visão e o modelo que contém as classes que compõem a visão. Uma instância de View contém portanto uma lista de classes. Cada classe tem um nome e contém uma lista de atributos (somente os indicados como atributos-chaves no esquema) e uma lista de relacionamentos (também apenas os indicados como chaves no esquema). Cada atributo tem um nome e uma referência para o índice correspondente. Cada relacionamento tem o nome da classe relacionada, o nome do papel relacionado e uma referência para o índice correspondente. Na verdade, uma visão equivale a um conjunto de índices, conforme mostrado na Figura 4.12.

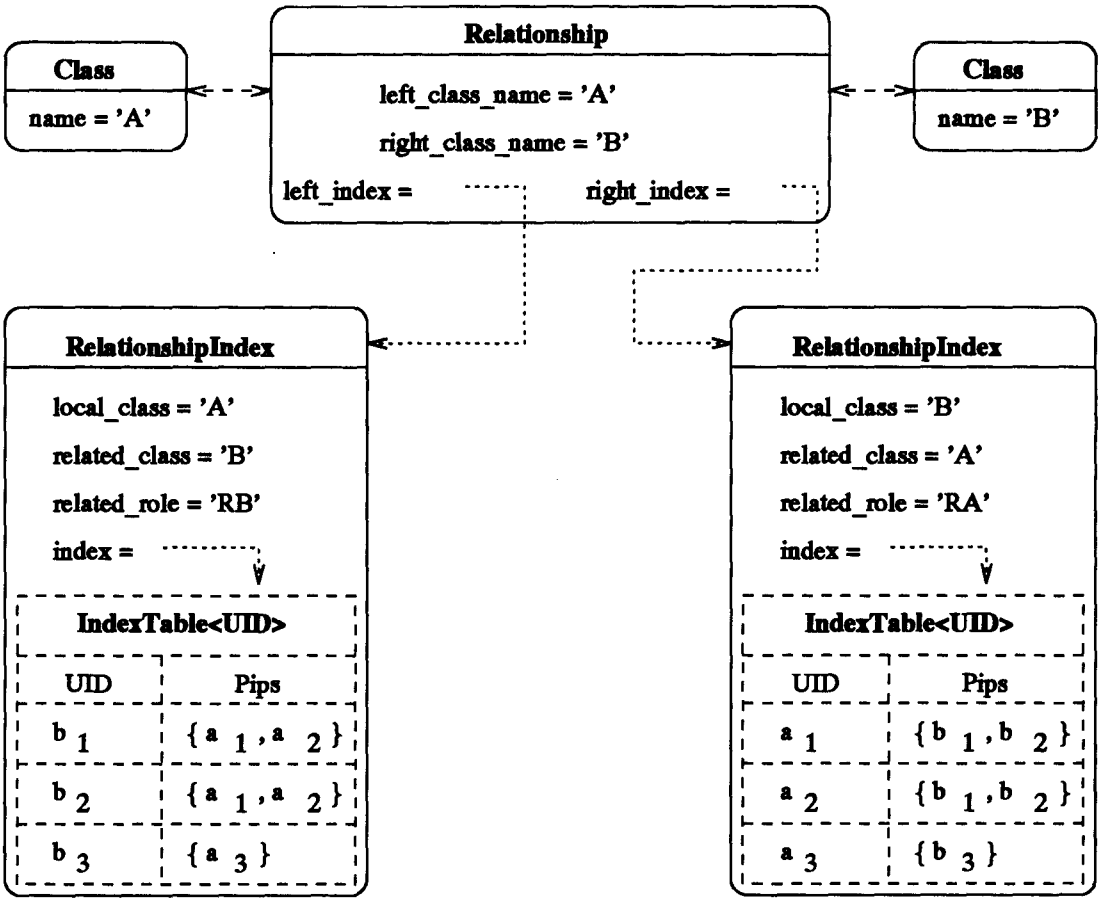

Figura 4.11: Índice de Relacionamento

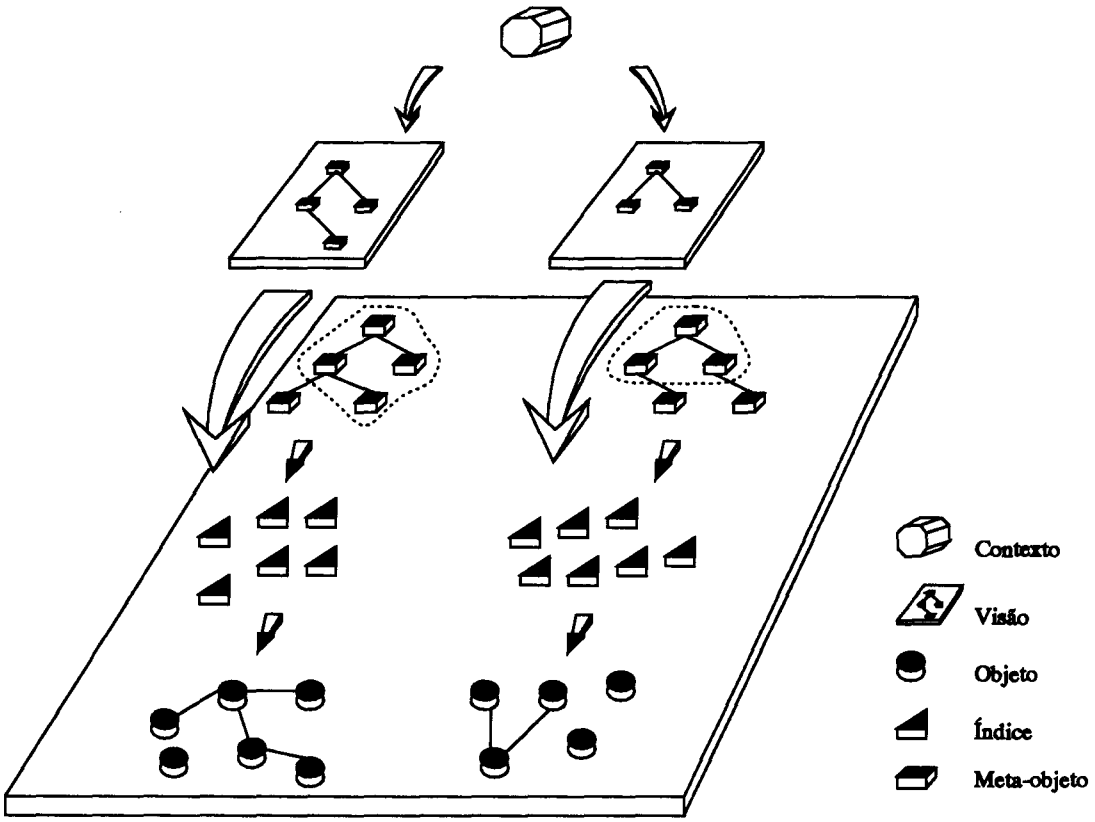

Figura 4.12: Arquitetura de Stabilis

Contextos

Modelos estruturais e visões constituem uma informação básica e conseqüentemente necessitam de gerenciamento próprio, tais como controle de acesso. Por esta razão, um modelo estrutural de objetos e suas visões correspondentes são mantidas por um contexto. Um contexto contém uma coleção de modelos estruturais e todas suas visões associadas. Então, programas de administração do sistema podem interagir com um contexto para criar, atualizar e apagar modelos estruturais e visões. Pode-se observar pela Figura 4.12 que visões são como planos, cuja projeção são conjuntos de meta-objetos (representados por cubos). Esses meta-objetos representam as classes selecionadas pela visão e contém os índices que referenciam os objetos. Um contexto armazena as visões.

Os nomes de classes, modelos e visões são relativos ao contexto, enquanto os nomes de contextos são globais. Isto é, classes e visões são nomeados dentro de cada contexto independentemente de qualquer outro contexto. A criação de um contexto gera um conjunto de objetos que representam as classes do modelo estrutural de Stabilis, mostrado na Figura 4.6. Esse conjunto especial de objetos faz parte da visão pré-definida chamada **Meta**. Essa visão contém também os objetos que representam as classes de Stabilis da aplicação do usuário.

4.4 Consultas

Como visto anteriormente, o escopo de consultas é definido por visões, isto é, consultas são resolvidas em uma visão. Uma questão independente do escopo de resolução da consulta, mas igualmente importante, é a linguagem de consulta: sua relação com a linguagem de programação, sua semântica e sua expressividade, ou seja, o quanto ela permite expressar consultas complexas de maneira concisa. Sobre esses aspectos, segundo Bancilhon [LRV92], ao se projetar uma linguagem de consulta para banco de dados orientado a objetos deve-se responder os seguintes questionamentos:

1. A linguagem de consulta pode violar o encapsulamento?
2. O que é consultado: dados, métodos ou ambos?
3. O que é a resposta de uma consulta?
4. Como a linguagem de consulta integra-se com a linguagem de programação?
5. Qual deve ser o relacionamento com o sistema de tipos?

A idéia de encapsulamento é que métodos são a única maneira de operar sobre objetos. Se a linguagem de consulta não pode violar o encapsulamento, então o acesso a objetos

é feito através de métodos e os dados são invisíveis. Encapsulamento é um mecanismo para reforçar uma boa modularidade de código e uma boa disciplina de programação. Em Stabilis, a linguagem de consulta viola o encapsulamento sintaticamente, embora na sua execução, a leitura e atualização de atributos de objetos são traduzidas para chamadas de métodos. Portanto, Stabilis trata atributos como métodos sem parâmetros no caso de leitura e com um único parâmetro no caso de atualização.

Métodos aumentam o poder de expressão da linguagem de consulta porque permitem derivar e utilizar informações complexas dos objetos de forma encapsulada. Isto torna mais concisa a notação de consultas complexas. Logo, métodos são desejáveis na linguagem de consulta. Porém, seu uso pode introduzir problemas semânticos quando integrado com a linguagem de programação.

Quanto à integração da linguagem de consulta com a linguagem de programação podem ocorrer os seguintes casos [LRV92]: a linguagem de consulta é distinta da linguagem de programação; a linguagem de consulta é distinta, mas assemelha-se à linguagem de programação; ou a linguagem de consulta é um subconjunto da linguagem de programação. Em Stabilis, a linguagem de consulta é C++. As expressões que criam ou recuperam objetos — passadas para os construtores das classes — utilizam a própria sintaxe usada para invocação de construtores em C++.

Uma consulta em Stabilis é uma especificação de objetos de acordo com suas propriedades e o seu resultado é formado pelo conjunto de objetos cujas propriedades estão em conformidade com as condições expressas na consulta. O propósito de consultas em Stabilis é permitir que objetos inteiros possam ser recuperados especificando algumas de suas propriedades. Desde que um objeto foi recuperado, então todas as suas propriedades são acessíveis através dos métodos públicos da classe. Logo, atributos de tipos simples como inteiros e strings não podem ser retornados como o resultado de uma consulta, uma vez que não são considerados objetos em Stabilis.

Uma consulta é formulada contra um modelo estrutural, isto é, os predicados são expressos em termos de classes, atributos, relacionamentos e métodos. Um conjunto de objetos obtido como o resultado de uma consulta é composto de instâncias de classes já definidas no modelo estrutural.

A gramática da linguagem de consulta de Stabilis, definida em BNF ("Backus-Normal-Form"), está apresentada na Figura 4.13. Esta gramática mostra como expressões de consulta devem ser especificadas.

