WAGNER SALAZAR PIRES

UMA LINGUAGEM DE ESPECIFICAÇÃO FORMAL ORIENTADA POR ASPECTOS

Belo Horizonte
29 de junho de 2007

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

UMA LINGUAGEM DE ESPECIFICAÇÃO FORMAL ORIENTADA POR ASPECTOS

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

WAGNER SALAZAR PIRES

Belo Horizonte
29 de junho de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Uma Linguagem de Especificação Formal Orientada por Aspectos

WAGNER SALAZAR PIRES

Dissertação defendida e aprovada pela banca examinadora constituída por:

Prof. Roberto da Silva Bigonha – Orientador
Departamento de Ciência da Computação - UFMG

Profa. Mariza Andrade da Silva Bigonha
Departamento de Ciência da Computação - UFMG

Prof. Vladimir Oliveira Di Iorio
Departamento de Informática - UFV

Profa. Elaine Gouvêa Pimentel
Departamento de Matemática - UFMG

Belo Horizonte, 29 de junho de 2007

Deus nos dá pessoas e coisas para aprendermos a alegria ...
Depois, retoma coisas e pessoas para ver se já somos
capazes da alegria sozinha.. Saudades Papai!
(Guimarães Rosa - Buriti)

Agradecimentos

Agradecer sempre é difícil: sempre se corre o risco de esquecer alguém. Assim, vou agradecer, primeiramente, a todo mundo, que se acabar não citando alguma pessoa que me ajudou, seja direta ou indiretamente, ela não estará esquecido de todo. Em primeiro lugar, gostaria de agradecer ao professor Bigonha a orientação, a paciência, as piadinhas e os ensinamentos que sem dúvida foram fundamentais para a realização deste trabalho.

Agradeço aos meus pais, Tarço e Maria Irene, que sempre estiveram incondicionalmente ao meu lado e me mostraram o caminho. Agradeço também a minha irmã, Denise, e ao meu cunhado, Daniel, que sempre me lembraram das minhas responsabilidades nos períodos de farra.

Agradeço aos amigos do LLP pelo ambiente descontraído de trabalho, principalmente ao Leo (*o tosko*). Não poderia deixar de mencionar o Italo Giovani, por ter proposto inúmeras construções para a linguagem Machĭna aumentando o meu trabalho.

Finalmente, agradeço a todos os meus amigos mais próximos que, de uma forma ou de outra, seja com cerveja ou mulheres, proporcionaram momentos de distração e contribuíram para a realização deste trabalho. A todos vocês, a minha mais sincera gratidão.

Resumo

Máquinas de Estado Abstratas oferecem um mecanismo poderoso e de fácil utilização para especificação formal da semântica de algoritmos. A Linguagem Aspect$\mathcal{M}$ incrementa esta metodologia com a capacidade de modularizar interesses transversais. Aspect$\mathcal{M}$ é uma linguagem de especificação formal orientada por aspectos que reúne as vantagens existentes em uma especificação formal, como a descrição precisa dos requisitos do sistema, a partir da qual podem ser realizadas verificações, e as vantagens da programação orientada por aspectos, como a modularização de interesses transversais.

Abstract

The Abstract State Machines methodology offers a powerful, easy-to-use mechanism to formally specify the semantics of algorithms. The Aspect$\mathcal{M}$ language adds to it the modularized crosscutting concern capability. Aspect$\mathcal{M}$ is an aspect oriented formal specification language that unifies the well known benefits of formal specification, such as rigorous requirement description, from which verification and validation can be carried out, and the improved modularity provided by aspect oriented programming, such as separation of concerns.

Sumário

Capítulo 1

Introdução

Especificações e métodos formais têm sido aplicados no intuito de dar uma base matemática ao desenvolvimento de sistemas. Esses métodos utilizam modelos matemáticos que fazem uso de sintaxe e semântica bem definidas. Assim, técnicas de verificação formal podem ser aplicadas para garantir a correção de determinadas propriedades do software no processo de desenvolvimento.