4.4.1 Expressões de Caminho

Em Stabilis, consultas são formadas por expressões de caminho que permitem manipular facilmente estruturas aninhadas mantendo a legibilidade. Tais expressões descrevem

```
<query_expression> → <simple_query> [. <methods>] |
                     <query_expression> <set_operator> <query_expression> |
                     (<query_expression>)

<simple_query> → <class_name>(<where_clause>)

<methods> → <method_invocation> |
            <method_invocation> . <methods>

<set_operator> → <intersection_operator> | <union_operator>

<intersection_operator> → &

<union_operation> → |

<class_name> → IDENTIFIER

<where_clause> → <term> |
                 <where_clause> <logical_operator> <where_clause> |
                 (<where_clause>)

<logical_operator> → <and_operator> | <or_operation>

<and_operator> → &&

<or_operator> → ||

<term> → [<relationship_path>] <property> <relational_operator> <value>

<relationship_path> → <role_name> <path_operator> |
                      <relationship_path> <role_name> <path_operator>

<path_operator> → ::

<role_name> → IDENTIFIER

<property> → <attribute_name> |
             <methods>

<relational_operator> → <equal_operator> | <different_operator> |
                        <great_operator> | <greater_equal_operator> |
                        <less_operator> | <less_equal_operator>

<attribute_name> → IDENTIFIER

<method_invocation> → <method_name> (<arguments_list>)

<arguments_list> → ε |
                   <arguments>

<arguments> → <value> |
              <simple_query> |
              <arguments> , <arguments>

<method_name> → IDENTIFIER
```

Figura 4.13: Gramática da linguagem de consulta de Stabilis

caminhos ao longo da hierarquia de composição e podem ser vistas como composição de métodos [BNPS92].

Em Stabilis, consultas só podem retornar objetos inteiros para manter a concordância com o modelo de objetos. Portanto, não há uma integração forte entre C++ e a linguagem de consulta de Stabilis.

4.4.2 Utilização de Métodos em Consultas

Instâncias da classe **Method** (Figura 4.6), armazenam assinaturas dos métodos de classes definidas pelo programador. O propósito desta classe é permitir a consulta dos métodos existentes em uma classe e fazer geração de código automática de seus respectivos cabeçalhos.

Do ponto de vista de formulação de consultas, métodos podem ser considerados similares a atributos. Portanto, a princípio, uma invocação de método pode sempre ser usada onde um atributo pode [BNPS92]. Cada parâmetro de entrada do método pode ser uma constante ou uma expressão de caminho. Logo, cada parâmetro está explícita ou implicitamente ligado a uma classe.

A invocação de um método pode retornar um objeto já existente no banco de dados ou um novo objeto. Para o caso de criação de novos objetos, Stabilis garante a consistência do banco de dados, visto que o método só pode criar instâncias de objetos utilizando os construtores das classes, gerados automaticamente pelo próprio Stabilis, que por sua vez garantem que os objetos são armazenados no banco de dados.

Outra questão importante com relação ao uso de métodos em consultas é efeitos colaterais. Um método com efeitos colaterais modifica o objeto sobre o qual é invocado ou alguns de seus parâmetros. O uso de métodos com efeitos colaterais em consultas pode torná-las dependentes da ordem de avaliação de seus predicados. Logo é importante restringir o uso de métodos apenas àqueles cujas implementações não têm efeitos colaterais [BNPS92]. Quando permite-se a utilização de métodos em consultas, a avaliação da consulta pode não terminar. Isto porque pode existir um método invocado na consulta, cuja execução não termina para um dado conjunto de valores de entrada (argumentos). A determinação se um método termina não é possível em geral [BNPS92]. Em Stabilis, é responsabilidade do programador tanto não permitir que métodos causem efeitos colaterais como garantir que a execução dos mesmos sempre termine.

Os métodos em Stabilis são codificados e compilados em C++. Como a linguagem de consulta de Stabilis não possui variáveis, os parâmetros de um método podem ser apenas constantes ou expressões de caminhos. Além disso, para que Stabilis possa fazer a execução de métodos, é feita a geração de código automática do método *handle_message()*, sobrecarregado em cada classe, que recebe como parâmetros os objetos ou constantes que são parâmetros do método a ser executado e a "string" contendo a assinatura do método. O

método *handle_message* invoca para o objeto o método especificado na consulta. Métodos só podem ser usados em consultas cujo modo é *Reincarnation*, pois é preciso garantir que o objeto sobre o qual ele será aplicado já exista.

Semântica de Métodos em Consultas

Para ter-se uma idéia precisa da semântica de métodos em consultas, considere primeiramente a consulta *"Pessoa(salario() > 20)"*. O resultado do método *salario()* é usado para determinar se o objeto sobre o qual ele foi aplicado faz parte ou não do conjunto solução. Portanto o método *salario()* deve ser aplicado a cada objeto que é instância direta ou indireta da classe **Pessoa**, ou seja, *ext*(***Pessoa***)*. Logo, é necessário a existência de um índice por classe que indexe todas as instâncias. O resultado desta consulta são instâncias da classe **Pessoa**. Nesta semântica de execução de métodos não é relevante se o retorno do método é um objeto ou um valor de tipo primário. Isto porque o resultado do método é utilizado apenas para determinar se o objeto sobre o qual foi aplicado pertence ao conjunto solução. Porém, quanto aos operadores relacionais (<, <=, >=, ==), se o resultado é um objeto, então a classe do objeto sobre o qual o método é aplicado deve sobrecarregá-los. Esquematicamente, essa semântica para execução de métodos é equivalente ao mapeamento de um conjunto **A** em um conjunto **B**, através de um método m, onde $B \subseteq A$ e $m \in A$. A Figura 4.14 exemplifica isto para a consulta mostrada anteriormente. Por analogia ao modelo relacional de banco de dados, esta semântica é chamada de *seleção*, pois o método é utilizado como/em um predicado que serve para selecionar os elementos do conjunto solução.

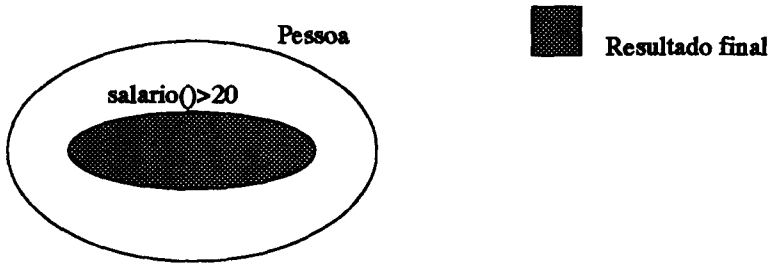

Figura 4.14: Semântica de Seleção

Considere agora a consulta *"Pessoa(idade < 60).endereco()"*, ilustrada pela Figura 4.15. A resolução desta consulta pode ser dividida em duas etapas: *(i)* recuperar as instâncias de **Pessoa** que tem o atributo *idade* menor que 60; *(ii)* em seguida, aplicação do método *endereco()* sobre os objetos obtidos na primeira etapa. O resultado final da consulta são instâncias da classe **Endereco** (Figura 4.15) e não é necessário um índice específico para aplicação do método, pois os objetos sobre os quais ele será aplicado já foram recuperados na primeira etapa da resolução da consulta. Esquematicamente, essa

semântica é equivalente ao mapeamento de um conjunto A em um conjunto B, através do método m, onde $m \in A$.

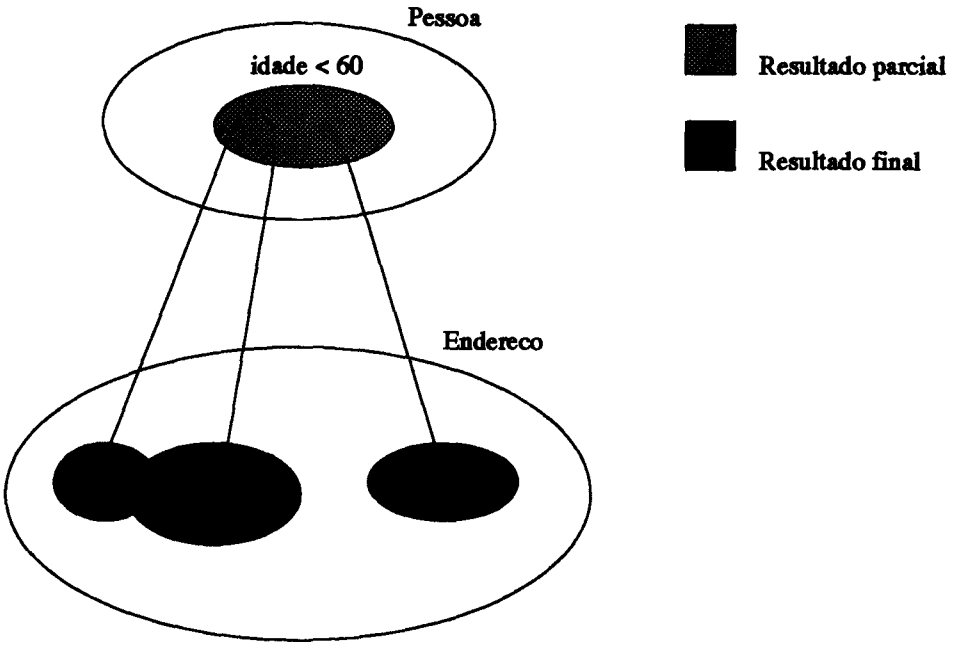

Figura 4.15: Semântica de Mapeamento

4.5 Resumo

Stabilis é uma ferramenta para construção de aplicações distribuídas tolerantes a falhas, implementado em Arjuna. Arjuna fornece, através do mecanismo de herança, persistência e ações atômicas aos objetos de Stabilis. Aplicações típicas que podem ser construídas usando Stabilis são máquinas de busca de objetos. Uma máquina de busca de objetos é um sistema de meta-informações cujo propósito é prover uma interface orientada a objetos para informações contidas em recursos de rede em ambientes distribuídos de larga escala. Aplicações em Stabilis são compostas de dois programas: esquema e consulta. Programas esquemas definem as relações estáticas do modelo de objetos da aplicação em termos de classes, atributos e relacionamentos. Programas consultas criam e/ou recuperam objetos que são instâncias de classes definidas no programa esquema. Stabilis permite ao programador definir visões do modelo estrutural da aplicação com propósitos de eficiência e administração. Visões são agrupadas em contextos, tal que seus nomes são relativos ao contexto ao qual pertencem. Contextos servem para propósitos de gerenciamento de visões e modelos estruturais. A linguagem de consulta de Stabilis comporta duas semânticas de

execução de métodos (seleção e mapeamento), sendo que a semântica de seleção exige a existência de um índice por classe.

Capítulo 5

RStabilis

5.1 Introdução

RStabilis (*Reflexive Stabilis*) foi construído a partir de Stabilis [Buz94, Cal96] adicionando-se um modelo de reflexão computacional. Como RStabilis é reflexivo, ele permite aos componentes da aplicações se comportarem diferentemente, de acordo com as necessidades do programador, sem alterações significativas nos componentes, mantendo a transparência.

Ao elaborar um modelo de reflexão computacional para um sistema, deve-se analisar os requisitos típicos das aplicações que utilizarão o sistema. Isto porque o modelo de reflexão computacional adotado por um sistema influi diretamente na possibilidade de se ter vários comportamentos ou não para um mesmo componente, sendo que esses comportamentos podem ser ortogonais ou não à aplicação. Os modelos aqui apresentados — em sua maioria, variações dos modelos apresentados no Capítulo 2 e adequados à arquitetura de Stabilis — foram analisados visando a concepção de um modelo para a implementação de RStabilis. Os critérios considerados foram a funcionalidade do modelo e sua complexidade de implementação. A Figura 5.1 mostra a hierarquia entre RStabilis, Stabilis e Arjuna.

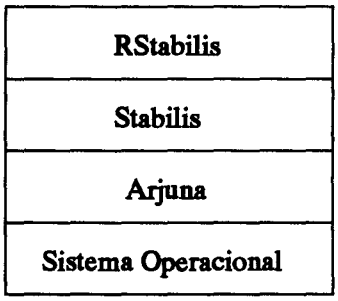

Figura 5.1: Hierarquia de Arjuna, Stabilis e RStabilis

5.2 Possíveis Modelos de Reflexão para Stabilis

Um modelo de reflexão computacional pode ser entendido como uma hierarquia de classes que cria uma arquitetura de software capaz de permitir que meta-objetos controlem a execução de objetos. Um protocolo é um conjunto de regras que visa determinado fim. Dentro do contexto de reflexão, um protocolo de meta-objetos estabelece as regras de criação de meta-objetos e define métodos que permitem meta-objetos controlarem a execução de objetos. Para um mesmo modelo pode-se ter protocolos de meta-objetos diferentes. Nesta seção enfatiza-se possíveis modelos de reflexão para Stabilis, porém não considerando especificamente um protocolo de meta-objetos associado ao modelo em questão.

5.2.1 Classe/Metaclasse

No modelo Classe/Metaclasse cada classe está relacionada no máximo a uma única metaclasse. A metaclasse contém o código dos metamétodos, sendo que a relação entre métodos e metamétodos é de 1-1. Dentro desta abordagem para acrescentar algum comportamento ao método reflexivo, basta inserir o código desejado no respectivo metamétodo. O metamétodo é então responsável pela invocação do método equivalente no nível base. O relacionamento de classe/metaclasse pode ser implementado por uma classe, **MetaclassRelationship**, que seria subclasse da classe **Relationship** de Stabilis. A Figura 5.2 exemplifica este modelo: a classe **A** possui uma única metaclasse chamada **M**. A notação usada para metaclasses é um retângulo de bordas arredondas e o relacionamento entre classe e metaclasse é uma seta pontilhada bidirecional.

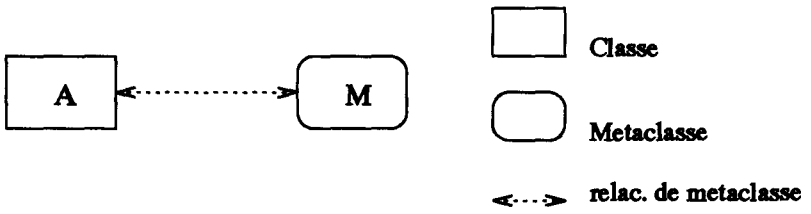

Figura 5.2: Modelo Classe/Metaclasse

Nesse modelo, uma metaclasse é específica para determinada classe, o que equivale dizer que metaclasse não é uma classe ordinária, e portanto o seu código não pode ser reutilizado por outras classes. Além disso, não se pode ter múltiplas atividades no metanível estruturadas adequadamente para um mesmo objeto, pois todas as atividades devem ser encapsuladas numa única metaclasse. Esse modelo é bastante próximo da implementação de metaclasses em Smalltalk, porém podendo variar na cardinalidade da relação objeto/meta-objeto, que em Smalltalk é N-1 e no modelo aqui apresentado pode

ter tanto cardinalidade 1-1 quanto N-1, dependendo apenas do protocolo de criação de meta-objetos. A cardinalidade de N-1 pode ser garantida criando-se automaticamente a instância da metaclasse assim que a primeira instância do nível base é criada, associando-a à classe, de tal forma que para instâncias posteriores a localização do meta-objeto pode ser determinada.

5.2.2 Classe/Múltiplas Metaclasses

Nesse modelo uma classe pode estar relacionada a múltiplas metaclasses, sendo que cada metaclasse provê um comportamento diferente para as instâncias da classe. É necessário indicar qual o relacionamento encontra-se ativo, para que seja feita a escolha correta do comportamento desejado. Este modelo está apresentado na Figura 5.3: a classe A possui metaclasses M1 , M2, ..., Mn.

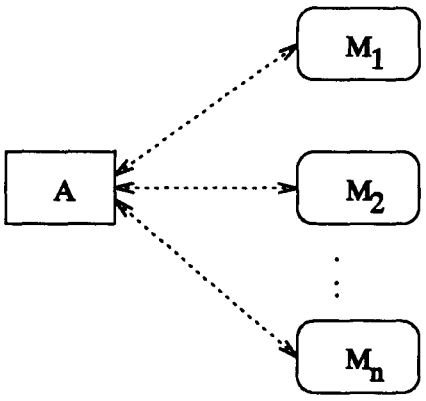

Figura 5.3: Modelo Classe/Metaclasses

Para mudar a metaclasse ativa, **Object** implementaria um método *activate_metaclass(String metaclass)* que receberia como parâmetro o nome da metaclasse. O programador deve então invocar *activate_metaclass()* antes de invocar o método do objeto. Esse modelo não permite que abstrações distintas — por exemplo: paralelismo e controle de acesso — possam ser usadas simultaneamente, a não ser que estejam implementadas dentro da mesma metaclasse, pois apenas um relacionamento pode estar ativo por vez. Isto acontece porque não é possível determinar para quais metaclasses o método do nível base possui um metamétodo correspondente.

5.2.3 Objeto/Meta-objeto

O modelo objeto/meta-objeto não considera a existência de metaclasses reais. Meta-objetos devem ser instâncias de uma classe pré-definida **MetaObject** ou de subclasses desta.

Esse modelo é bastante flexível pois nele metaclasses são classes ordinárias não guardando nenhuma relação direta com as classes. A associação é feita diretamente entre instâncias, ou seja, entre objetos e meta-objetos. Os métodos da metaclasse devem encapsular tanto o controle do nível base quanto a abstração a ser usada pelo nível base, o que limita sua reutilização, pois que acabam por manipular informações intrínsecas de classes do nível base. A Figura 5.4 mostra um esquema deste modelo. Aqui, a notação gráfica para metaclasses é a mesma de classe pois metaclasses são classes ordinárias.

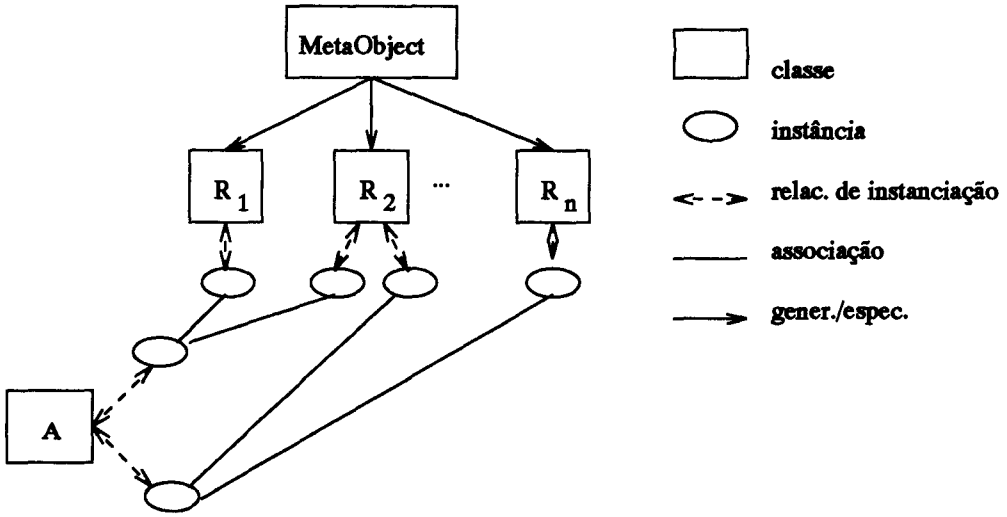

Figura 5.4: Modelo Objeto/Meta-objeto

5.2.4 Objeto/Meta-objeto/Refletores

Para esse modelo não há relacionamento explícito entre classes e metaclasses. O relacionamento é feito apenas entre instâncias. O modelo também considera que a metaclasse não deve implementar simultaneamente o protocolo de meta-objetos e a abstração a ser usada pelo nível base. Assim, é introduzido o conceito de refletores para permitir a separação explícita entre o protocolo e a abstração implementada no metanível. Refletores são instâncias de subclasses da classe pré-definida **Reflector**, que encapsulam as abstrações a serem usadas pelo nível base. Meta-objetos são instâncias da classe pré-definida **MetaObject** ou de suas subclasses e implementam o protocolo de meta-objetos. As abstrações implementadas pelas classes refletoras são a princípio ortogonais às classes que as utilizam. O relacionamento apenas entre instâncias permite que as abstrações implementadas pelas classes refletoras sejam compartilhadas por várias classes. Dentro desse mesmo modelo, pode-se considerar duas abordagens:

- *(i)* cada objeto do nível base está associado diretamente ao seu meta-objeto e aos refletores, conforme representado na Figura 5.5.

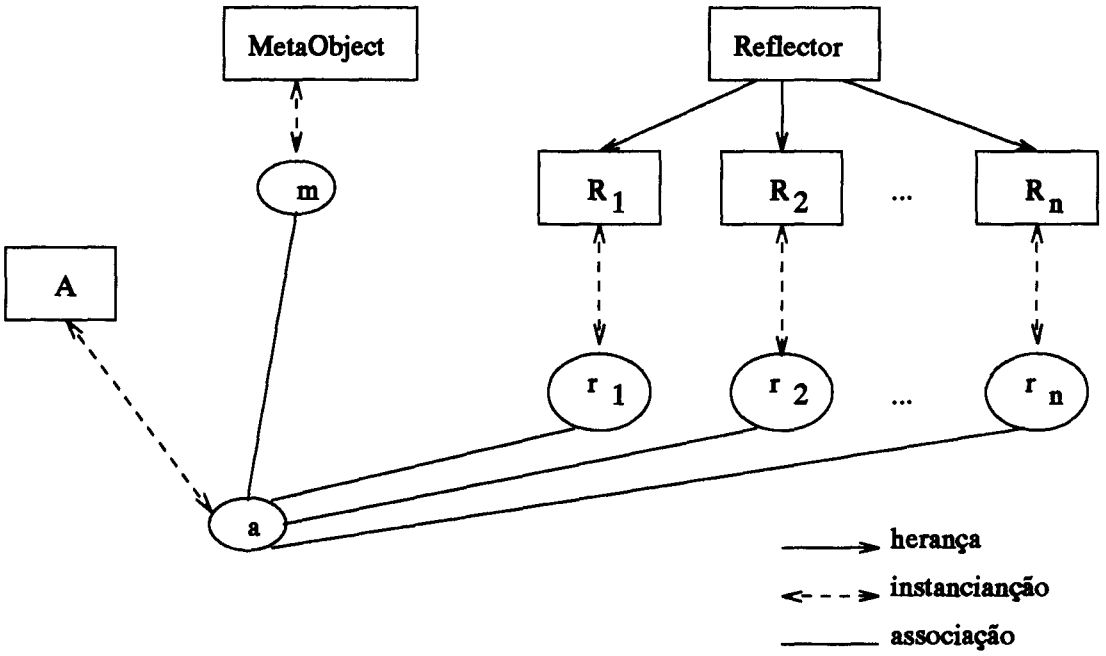

Figura 5.5: Modelo Objeto/Meta-objeto/Refletores

- *(ii)* cada objeto do nível base está associado diretamente a seu meta-objeto e esse aos refletores, representado na Figura 5.6.

A primeira abordagem considera que os refletores "pertencem" ao objeto. Portanto o meta-objeto, para controlar a ordem de execução dos refletores, deve consultar o objeto para obter informações sobre os refletores. Essa abordagem é interessante quando o protocolo de criação de meta-objetos e refletores é feito por instância e não por classe, ou seja, cada objeto pode estar associada a refletores que são instâncias de classes diferentes. A segunda abordagem considera que os refletores "pertencem" ao metanível — mais especificamente ao meta-objeto — e que a única ligação entre o nível base e o metanível é a associação objeto/meta-objeto. O objetivo nessa segunda abordagem é procurar manter o metanível o mais transparente possível para o nível base.

Em ambas abordagens, as classes refletoras são classes ordinárias, subclasses de Reflector. Isto permite evidenciar a separação entre classes que implementam protocolos de meta-objetos e classes que implementam abstrações a serem usadas pelo nível base. Nas Figuras 5.5 e 5.6 pode-se observar que não há relacionamento entre a classe A, MetaObject, R1, R2, ..., Rn e sim apenas entre suas instâncias. Portanto R1, R2, ..., Rn são classes ordinárias e não metaclasses para A. O meta-objeto **m** — instância de MetaObject — é responsável pelo protocolo de execução dos refletores da instância **a**.