Métodos formais podem ser aplicados em qualquer etapa do ciclo de desenvolvimento do software, tanto no processo de especificação, para verificar se o software atende aos seus requisitos, como no processo de implementação, para concluir acerca da correção da implementação em relação à especificação. Desta forma, a solução de um problema pode ser representada por uma especificação formal, que pode ser validada, ainda que parcialmente por meio de sua execução (simulação), ou ainda ser analisada e verificada formalmente, provando-se propriedades do sistema antes mesmo de sua implementação.

Os métodos formais não se limitam somente a sistemas de software, mas a qualquer sistema, pois eles proporcionam a construção e análise de modelos matemáticos que representam a realidade. Assim, o comportamento de implementações de sistemas físicos podem ser previamente conhecidos e validados.

Métodos informais de definição de linguagens de programação carregam uma imprecisão semântica que pode resultar em descrições inconsistentes ou ambíguas, sendo, por isso, inadequados para descreverem precisamente linguagens de programação. Métodos formais são fortes onde os métodos informais apresentam problemas, garantindo precisão e diminuindo as possibilidades de inconsistências em definições de linguagens. Porém, eles também apresentam problemas: podem tornar a leitura e escrita de programas tarefas complicadas, e, além disto, sua aplicação é restringida pela falta de ferramentas que sejam fáceis de utilizar e, ao mesmo tempo, gerem código eficiente.

Máquinas de Estado Abstratas (ASM, *Abstract State Machines*), introduzidas por

Yuri Gurevich [Gur94] como um novo modelo computacional, utiliza conceitos simples e bem conhecidos, tornando a leitura e a escrita da especificação bastante direta. Foi originalmente proposta com o objetivo de prover semântica operacional para algoritmos de uma forma mais natural que a máquina de Turing. Este modelo tem sido usado com sucesso para formalizar sistemas de tempo real, seqüenciais, paralelos e distribuídos, além de arquitetura de computadores [BTI+07].

A linguagem Machĭna, baseada nesse método ASM, é uma linguagem de especificação formal que foi desenvolvida no Laboratório de Linguagens de Programação da Universidade Federal de Minas Gerais (LLP-UFMG). A primeira versão da linguagem sofreu alterações e inclusões de novas característica dando origem em 2005 à versão 2.0 [BTI+07].

Machĭna é uma linguagem fortemente tipada e possui suporte à modularidade, criação de novos tipos de dados e construções de alto nível. Um programa em Machĭna consiste na definição de um vocabulário, do estado inicial e da regra de transição que promove a mudança de estado.

1.1 Definição do Problema

Com o aumento crescente da complexidade dos sistemas foi constatado que certos interesses (do inglês, *concerns*) não se encaixam em um único módulo de programa, ou nem mesmo em um conjunto de módulos altamente relacionados. Desta forma, segundo Tzilla Elrad et al. [EFB01], qualquer decomposição de sistemas (procedimental ou orientada por objetos) revelará que alguns interesses estão localizados dentro de módulos específicos, enquanto outros estão espalhados e entrelaçados em vários módulos. Esses últimos são chamados de interesse transversais. Pode-se ter, por exemplo, o entrelaçamento de código de negócio com código de apresentação, e o espalhamento de código de autorização em vários módulos. A Figura 1.1 ilustra o espalhamento de código causado pelo interesse relacionado a autorização.

Desta forma, na especificação de sistemas, pode-se perceber o espalhamento e entrelaçamento de muitos interesses. Na maioria das vezes, estes interesses são classificados e separados (decompostos) de maneiras diferentes, baseadas, por exemplo: em se são interesses funcionais ou não; nas pessoas que estão interessadas no serviço, que o requisitaram, ou que o implementam; em opiniões dadas pelos requerentes; ou no tipo de serviço que oferecem; dentre outros.