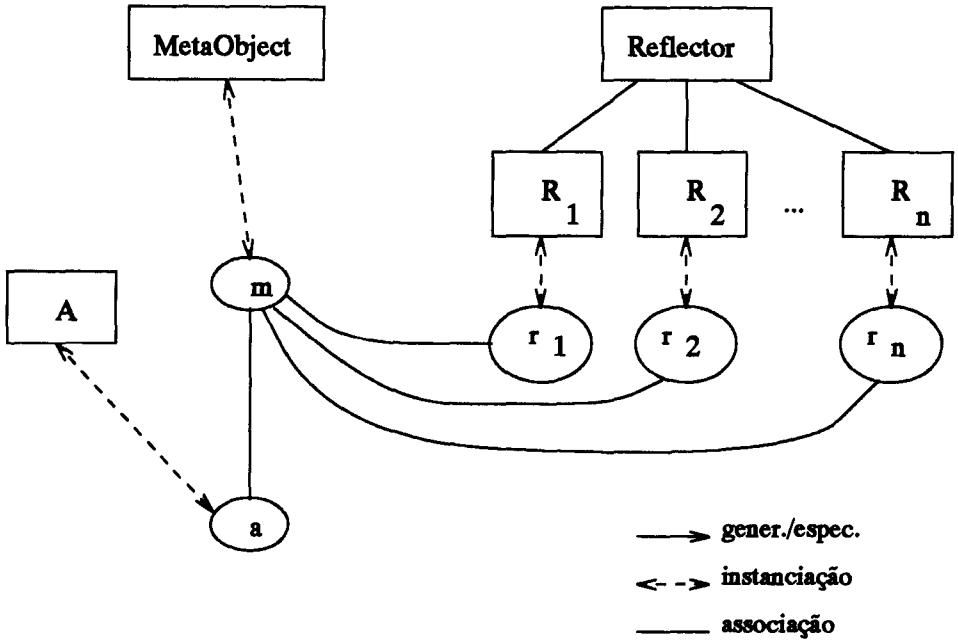

Figura 5.6: Modelo Objeto/Meta-objeto/Refletores

Modelo	Classificação segundo Ferber
classe/metaclasse	meta-objeto ou metaclasse
classe/múltiplas metaclasses	meta-objeto ou metaclasse
objeto/meta-objeto	meta-objeto
objeto/meta-objeto/refletores	meta-objeto

Tabela 5.1: Classificação dos Modelos de Reflexão para Stabilis

A classificação para modelos de reflexão computacional adotada por Ferber [Fer89], mostrada no Capítulo 2, leva em consideração a cardinalidade do relacionamento objeto/meta-objeto e se métodos são também objetos. Mas ela não comporta como um critério de classificação a existência de uma relação implícita entre classes e metaclasses. Ou seja, se uma metaclasse é específica para determinada classe. Isto ocorre sempre para modelos de metaclasse, porém não acontecendo o mesmo para modelos de meta-objeto. Portanto, a classificação de Ferber é limitada, não informando todas as características de um determinado modelo de reflexão. Isto explica porque o modelos classe/metaclasse e classe/múltiplas metaclasses tanto podem ser classificados de meta-objeto como de meta-classe, conforme mostra a Tabela 5.1

5.3 Metacircularidade e Protocolos de Meta-objetos

Em todos os modelos vistos anteriormente um meta-objeto também pode ter um meta-objeto — um metameta-objeto — que por sua vez também pode ter um meta-objeto e assim por diante. Logo há uma torre de meta-objetos onde o nível de baixo é sempre controlado por um nível acima. Muitos modelos de reflexão quebram essa metacircularidade criando um objeto cujo meta-objeto é ele próprio. Smalltalk [Gol83] faz isto tornando todas as metaclasses subclasses da classe pré-definida **MetaClass**, que é metaclasse dela mesma. Para os modelos apresentados na seção anterior, a metacircularidade não é quebrada no próprio modelo, ficando a cargo do protocolo de criação de meta-objetos.

Outra questão em relação aos modelos de reflexão que pode ser tratada quase sempre separadamente do modelo adotado é a cardinalidade da relação objeto/meta-objeto. Por exemplo, no modelo de Classe/Metaclasse cada objeto pode ter um meta-objeto ou todos objetos compartilham o mesmo meta-objeto. O compartilhamento de meta-objetos é útil quando eles operam sempre da mesma forma, independentemente do objeto. Porém, se o meta-objeto armazena informações específicas sobre o objeto, então é recomendável que os objetos estejam associados a meta-objetos diferentes. A cardinalidade objeto/meta-objeto está relacionada diretamente com o protocolo de criação dos meta-objetos. Se um meta-objeto é compartilhado por vários objetos é aconselhável que ele já exista antes da criação de qualquer objeto. Isto ocorre em Smalltalk, onde o meta-objeto — na verdade, a metaclasse — é criado automaticamente quando se cria a classe do nível base. No modelo de Objeto/Meta-objeto/Refletores cada objeto tem apenas um meta-objeto. Porém, cada objeto pode ter um conjunto de refletores diferentes de outro objeto que é instância da mesma classe, dependendo apenas do protocolo de criação de meta-objetos adotado.

5.4 Protocolo de Meta-objetos de RStabilis

Dentre os modelos de reflexão apresentados neste capítulo, o modelo de Objeto/Meta-objeto/Refletores é o mais flexível e permite maior reutilização, resultante de uma estruturação melhor do metanível. Nele é feita a separação entre classes que implementam protocolos de meta-objetos e que implementam abstrações a serem usadas pelo nível base. O modelo de RStabilis é obtido acrescentando-se ao modelo de objetos de Stabilis, o modelo de reflexão Objeto/Meta-objeto/Refletores. A Figura 5.7 mostra como o modelo de reflexão se relaciona com o modelo de objetos de Stabilis. A classe **RObject** (*Reflexive Object*) é subclasse da classe **Object** de Stabilis e descreve objetos reflexivos. **RObject** é instância de **Class**. As classes **MetaObject** e **Reflector** são subclasses de **RObject**, indicando que meta-objetos e refletores também são objetos reflexivos e portanto, uns e outros também podem ter meta-objetos e refletores.

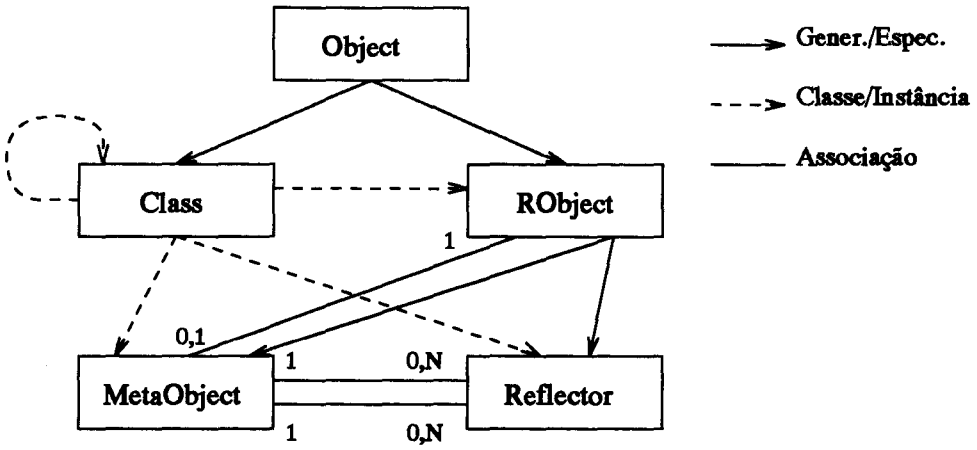

Figura 5.7: Modelo de Reflexão para Stabilis

A associação entre RObject e MetaObject indica que um objeto reflexivo pode ter no máximo um meta-objeto e que um meta-objeto está associado a um único objeto do nível base. Logo, não há compartilhamento de meta-objetos. Por sua vez, cada meta-objeto tem dois conjuntos de refletores. Os refletores podem ser ativados antes e/ou depois da execução do método reflexivo, dependo a qual conjunto o refletor pertença. Observa-se que esse modelo equivale à variante do modelo objeto/meta-objeto/refletores apresentado na Figura 5.6. Optou-se por esta variante porque ela oferece uma transparência maior do metanível para o nível base. A Figura 5.8 mostra a interface da classe MetaObject. O atributo *before_reflectors* contém o conjunto de refletores que devem ser ativados antes do método reflexivo. Analogamente *after_reflectors* contém os que devem ser ativados após o método reflexivo. Os métodos privados *get_before_reflectors()* e *get_after_reflectors()* recuperam os refletores do objeto. O atributo *base_obj_uid* contém a identificação única do objeto do nível base e é usado pelo método *get_base_level_object()* para recuperar o objeto reflexivo. O método *handle_reflexive_message()* faz a ativação dos refletores.

5.4.1 Protocolo de criação de meta-objetos

Meta-objetos podem ser criados de forma explícita ou implícita. Dentro do modelo de Objeto/Meta-objeto/Refletores, criação implícita significa que ao se criar um objeto sabe-se a priori de quais classes seu meta-objeto e refletores são instâncias. Logo, criação de meta-objetos e refletores torna-se transparente. Para isto, a especificação das classes refletoras e de meta-objetos para determinada classe do nível base deve ser feita durante a criação do programa esquema da aplicação, relacionando-as. Portanto, para criação implícita do metanível todos os objetos de uma mesma classe terão meta-objetos e refletores similares.

```
class MetaObject : public RObject
{
protected:

    String base_obj_uid;

    ObjectSet* before_reflectors;
    ObjectSet* after_reflectors;

    OpHistory* get_before_reflectors();
    OpHistory* get_after_reflectors();

    OpHistory* get_reflectors();                    // busca conjunto de refletores

    String get_base_obj_uid() { return base_obj_uid; }
    OpHistory* get_base_level_object(RObject*&);

 public:

  MetaObject(String sexpr, View*, const Birth, OpHistory*, unsigned meta_level =
1);
  MetaObject(String sexpr, View*, const Reincarnation, OpHistory*);
  MetaObject(String sexpr, View*, const Provide, OpHistory*);

  ~MetaObject();

  virtual OpHistory* handle_reflexive_message(String signature,
                                              List<Argument>* param_list,
                                              Result*& reply_obj);
};
```

Figura 5.8: Interface da classe **MetaObject**

A criação explícita permite a cada objeto ter um conjunto de refletores diferentes. Por exemplo, a instância `Pessoa(''pessoa1'', 15)` pode ter como refletores instâncias das classe DistributedExecution e Security enquanto `Pessoa(''pessoa2'', 20)` tem como refletores instâncias de RecoveryBlock, DistributedExecution e MethodInvocationAccount. A desvantagem de criação explícita é que o programador ao criar o objeto, deve informar os nomes das classes refletoras e respectivos parâmetros. Adotou-se criação implícita, porém não sendo necessário informar as classes refletoras e de meta-objeto durante a criação do esquema. Para este propósito, a classe RObject define o método *create_meta_level()* que é invocado automaticamente quando uma instância do nível base é criada, fazendo a criação do metanível. Esse método deve ser sobrecarregado em cada classe que usa reflexão. A Figura 5.9 mostra a interface da classe RObject. Stabilis gera automaticamente em cada classe o cabeçalho do método *create_meta_level()*. O programador deve colocar nesse método a criação dos refletores e do meta-objeto bem como associá-los. Assim, as classes refletoras e de meta-objetos são estabelecidas durante a programação da aplicação.

```
class RObject : public Object
{
 public:

   RObject(String sexpr, View*, const Birth, OpHistory*, unsigned meta_level = 1);
   RObject(String sexpr, View*, const Reincarnation, OpHistory*);
   RObject(String sexpr, View*, const Provide, OpHistory*);

   ~RObject();

   virtual OpHistory* create_meta_level();
};
```

Figura 5.9: Interface da classe RObject

Como dito anteriormente, um meta-objeto pode também ter um meta-objeto: um metameta-objeto. Quando o metameta-objeto é instância da mesma classe do meta-objeto é necessário prover uma forma de se quebrar a metacircularidade, pois o método *create_meta_level()*, invocado automaticamente durante a criação de uma instância do nível base, contém chamada de construtores da própria classe; o que causaria a criação de meta-objetos indefinidamente. Para isso, os construtores das classes gerados automaticamente por RStabilis, cujo modo de consulta é *Birth*, contêm o parâmetro **unsigned** *meta_level* com valor "default" 1 (TRUE). Este parâmetro é usado para decidir se o

método *create_meta_level()* deve ser invocado ou não. Assim, a criação de metameta-objetos que são instâncias da mesma classes dos meta-objetos é feita invocando-se o construtor com o parâmetro *meta_level* igual a 0 (FALSE). O programador também pode usar este parâmetro para criar um metanível diferente para uma instância específica, ou seja, meta-objeto e refletores diferentes dos invocados no método *create_meta_level()*. Para isto ele deve: *(i)* chamar o construtor da classe do nível base com o parâmetro *meta_level* igual a 0 (FALSE) para evitar que o método *create_meta_level()* seja invocado; *(ii)* invocar os construtores das classes de meta-objeto e refletores; *(iii)* relacionar o objeto ao meta-objeto e este aos refletores usando o método *relate*, herdado da classe **Object** de Stabilis. Com isto, a instância criada será controlada por um meta-objeto diferente do especificado no método *create_meta_level()*.

5.4.2 Protocolo de execução de meta-objetos

O protocolo de execução de meta-objetos funciona como se segue. Quando para resolver uma consulta é necessário aplicar um método a algum objeto; antes da execução propriamente do método é verificado se o objeto possui um meta-objeto. Em caso afirmativo, a mensagem que constitui o método (assinatura, e parâmetros) são passados para o meta-objeto que cuida da ordem de execução do refletores e do próprio método do nível base. Se o objeto não possui meta-objeto então o método do nível base é invocado para o objeto. A mensagem passada para o metanível (assinatura e parâmetros do método do nível base) contém as informações necessárias para que o meta-objeto decida se os refletores devem ser ativados ou não. O método *handle_reflexive_message()* de **MetaObject** (Figura 5.8) pode ser sobrecarregado por subclasses e deve encapsular as ativações dos refletores e a execução do método do nível base. Ele é invocado automaticamente para o programador.

5.5 Visões e Reflexão

Como visto no Capítulo 4, visões em Stabilis são constituídas por um conjunto de classes que pertencem a um mesmo modelo estrutural. Além disso, todas as classes são descendentes de **Object**. Essa classe não possui representação no programa esquema que cria a visão pré-definida **Meta**, uma vez que não é necessária para descrever os modelos das aplicações.

RObject foi criada com o intuito de descrever objetos reflexivos. Ao contrário de **Object**, **RObject** precisa ser representada no programa esquema que cria a visão **Meta**. Isto porque objetos reflexivos mantém associações com meta-objetos, sendo que a representação das associações necessitam ser persistentes. Portanto, as classes **RObject**, **MetaObject** e

Reflector, que compõem o modelo chamado RStabilis, pertencem à visão Meta. Logo, para um determinado modelo de uma aplicação usar reflexão, ele deve indicar que o modelo RStabilis é seu submodelo. Com isto as classes RObject, MetaObject e Reflector sempre farão parte de visões que possuem objetos reflexivos, conforme representado na Figura 5.10.

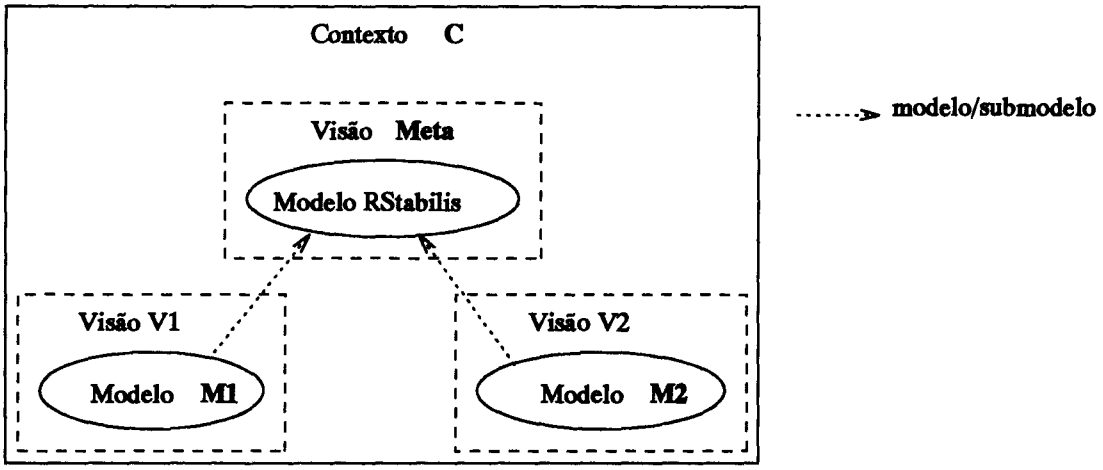

Figura 5.10: Visões e RStabilis

Classes refletoras e de meta-objetos fazem parte da mesma visão das classes do nível base. Isto ocorre porque uma visão está intimamente relacionada a um modelo da aplicação. Além disso, há casos em que isto é necessário. Por exemplo, quando meta-objetos também possuem meta-objetos (metameta-objetos) que são instâncias da mesma classe.

5.6 Herança e Reflexão em RStabilis

Em todos os possíveis modelos de reflexão para RStabilis apresentados neste capítulo não há definição explícita de como é o relacionamento entre as hierarquias do nível base e do metanível. De maneira suscinta: dadas uma metaclasse M1, usada para controlar objetos da classe A; uma metaclasse M2 usada para controlar objetos da classe B, sendo que B é subclasse de A. A questão é: M2 é subclasse de M1? Em Smalltalk [Gol83] isto sempre ocorre porque metaclasses são criadas automaticamente, garantindo assim esta relação. Em OpenC++ 2.0 [Chi96], isto também ocorre, devido a uma checagem que o sistema faz para garantir este relacionamento. Já em Open C++ 1.2 [CM93, Chi93], a metaclasse M2 poderá ser subclasse ou não de M1. Em RStabilis, ficou estabelecido que M2 não é necessariamente subclasse de M1, cabendo ao programador definir a relação de herança explicitamente, se o desejar. Preferiu-se essa abordagem por ser mais flexível e em

virtude do modelo adotado (objeto/meta-objeto/refletores). No modelo de objeto/meta-objeto/refletores, objetos da mesma classe podem ter meta-objetos e refletores diferentes, dependendo do protocolo de criação do metanível. Logo, não se pode garantir que os refletores de objetos diferentes do nível base possam ser ativados da mesma forma que o são para uma instância da superclasse. Caso esta relação de herança fosse garantida, M2 poderia ter várias superclasses (herança múltipla), já que instâncias da mesma classe podem ter meta-objetos diferentes. Além disso, uma metaclasse pode ser usada por mais de uma classe do nível base, caso esta não contenha informações sobre uma classe específica do nível base. Isto poderia ocasionar circularidade na herança, caso em que A e B possuissem uma metaclasse em comum.

5.7 Aplicação Exemplo: Otimização de Consulta

Esta seção apresenta um exemplo de utilização do protocolo de meta-objetos de RS-tabilis. As consultas aqui consideradas são aquelas que utilizam métodos. Além disso, os métodos realizam operações complexas tal que o "overhead" de tempo introduzido pelo metanível é insignificante em relação a execução do método. A otimização desejada é diminuir o número de execuções de métodos para obtenção do conjunto solução.

O metanível é composto pelas classes **MetaExecMethod**, **MaterialMethod** e **MethodResultObject**, conforme mostrado na Figura 5.11. A classe **MethodResultObject** é responsável pelo armazenamento das informações necessárias que permitam recuperar o objeto retornado como resultado de uma prévia execução de um método. **MaterialMethod** armazena os resultados de todos os métodos para um objeto. É responsável pela criação de instâncias da classe **MethodResultObject**. **MetaExecMethod**, subclasse de **MetaObject** implementa a ativação dos refletores e faz a invocação do método do nível base, se necessário. A classe A representa uma classe arbitrária de uma aplicação que deseja otimizar consultas como descrito anteriormente. O método *create_meta_level()* de A cria o meta-objeto e refletores e associa-os. Quando o método é invocado dentro de uma consulta, ele é interceptado e tem sua assinatura e parâmetros passados para o meta-objeto (instâncias de **MetaExecMethod**). A ativação dos refletores deve ser encapsulada no método *handle_reflexive_message()* de **MetaExecMethod**, chamado automaticamente após a interceptação do método invocado por uma instância da classe A. O meta-objeto ativa o refletor (instância de **MaterialMethod**) e invoca o método *get_MethodResultObject()* passando a assinatura do método do nível base. Esse método retorna para o meta-objeto o objeto resultante da execução do método, caso o método tenha sido executado anteriormente, ou um erro indicando que o método não foi executado para aquele objeto. Se o método nunca foi executado, o meta-objeto envia uma mensagem para o objeto do nível base para executá-lo e o resultado é passado para o refletor armazená-lo. Caso contrário, o meta-objeto simplesmente retorna

o objeto. Desta forma, métodos anteriormente invocados não são executados novamente, ocorrendo a otimização da consulta.

Cada instância da classe **MethodResultObject** armazena o resultado da execução de um único método para um único objeto do nível base. Cada instância da classe **MaterialMethod** controla o armazenamento de todos os métodos de um único objeto do nível base. A associação entre **MaterialMethod** e **MethodResultObject** (Figura 5.11) mostra como o modelo permite que cada objeto do nível base possa ter armazenados os resultados da execução de seus métodos. No apêndice B, encontra-se uma descrição mais detalhada de cada classe e respectivos métodos.

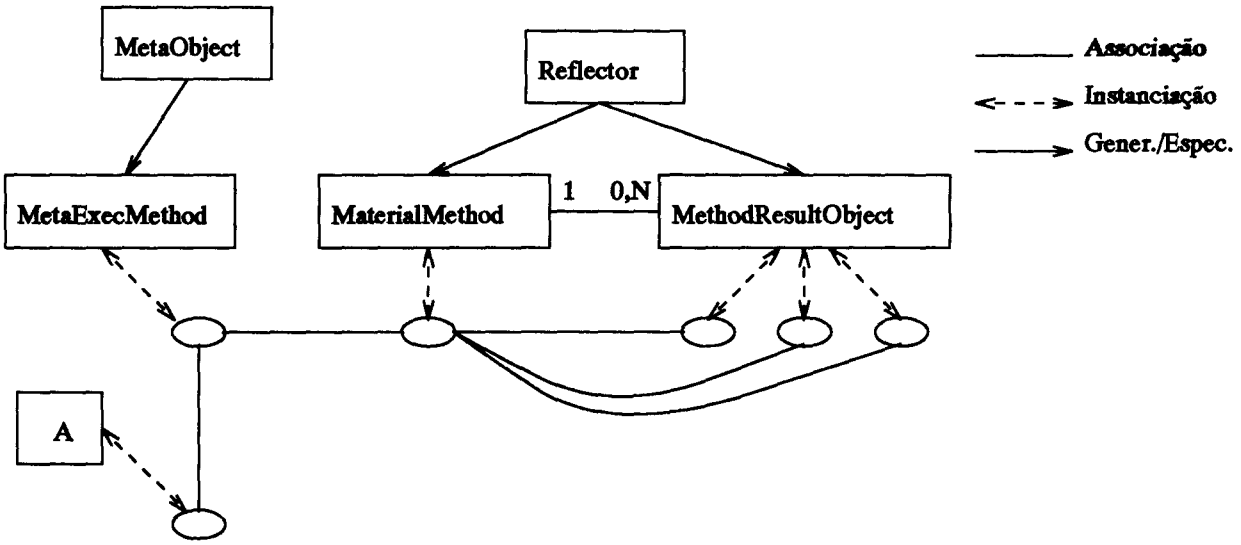

Figura 5.11: Modelo de objetos para Otimização de Consultas

É importante ressaltar que este esquema de otimização é válido somente se a execução dos métodos sempre retorna o mesmo resultado. Para os casos em que o resultado pode ser diferente, é necessário implementar no metanível um mecanismo que emita eventos para o meta-objeto quando ocorra alterações em informações que influem no resultado da execução do método. Esse mecanismo é equivalente aos existentes em sistemas ativos.