A classificação e separação de interesses são úteis para facilitar a leitura da especificação utilizada e identificar nela partes importantes em cada momento do processo de desenvolvimento, seja para elicitar novos interesses, validá-los ou analisar alguns con-

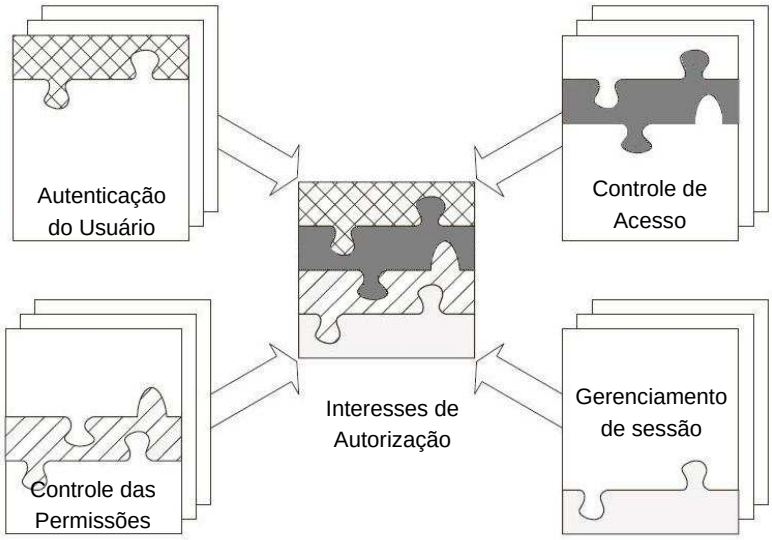

Figura 1.1: Espalhamento de código causado pela necessidade de implementar blocos de código em vários módulos

flitos. Entretanto, muitas vezes estes interesses estão dispostos de forma transversal; como o exemplo do interesse de autorização supra ilustrado.

O problema de espalhamento e entrelaçamento na especificação traz sérias dificuldades para o entendimento de cada interesse, para manipulação da especificação, para a reutilização de partes da especificação e para realizar modificações nos interesses. Estes problemas trazem dificuldades ainda maiores se consideramos o processo de desenvolvimento de software inteiro, pois este depende da boa elaboração dos interesses para o efetivo desenvolvimento e aceitação do produto [Som00].

Assim, o principal problema abordado nesta dissertação é o espalhamento e entrelaçamento de interesses transversais na especificação formal de sistemas. Consideramos que eles são problemas naturais, decorrentes da complexidade dos sistemas e, portanto, uma linguagem de especificação formal deve possuir recursos apropriados para especificar estes interesses transversais. Desta forma, como a linguagem Machĭna, na versão atual, não possui tais recursos, pode-se observar os seguintes problemas em uma especificação:

- o entendimento do código da especificação fica mais difícil, uma vez que temos códigos com propósitos distintos misturados;

- como o código da especificação referente a um interesse transversal não é apropriadamente encapsulado, ele encontra-se espalhado por todo o sistema, desta forma,

uma alteração neste código provocará modificações em vários lugares, tornando com isso difícil a manutenção/evolução do sistema, que é a parte responsável pela maior parcela de custos no processo de desenvolvimento de software;

- visto que o código do interesse principal está misturado com o código do interesse transversal, a reusabilidade do componente fica prejudicada caso desejarmos fazer uso da sua funcionalidade básica em um contexto diferente.

1.2 Solução Proposta

Em busca de uma solução para o problema da separação e composição de interesses transversais, nos últimos anos foi criado o paradigma de orientação por aspectos [KLM+97]. A Programação Orientada por Aspectos (AOP, do inglês *Aspect-Oriented Programming*) propõe mecanismos e abstrações inovadores para melhorar a separação de interesses oferecida pelos paradigmas atuais.

AOP tem por objetivo tornar mais fácil a manutenibilidade e evolução de softwares. Entretanto, segundo Silva e Prado Leite [SL05a], AOP trata de aspectos ou interesses transversais de baixo nível de abstração, quando os requisitos já foram congelados e muitas decisões de gerência e projeto já foram tomadas.

Com o intuito de possibilitar o tratamento de aspectos em todas as etapas do desenvolvimento, têm sido propostos métodos e linguagens de modelagem considerando estes novos elementos e relacionamentos de desenho de maneira a permitir a separação e composição de interesses transversais no nível de desenho [SL05b, dM05, CPA05, RC05]. Estas abordagens têm se estendido também, à fase de definição de requisitos, denominada por alguns de *early-aspects* [RSMA02b].