5.8 Resumo

Foram apresentados neste capítulo alguns modelos possíveis de reflexão computacional para Stabilis. Discutiu-se suas vantagens, desvantagens e a adoção de um deles para implementação. Os modelos apresentados foram: *(i)* classe/metaclasse, onde cada classe está associada a uma metaclasse sendo que essa possui o código dos metamétodos; *(ii)* classe/múltiplas metaclasses, onde cada classe pode estar associada a várias metaclas-

ses, embora apenas uma esteja ativa por vez; *(iii)* objeto/meta-objeto, onde há apenas relacionamento entre instâncias e a metaclasse deve implementar todo o protocolo de meta-objetos além da própria abstração a ser usada no nível base; *(iv)* objeto/meta-objeto/refletores que faz separação entre classes que implementam abstrações e classes que implementam protocolos de meta-objetos. Foram mostrados o projeto de um protocolo de meta-objetos para o modelo adotado (objeto/meta-objeto/refletores) e as questões relevantes no projeto de protocolos de meta-objetos (criação implícita X explícita). Por fim, demonstrou-se a utilização do modelo de reflexão e do protocolo de meta-objetos com um exemplo de otimização de consultas que fazem uso de invocações de métodos.

Capítulo 6

Conclusão

Este trabalho foi concentrado em projetar e implementar um modelo de reflexão computacional e um protocolo de meta-objetos para uma máquina de busca de objetos que permitam o programador separar as atividades internas e externas de sua aplicação, aumentando sua flexibilidade. Neste capítulo são apresentadas as conclusões obtidas dos experimentos e pesquisas feitas, e em seguida são sugeridos trabalhos futuros, levantados durante estes experimentos, que visam o desenvolvimento de aplicações mais flexíveis.

No capítulo 2 foram mostrados os principais conceitos envolvidos no desenvolvimento de componentes de software e da técnica de orientação a objetos. Foram apresentados também os conceitos básicos de tolerância a falhas, sua importância para construção de sistemas confiáveis; e os conceitos básicos de reflexão computacional, sua utilidade e uma classificação para os modelos. No capítulo 3 discutiu-se alguns trabalhos relacionados à reflexão computacional e utilização de métodos em consultas de banco de dados orientados a objetos. Foram mostrados os modelos de reflexão de algumas linguagens de programação, comparando-os quanto a sua funcionalidade e facilidade de uso. No capítulo 4 descreveu-se Stabilis, o ambiente alvo para o qual foi projetado e implementado um modelo de reflexão computacional. Discutiu-se as implicações do uso de métodos em consultas para o modelo de objetos de Stabilis. No capítulo 5 avaliou-se alguns modelos de reflexão possíveis para o ambiente alvo; escolheu-se um e definiu-se um protocolo de meta-objetos. Foi demonstrada a utilização do modelo/protocolo de meta-objetos por uma aplicação que otimiza a resolução de consultas. Estão assinaladas a seguir as principais contribuições deste trabalho.

6.1 Contribuições

As principais contribuições deste trabalho são:

1. A apresentação dos principais conceitos e técnicas usados na construção de componentes de software.

   Orientação a objetos tem se mostrado uma técnica efetiva para a construção de componentes de software, graças à abstração provida pelo conceito de objeto. A seu lado outras técnicas vêm sendo utilizadas tais como tolerância a falhas e reflexão computacional. Tolerância a falhas consiste de um conjunto de técnicas que visa aumentar a confiabilidade dos componentes e, portanto, do sistema. Reflexão computacional surge como uma técnica que permite melhorar a organização interna do sistema. No capítulo 2, foram apresentados os fundamentos dessas três técnicas.

2. Um "survey" de sistemas reflexivos e a identificação de questões de projeto dos modelos de reflexão computacional que determinam uma maior ou menor flexibilidade da aplicação.

   Nessa abordagem procurou-se separar o conceito de modelo de reflexão do conceito de protocolo de meta-objetos. Um modelo de reflexão computacional é considerado como uma hierarquia de classes que permite a meta-objetos controlarem a execução de objetos. Um protocolo de meta-objetos é um conjunto de regras que determinam como o metanível é criado e o modo pelo qual ele controla o nível base. Apesar de estarem relacionados, pode-se ter para um mesmo modelo de reflexão, protocolos de criação do metanível diferentes. No que tange aos modelos, no capítulo 3 foram apresentados vários sistemas reflexivos, questões importantes como: o espelhamento da hierarquia do metanível na hierarquia de herança, cardinalidade da relação objeto/meta-objeto e número de metaníveis. No capítulo 5 discutiu-se possíveis modelos de reflexão e protocolos de meta-objetos para a máquina de busca de objetos Stabilis, avaliando as implicações da adoção de um deles para o desenvolvimento e flexibilidade das aplicações.

3. Implementação de chamadas de métodos em consultas.

   A linguagem de consulta original de Stabilis não reconhecia invocações de métodos. A linguagem de consulta foi estendida para permitir a utilização de métodos segundo a semântica de mapeamento, apresentada no Capítulo 4. Também foram avaliadas questões pertinentes ao uso de métodos em consultas, tais como: efeitos colaterais, terminação de execução e concordância entre o resultado de consultas e o modelo de objetos.

4. Projeto e implementação de um modelo de reflexão computacional e protocolo de meta-objetos.

   Diante das questões levantadas no capítulo 3, quanto aos modelos de reflexão, foi projetado um modelo, apresentado no capítulo 5, visando dar uma flexibilidade

maior à aplicação, procurando manter a transparência. O modelo proposto permite ao metanível ser estruturado de tal forma que classes que implementam funcionalidades ortogonais à aplicação possam ser reutilizadas. Isto é possível porque o controle dos objetos do nível base é feito pelos meta-objetos, isto é, decidem quando ativar os refletores e em que ordem. Os refletores encapsulam o que deve ser executado, porém não são capazes de determinar sua própria ordem no fluxo de execução. No capítulo 5, também discutiu-se sobre a criação de meta-objetos de forma transparente ou não e mostrou-se que ambas formas podem ser implementadas sobre o mesmo modelo, embora apresentem conseqüências diferentes para o desenvolvimento de aplicações. O uso do modelo e do protocolo de meta-objetos foi demonstrado com uma aplicação que otimiza consultas.

Das contribuições expostas anteriormente, pode-se concluir que aplicações desenvolvidas usando RStabilis são mais flexíveis que quando desenvolvidas usando Stabilis. Além disso, possuem um custo adicional de desenvolvimento relativamente pequeno comparado à facilidade de estruturação oferecida pelo modelo de reflexão computacional. Espera-se que os resultados deste trabalho possam ser utilizados não apenas em trabalhos futuros que visem melhorar a atual versão de RStabilis, mas também em outros sistemas de desenvolvimento de software.

6.2 Trabalhos Futuros

Durante a realização da pesquisa e implementação do modelo de reflexão foram identificadas: alterações arquiteturais que facilitariam nossa implementação e possível utilização dos resultados para desenvolvimento de outros trabalhos. Está listado a seguir os principais trabalhos futuros:

1. Implementação da semântica de seleção, apresentada no capítulo 4, para chamada de métodos. Dentro desta semântica, o resultado do método é usado para determinar se o objeto sobre o qual o método foi aplicado satisfaz determinado predicado. Portanto, o resultado do método comporta-se como um atributo. A implementação da semântica de seleção exige a existência de um índice por classe, que indexe todas as instâncias diretas e indiretas. Na versão atual de Stabilis tal índice não se encontra implementado.

2. Tratamento de tipos simples C++ como objetos Stabilis. Métodos que retornam tipos simples C++ não podem ser invocados dentro de consultas — pois o seu retorno não são objetos — para manter a concordância com o modelo de objetos. Tratando tipos simples C++ como objetos, a linguagem de consulta mantém a uniformidade e a concordância com o modelo de objetos.

3. Remoção de objetos. A implementação de remoção de objetos é razoavelmente simples, porém exigindo certos cuidados para manutenção da consistência do banco de dados. Ao se remover um objeto, deve-se remover igualmente todos os seus relacionamentos com outros objetos incluindo os objetos componentes que se relacionam por meio de agregação forte com o objeto. Isto é equivalente à integridade referencial de banco de dados relacionais. Quanto à reflexão, o metanível do objeto — meta-objeto e refletores — deve ser removido igualmente, pois refere-se especificamente ao objeto removido do nível base.

4. Disponibilização de métodos utilitários. Stabilis provê um programa chamado *parla* que permite ao usuário formular consultas que criam ou recuperam objetos que descrevem as classes de sua aplicação de maneira interativa. Com ele o usuário pode obter informações como: os atributos de determinada classe, seus métodos e relacionamentos. Como na nova versão consultas reconhecem invocações de métodos, então pode-se disponibilizar para o *parla* métodos utilitários implementados pelas classes de Stabilis, como por exemplo o método *gen_code* da classe **Model**. Com isto o usuário não precisa mais escrever e compilar programas esquema, pois os objetos que representam classes podem ser definidos interativamente através de *parla* e além disso, o usuário pode invocar o método *gen_code* para gerar o código de sua aplicação.

5. Controle de versões e evolução de esquema. Aplicações como sistemas de informações geográficas, CAD e CASE necessitam do uso de versões e evolução de esquema para refletir a realidade que estas aplicações se propõem. Versões são históricos do banco de dados, mantidas para mostrar a evolução da aplicação. Sistemas que suportam evolução de esquema permitem que definições de classes sejam alteradas, provendo assim uma grande flexibilidade às aplicações.

Apêndice A

Processo de Resolução de Consultas em Stabilis

Neste apêndice é apresentado em detalhes o processo de resolução de uma consulta em Stabilis. O programador deve escolher entre recuperar apenas um ou vários objetos que atendam às condições expressas na consulta. Para recuperar vários objetos ele deve criar instâncias da classe ObjectSet que encapsula um conjunto de objetos ("template" Set<Object>) enquanto que para recuperar apenas um objeto deve utilizar o construtor da própria classe do objeto. O processo de resolução de consulta é o mesmo para ambos os casos, sendo que a recuperação de um único objeto é feita escolhendo-se o primeiro objeto do "ObjectSet" obtido. Dessa forma, serão consideradas neste apêndice, para efeito de simplificação, apenas consultas que retornam um único objeto, ou seja, que após a obtenção do conjunto de objetos que satisfazem às condições expressas na consulta, será retornado como solução o primeiro elemento.

Uma consulta deve ter um modo — *Birth*, *Reincarnation* ou *Provide* — que determina como a expressão será interpretada. O modo *Birth* indica que o resultado da consulta será um novo objeto, enquanto *Reincarnation* indica que um objeto já existente será recuperado.*Provide* funciona como *Reincarnation* inicialmente. Porém, se nenhum objeto satisfazendo as condições expressas na consulta for encontrado então a consulta será tratada como uma consulta de modo *Birth*.

Uma expressão de consulta é passada no formato de "string" como parâmetro para o construtor da classe do objeto a ser criado ou recuperado. Os construtores são gerados automaticamente por Stabilis, e sempre invocam os construtores das superclasses passando a expressão de consulta recebida. Como todas as classes são subclasses diretas ou indiretas da classe Object de Stabilis, então a expressão de consulta é sempre um parâmetro para o construtor de Object. Essa classe possui um atributo do tipo Expression*, cujo propósito é encapsular o "parse" da expressão. De acordo com o modo de

consulta (*Birth*, *Reincarnation* ou *Provide*) é invocado um método correspondente sobre Expression*, que utilizando a árvore de "parse" construída faz a criação ou recuperação do objeto. A Figura A.1 mostra a hierarquia de classes utilizada para a construção da árvore de "parse". A classe Expression possui uma referência para Expr, encapsula todo o "parse" usando a função *yyparse()* de **Bison**[1] e verifica o "status" retornado. Expr é uma classe abstrata que representa as variedades de expressões que podem ser usadas na consulta, como por exemplo, expressões que utilizam união, interseção ou métodos, representadas pelas classes UnionOp, IntersectionOp e IterFilterOp respectivamente. ClassOp representa a expressão mais simples possível, constituída apenas por expressões de atributos. Como demonstra a Figura A.1. A classe IterFilterOp possui uma referência para ClassOp e uma lista de referências para MethodOp. A referência para ClassOp é destinada à recuperação dos objetos sobre os quais o primeiro método (MethodOp) da lista será aplicado. O resultado obtido é aplicado ao método seguinte e assim por diante.

Uma consulta simples — sem operadores de união e interseção — pode ser esquematizada da seguinte forma:

$E_1.mth_1(E_{11}, E_{12}, ..., E_{1n_1}).mth_2(E_{21}, E_{22}, ..., E_{2n_2}).....mth_k(E_{k1}, E_{k2}, ..., E_{kn_k})$

onde:

- E_1 é uma expressão da forma $< classe > (< expr - de - atributo >)$, que especifica objetos sobre os quais o método mth_1 será aplicado.
- mth_1, mth_2, ... , mth_k são os métodos a serem aplicados em seqüência.
- E_{r1}, E_{r2}, ..., E_{rn} com $r = 1, ..., k$ são expressões da mesma forma que E_1 e usadas para recuperar os objetos que serão parâmetros do método mth_r.

À exceção de invocação de métodos, na resolução de consultas não são utilizados objetos diretamente, pois além dos atributos da própria classe, os objetos também herdam todos os atributos das superclasses, o que tornaria o processamento muito lento. Na verdade são utilizadas instâncias das classes Pip e Pips. A classe Pip tem como atributos apenas as informações essenciais para localização e recuperação dos objetos. Esses atributos são: o nome da classe e UID do objeto além do nó que contém o objeto. Pips é um conjunto de Pip, tendo métodos usuais em conjuntos como união, interseção e inclusão de novos elementos. Quando é necessário executar um método, o objeto é recuperado e então feita a aplicação do método. Após a obtenção do resultado, que também é um objeto, é construído um "pip" para identificá-lo.

Para elucidar melhor a execução de métodos, considere as classes Estudante, Livro e Emprestimo, cujos objetos estão nas tabelas A.1, A.2 e A.3, respectivamente. Considere também a consulta:

[1] Implementação de *yacc* de Free Software Foundation, Inc — projeto GNU

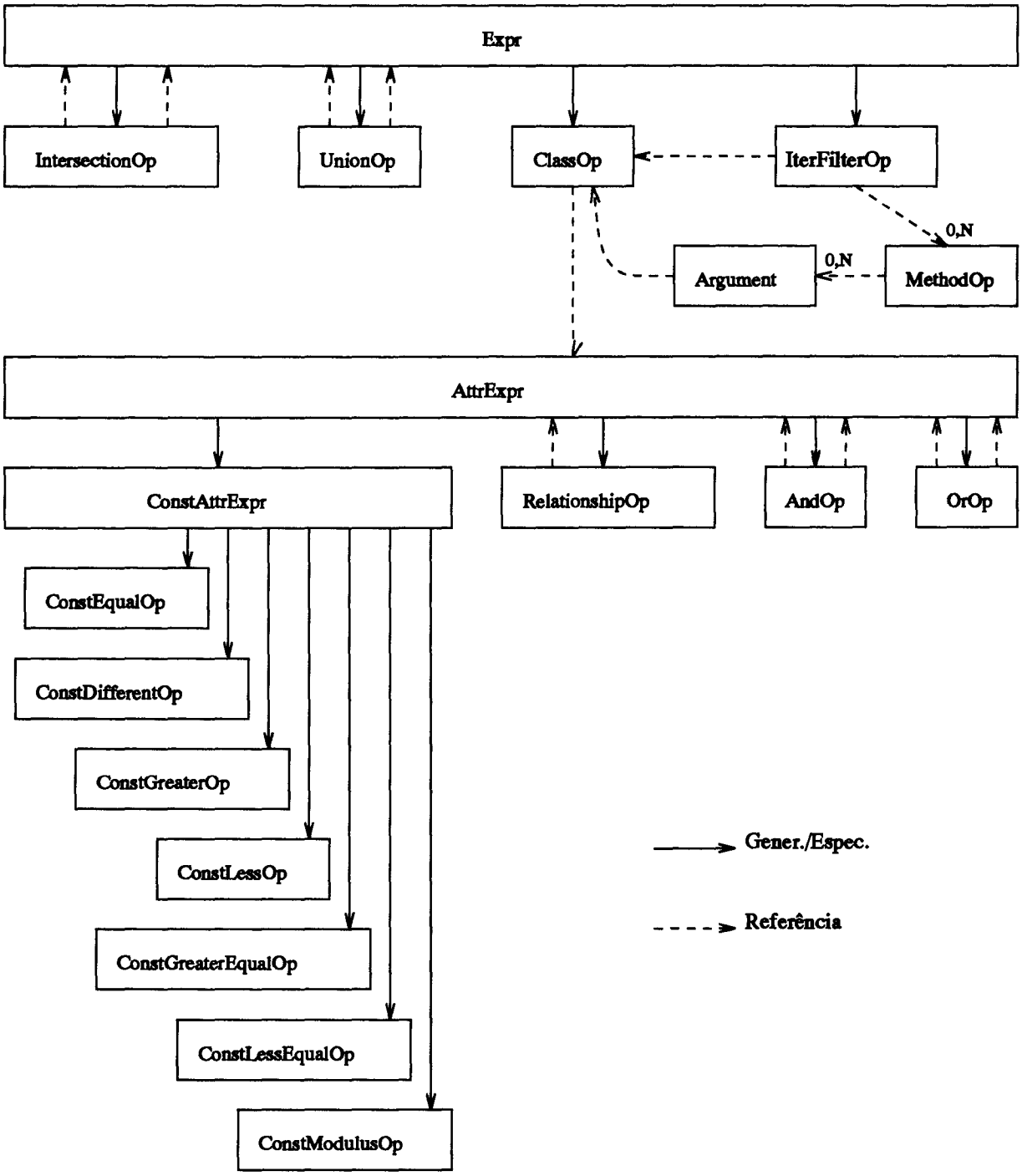

Figura A.1: Hierarquia de classes para resolução de consultas

Emprestimo emp("Estudante(escola=='E.E.Z.M'). get_emprestimo(Livro(editora== 'Circulo do Livro'))", REINCARNATION, oph);

Estudante	*nome*	*idade*	*ra*	*escola*
e_1	Antônio	17	973542	E.E.Z.M
e_2	Daniela	19	973872	E.E.Z.M
e_3	Robson	22	963935	E.E.P.M.B
e_4	Gabriela	20	972758	E.E.T.
e_5	Luiza	23	948629	E.E.Z.M

Tabela A.1: Instâncias da classe Estudante

Livro	*titulo*	*autor*	*editora*
l_1	Dom Casmurro	Machado de Assis	Globo
l_2	O Mundo de Sofia	Jostein Gaarder	Círculo do Livro
l_3	O Alienista	Machado de Assis	Globo
l_4	O Discurso do Método	Rene Descartes	Círculo do Livro

Tabela A.2: Instâncias da classe Livro

Emprestimo	*usuario*	*livro*	*data_emp*	*data_dev*
em_1	Daniela	O Alienista	20/02/07	27/02/97
em_2	Luiza	O Mundo de Sofia	10/02/97	17/02/97
em_3	Antônio	Dom Casmurro	13/02/97	20/02/97
em_4	Luiza	O Discurso do Método	15/02/97	22/02/97

Tabela A.3: Instâncias da classe Emprestimo

Essa consulta recupera um empréstimo que um estudante da escola "E.E.Z.M." fez de um livro da editora "Circulo do Livro". A árvore de "parse" simplificada equivalente a essa consulta está mostrada na Figura A.2. Os objetos resultantes das consultas parciais de *Estudante(escola=='E.E.Z.M')* e de *Livro(editora =='Circulo do Livro')* são mostrados na Tabelas A.4 e A.5, respectivamente. Observa-se que estes resultados são equivalentes a seleções no modelo relacional. O método *get_emprestimo*, cujo código está

na Figura A.3, recupera uma instância da classe Emprestimo cujo usuário seja o nome do próprio estudante, e livro o mesmo passado como parâmetro. Métodos podem realizar operações mais complexas, sendo o exemplo aqui apresentado bastante simples para facilitar a compreensão de como métodos e consultas estão relacionados.

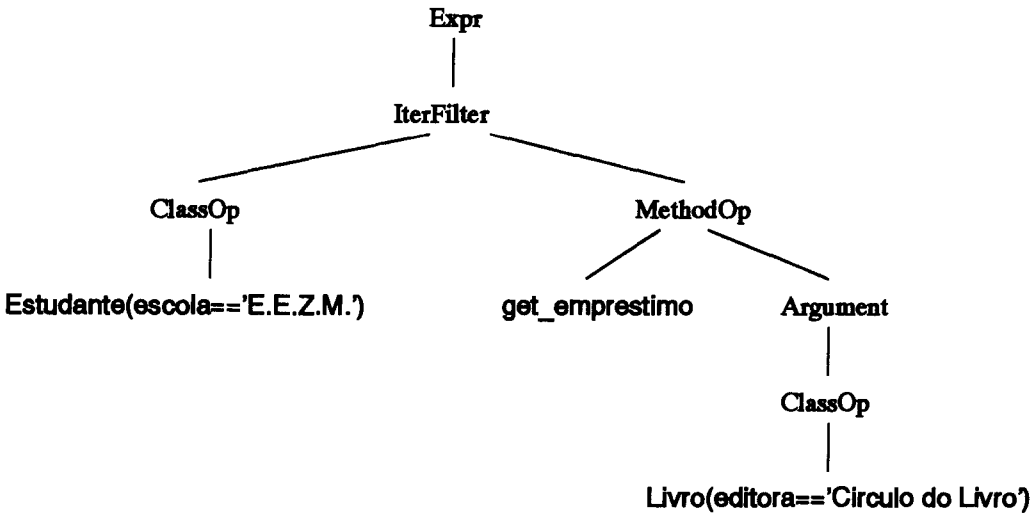

Figura A.2: Árvore de "parse"

Estudante	*nome*	*idade*	*ra*	*escola*
e_1	Antônio	17	973542	E.E.Z.M
e_2	Daniela	19	973872	E.E.Z.M
e_5	Luiza	23	948629	E.E.Z.M

Tabela A.4: Resultado da consulta parcial *"Estudante(escola=='E.E.Z.M')"*

Livro	*titulo*	*autor*	*editora*
l_2	O Mundo de Sofia	Jostein Gaarder	Círculo do Livro
l_4	O Discurso do Método	Rene Descartes	Círculo do Livro

Tabela A.5: Resultado da consulta parcial *"Livro(editora=='Circulo do Livro')"*

O algoritmo de resolução de consulta para esse exemplo toma cada objeto resultante da consulta parcial *Estudante(escola=='E.E.Z.M')* — isto é: e_1, e_2 e e_5 — e aplica o

```
Emprestimo* Estudante::get_emprestimo(Livro* lv, OpHistory* oph)
{
  Emprestimo* emp = new Emprestimo("Emprestimo(livro==' " + lv→get_titulo()
+ "' && usuario == '" + this→get_nome() + "')", Object::view,
REINCARNATION, oph);
  return emp;
}
```

Figura A.3: Código do método *get_emprestimo()*

método *get_emprestimo* passando como parâmetro cada um dos objetos obtidos pela consulta parcial *Livro(editora=='Circulo do Livro')* — isto é: l_2 e l_4. Portanto, o método *get_emprestimo* é executado seis vezes. Quando o método retorna um objeto válido, ele entra para o conjunto solução, caso contrário é descartado. Um objeto é válido se existe uma cópia persistente do mesmo. A classe OpHistory tem como propósito armazenar o "status" do objeto retornado, isto é, válido ou não. A Figura A.6 mostra o resultado parcial da consulta após a aplicação do método *get_emprestimo*.