Dentro deste contexto, este trabalho de dissertação de mestrado teve como objetivo geral reunir as vantagens existentes em uma especificação formal, como a descrição precisa dos requisitos do sistema, a partir da qual podem ser realizadas verificações, e as vantagens da programação orientada por aspectos, como modularização de interesses transversais. Com isso, obtém-se uma nova versão da linguagem Machĭna, denominada, Aspect$\mathcal{M}$, que possui recursos necessários para especificar interesses transversais de forma modular.

Uma das vantagens do uso do modelo ASM para a especificação da semântica de um algoritmo é que esta especificação pode ser executada. Para isso, além de especificar as novas construções da linguagem Machĭna, foi implementado um compilador para a nova versão da linguagem. Assim, a linguagem Machĭna e suas novas construções relativas a aspectos são compilados para código C++ e AspectC++, respectivamente. Deste modo, tendo a linguagem C++ como linguagem alvo, torna-se possível uma execução

rápida e eficiente.

Este processo de compilação está ilustrado na Figura 1.2. Cabe ressaltar que a parte relacionada a AspectC++, arquivo de extensão .ah, está destacada apenas para indicar que sua geração é condicional. Maiores detalhes serão apresentados no Capítulo 5.

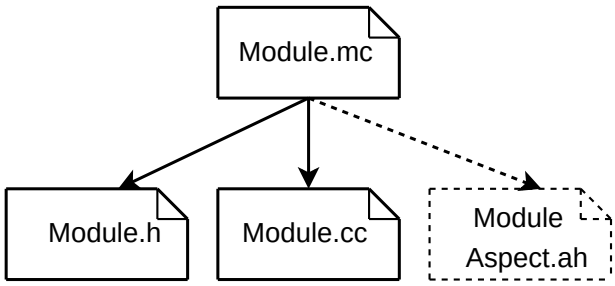

Figura 1.2: Geração de código C++ e AspectC++ a partir de um arquivo Machĭna.

1.3 Validação do Trabalho

Como validação da linguagem proposta, são implementados exemplos, descritos no Capítulo 6, que procuram destacar as vantagens em utilizar as novas construções, dentre elas a modularização obtida. Além disso, a linguagem proposta é avaliada de acordo com os arcabouços conceituais apresentados no Capítulo 2.

1.4 Contribuições

As principais contribuições deste trabalho são:

- Projeto de uma linguagem de programação orientada por aspectos em nível de especificação de requisitos.
- Desenvolvimento de recursos lingüísticos para especificação de requisitos transversais em ambiente de especificação ASM, estendendo assim os conceitos relacionados a AOP para ASM.
- Aumento do poder de abstração de uma linguagem de especificação formal, Machĭna, possibilitando a implementação de interesses transversais de forma modular.
- Criação de um compilador para a nova versão da linguagem Machĭna, possibilitando que as especificações implementadas sejam executadas.

1.5 Organização do Texto

O texto desta dissertação está estruturado em capítulos, da seguinte forma:

Introdução Inicialmente, é feita a apresentação do contexto no qual o trabalho está inserido, e também a identificação do problema a ser resolvido. Em seguida, são citados os objetivos e apresentadas as contribuições.

Programação Orientada por Aspectos São apresentados os principais conceitos relacionados com a programação orientada por aspectos, além de uma teoria informal. Por fim, são apresentadas algumas linguagens orientadas por aspectos.

A Linguagem Machĭna A linguagem formal utilizada para descrever as especificações é apresentada neste capítulo. A sintaxe e semântica são apresentadas e, no final, dois exemplos são descritos objetivando destacar o problema de modularização dos interesses.

A Linguagem Aspect$\mathcal{M}$ Neste capítulo é apresentado a nova versão da linguagem Machĭna. Assim, a sintaxe e a semântica das novas construções são descritas e exemplificadas.

Implementação de Aspect$\mathcal{M}$ É detalhado o mapeamento dos elementos da linguagem Machĭna para C++ e das novas construções para AspectC++, assim como a implementação da costura de código referente a transversalidade estática.