Emprestimo	*usuario*	*livro*	*data_emp*	*data_dev*
em_2	Luiza	O Mundo de Sofia	10/02/97	17/02/97
em_4	Luiza	O Discurso do Método	15/02/97	22/02/97

Tabela A.6: Resultado após a aplicação do método *get_emprestimo()*

O conjunto solução final é então formado apenas pelo objeto em_2, já que a consulta foi passada para o construtor da classe Emprestimo, e não de ObjectSet, conforme explicado anteriormente.

Apêndice B

Uso de RStabilis

Neste apêndice é mostrado mais detalhadamente o exemplo de utilização do modelo de reflexão computacional RStabilis para otimização de consultas, que fazem uso de métodos, apresentado no Capítulo 5.

A Figura B.1 mostra as interfaces das classes MaterialMethod e MethodResultObject, ambas subclasses de Reflector.O atributo *base_obj_uid* de MaterialMethod armazena na forma de uma string a identificação única do objeto do nível base, usado como atributo-chave. O atributo *mth_name* de MethodResultObject contém a assinatura do método do objeto do nível base. Os atributos *uid_obj_res*, *class_name_res* e *host_res* permitem a recuperação do objeto resultante da execução do método.

A Figura B.2 mostra a implementação do método *handle_reflexive_message()* de MetaExecMethod, chamado automaticamente quando um método do nível base é invocado dentro de uma consulta. Esse método recupera o refletor do objeto do nível base e ativa-o, enviando uma mensagem para ele executar o método *get_MethodResultObjet()*, que tenta recuperar o objeto resultante de uma execução anterior do método do nível base. Se o método já foi executado anteriormente, então objeto resultante é retornado para o nível base. Caso o método não tenha sido executado antes, o meta-objeto então envia uma mensagem para o objeto do nível base executá-lo. Em seguida, o refletor é ativado novamente para executar o método *create_MethodResultObject()* que recebe como parâmetros a assinatura do método e o objeto resultante.

Esse método cria uma instância da classe MethodResultObject e associa-se a ela. O método *get_MethodResultObject()* recebe como parâmetro a assinatura do método do nível base e recupera o objeto resultante da execução do método especificado na assinatura sobre o objeto do nível base.

Uma instância da classe OpHistory é passada também como parâmetro para todos os métodos com o objetivo de armazenar o "status" retornado pela execução do método. Como pode-se observar, essa classe não consta na assinatura do método. Isto deve-se

```
class MaterialMethod: public Reflector {

protected:
  String base_obj_uid;                                          // chave

public:
  RObject* create_MethodResultObject(String mth_name, RObject* obj, OpHistory*
oph);
  RObject* get_MethodResultObject(String meth_name, OpHistory* oph);

}

class MethodResultObject: public Reflector {

protected:
  String mth_name;                                              // chave
  String uid_obj_res;
  String class_name_res;
  String host_res;
}
```

Figura B.1: Interface de Materialmethod e MethodResultObject

```
OpHistory* MetaExecMethod::handle_reflexive_method(String signature,
                                                    List<Argument> * args,
                                                    Result*& result_obj)
{
  OpHistory* oph = new OpHistory;
  RObject base_obj;
  *oph += get_base_level_object(base_obj);
  *oph += get_before_reflectors();
  Reflector* reflec = before_reflectors→first();
  List<Argument>* args_reflec = new List<Argument>;
  args_reflec→insert(new ValueParameter(signature));
  *oph += reflec→handle_message("get_MethodResultObject(String)", args_reflec,
res_obj);
  if(res_obj→get_oph()→abnormal())                    // metodo nao executado
    {
      *oph += base_obj→handle_message(signature, args, result_obj);

      args_reflec→insert(new PipParameter(result_obj→get_object_pip()));
      *oph += reflec→handle_message("create_MethodResultObject(String,
RObject*)",args_reflec);
    }
  else result_obj = res_obj;
  return oph;
}
```

Figura B.2: Implementação do método *handle_reflexive_message()*

ao fato de que a classe OpHistory não faz parte do metamodelo de Stabilis, isto é, suas instâncias não são persistentes. Uma vez estando a assinatura dos métodos definidas no programa esquema, RStabilis gera o código do método *handle_message()* automaticamente, acrescentando a eles a classe OpHistory. As Figuras B.3, B.4 e B.5 mostram a implementação dos métodos *create_MethodResultObject()*, *get_MethodResultObject()* e *handle_message()* da classe MaterialMethod, respectivamente.

```
RObject* MaterialMethod::create_MethodResultObject(String mth_name, RObject* res,
OpHistory* oph)
{
  Pip res_pip = res→get_object_pip();
  String class_name_res = res_pip→get_class_name();
  String uid_obj_res = res_pip→get_uid();
  Stirng host_res = res_pip→get_host();
  MethodResultObject* mth_res_obj = new
MethodResultObject("MethodResultObject(mth_name=" + mth_name + " &&
class_name_res= " + class_name_res + "uid_obj_res= " + uid_obj_res + " &&
host_res= " + host_res, BIRTH,view,oph);
  this→relate("MethodResultObject", mth_res_obj);
}
```

Figura B.3: Método *create_MethodResultObject()*

```
RObject* MaterialMethod::get_MethodResultObject(String mth_name, OpHistory* oph)
{
  MethodResultObject* mth_res_obj = new
MethodResultObject("MethodResultObject(MaterialMethod::base_obj_uid==" +
base_obj_uid + " && mth_name==" + mth_name, REINCARNATION, view, oph);

 RObject res_obj = new
RObject(mth_res_obj→get_pip_data(),REINCARNATION,view, oph);
 return res_obj;
}
```

Figura B.4: Método *get_MethodResultObject()*

```
OpHistory* MaterialMethod::handle_message(String signature, List<Argument>*
arguments_list, Result*& reply_object)
{
  OpHistory* oph = new OpHistory;
  if ( signature == "create_MethodResultObject(String,RObject*)" )
    {
      String p1;
      Value<String,1> p_aux1;
      *oph += arguments_list->retrieve()->get_value(p_aux1, Object::view);
      p1 = p_aux1.get_value();
      arguments_list->forward();
      Object* p2_RObject;
      *oph += arguments_list->retrieve()->get_value(p2_RObject, Object::view);
      RObject p2 = (RObject*) p2_RObject;
      arguments_list->forward();

      OpHistory* oph_res = new OpHistory;
      reply_object = new Result(create_MethodResultObject(p1,p2,oph_res));
      if(oph_res->normal())
        {
          reply_object->set_oph(oph_res);
        }
    }
  else
    if(signature == "get_MethodResultObject(String)")
      {
        String p1;
        Value<String,1> p_aux1;
        *oph += arguments_list->retrieve()->get_value(p_aux1, Object::view);
        p1 = p_aux1.get_value();
        arguments_list->forward();
        OpHistory* oph_res = new OpHistory;
        reply_object = new Result(create_MethodResultObject(p1,oph_res));
        if(oph_res->normal())
          {
            reply_object->set_oph(oph_res);
          }
      }
  else *oph += Reflector::handle_message(signature, arguments_list, reply_object);
  return oph;
}
```

Figura B.5: Método *handle_message()*

Bibliografia

[Bed97] Delano Medeiros Beder. Integração dos Mecanismos de Recuperação de Erros por Avanço e por Retrocesso. Dissertação de Mestrado, Unicamp, 1997.

[BHS] G. Blair, D. Hutchison, e D. Shepherd, editores. *Object-Oriented Languages, Systems and Applications.*

[BNPS92] Elisa Bertino, Mauro Negri, Giuseppe Pelagatti, e Licia Sbattella. Object-Oriented Query Languages: The Notion and the Issues. *IEEE Trasactions on Knowlegde and Data Engineering*, 4(3):223–237, 1992.

[Buz94] L. E. Buzato. *Management of Object-Oriented Action-Based Distributed Programs.* PhD thesis, University of NewCastle upon Tyne, 1994.

[Cal96] Alcides Calsavara. *Constructing Highly-Available Distributed Metainformation Systems.* PhD thesis, University of NewCastle upon Tyne, 1996.

[Chi93] Shigeru Chiba. Open C++ 1.2 - Programmer's Guide. Technical Report 93-3, Department of Information Science, University of Tokyo, 1993.

[Chi96] Shigeru Chiba. Open C++ 2.0 Programmer's Guide. Technical report, University of Tokyo, 1996.

[CM93] Shigeru Chiba e Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. *ECOOP'93*, pp. 482–501, julho de 1993.

[Coi93] Pierre Cointe. CLOS and Smalltalk. In Andreas Paepcke, editor, *Object-Oriented Programming - The CLOS Perspective*, capítulo 9, pp. 215–249. The MIT Press, 1993.

[DeM93] Linda G. DeMichiel. An introduction to CLOS. In Andreas Paepcke, editor, *Object-Oriented Programming - The CLOS Perspective*, capítulo 1, pp. 3–27. The MIT Press, 1993.

[dHF93] Aurélio Buarque de Holanda Ferreira. *Minidicionário da Língua Portuguesa.* 1993.

[Fer89] Jacques Ferber. Computational Reflection in Class based Object Oriented Languages. *OOPSLA'89*, 24(10):317–326, 1989.

[Gaa91] Jostein Gaarder. *O Mundo de Sofia.* Círculo do Livro LTDA, 1991.

[GH93] Jutta Göers e Andreas Heuer. Definition and Application of Metaclasses in an Object-Oriented Database Model. *IEEE*, 1993.

[Gol83] A. Robson Goldberg. Smalltalk-80 - The Language and its Implementation, 1983.

[Jal94] P. Jalote. *Fault Tolerance in Distributed Systems*, capítulo 1, pp. 01–44. Prentice-Hall, Inc., 1994.

[KdRB92] Gregor Kiczales, Jim des Rivières, e D. G. Bobrow. The Art of the Metaobject Protocol, 1992.

[LAR] M. R. Stonebraker L. A. Rowe. The POSTGRES Data Model. *Relational Extensions and Extensible Databases.*

[LRV92] C. Lécluse, P. Richard, e F. Vélez. *Building an Object-Oriented Database System: The Story of O2*, capítulo 4, pp. 77–97. Morgan Kaufmann Publishers, Inc., 1992.

[Mae87] Patie Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA'87 Proceedings*, pp. 147–155, outubro de 1987.

[MC93] Philippe Mulet e Pierre Cointe. Definition of a Reflective Kernel for a Prototype-Based Language. *LNCS 742, Object Technologies for Advanced Software*, 93.

[MNC+91] G. Masini, A. Napoli, D. Colnet, D. Leonard, e Karl Tombre. *Object-Oriented Languages*, volume 34 de *The APIC Series.* Academic Press Inc., 1991.

[MWY91] Satoshi Matsuoka, Takuo Watanabe, e Akinori Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In Pierre America, editor, *Lecture Notes Computer Science 512*, julho de 1991.

[Pae93] Andreas Paepcke. User-Level Language Crafting: Introducing the CLOS Metaobject Protocol. In Andreas Paepcke, editor, *Object-oriented Programming - The CLOS Perspective*, capítulo 3, pp. 65–99. The MIT Press, 1993.

[Smi85] Brian C. Smith. Prologue to "Reflection and Semantics in a Procedural Language". In Morgan Kaufmann, editor, *The Knowledge Representation Enterprise*, pp. 31–39. R. J. Brachman and H. S. Levisque, 1985.

[SP91] S. K. Shrivastava e G. D. Parrington. An Overview of Arjuna: A Programming System for Reliable Distributed Computing. *IEEE Software*, 8(1):63–73, 91.

[Tan] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall.

[TT95] T. Takahashi e M. Takeda. An Efficient Reflective Architecture for Parallel Logic Programming Languages. *Proceedings of ICLP'95 - Workshop on Parallel Logic Programming Language*, 95.

[Weg90] Peter Wegner. *OOPS Messenger - Concepts and Paradigms of Object-Oriented Programming.* ACM SIPLAN Messenger. ACM PRESS, 1990.

[Yok92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. *OOPSLA'92*, pp. 414–434, 1992.