Avaliação e Validação dos Resultados É apresentada uma pequena avaliação conceitual, utilizando os arcabouços apresentados no Capítulo 2. Além disso, são apresentados alguns exemplos que procuram ilustrar a modularização e a clareza da especificação obtida utilizando a linguagem proposta.

Conclusões Conclusões gerais sobre o trabalho são feitas neste capítulo final, bem como são apontados alguns trabalhos futuros.

Capítulo 2

Programação Orientada por Aspectos

Neste capítulo, são apresentados os principais conceitos relacionados com a programação orientada por aspectos (POA). Além disso, é mostrada uma teoria informal sobre aspectos, utilizada para avaliar conceitualmente a linguagem proposta. Por fim, apresentam-se algumas linguagens orientadas por aspectos.

2.1 Visão Geral

Segundo Hugo e Grott [HG05], o advento da Programação Orientada por Objetos (POO) tornou o desenvolvimento de sistemas mais reutilizável, flexível e com uma maior facilidade de manutenção. A POO trouxe algumas boas práticas e princípios, como, o princípio de encapsulamento ou ocultação de informação (*information hiding*); a visão interna de um objeto permanece oculta, de forma que futuras mudanças não afetem outros objetos.

Com o aumento crescente da complexidade das aplicações de software foi constatado que certos interesses (*concerns*) não se encaixam em um único módulo de programa, ou nem mesmo em um conjunto de módulos altamente relacionados. Desta forma, segundo Elrad [EFB99], qualquer realização de decomposição de sistemas (procedural ou orientada por objetos) revelará que alguns interesses estão localizados dentro de módulos específicos, enquanto outros estão espalhados e entrelaçados em vários módulos. Esses últimos são chamados de interesse transversais (*crosscutting concerns*), porque afetam naturalmente os limites de outros interesses.

Um exemplo de interesse transversal é ilustrado pela Figura 2.1. Esta figura é uma representação gráfica do código do servidor Tomcat. As barras verticais representam classes individuais onde o comprimento da barra é proporcional ao número de linhas

de código na respectiva classe. Cada linha horizontal em uma barra representa uma linha de código que faz uma chamada ao pacote de registro de operações no servidor Tomcat. Observe como as chamadas ao pacote de registro de operações encontram-se espalhadas por todo o sistema, este padrão de espalhamento é típico de um interesse transversal.

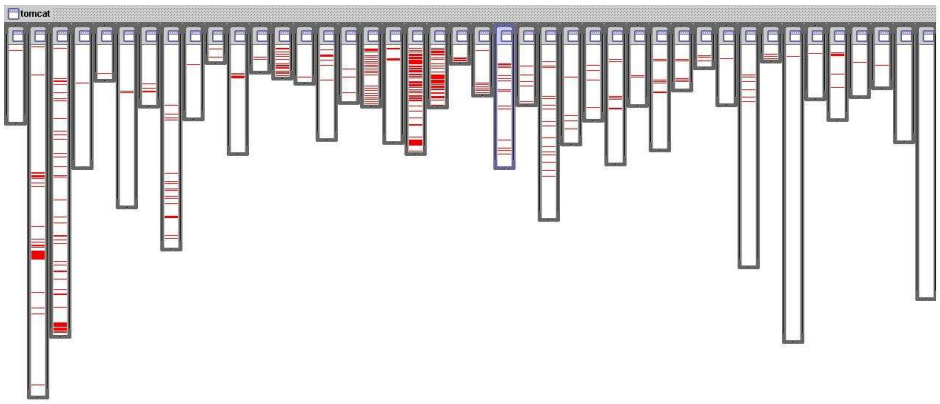

Figura 2.1: Registro de operações no servidor Tomcat [Lem05]

Em geral, interesses transversais cuidam de requisitos que envolvem restrições globais, propriedades sistêmicas e protocolos, como sincronização, persistência, tratamento de erros, mecanismos de auditoria, entre outros. Sem os meios apropriados para a separação, os interesses transversais tendem a ficar espalhados e entrelaçados com outros interesses [LK97]. As conseqüências naturais são uma menor compreensibilidade, uma menor extensibilidade e reusabilidade [Mey97] dos artefatos do código.

As tecnologias de Programação Pós-Objeto propõem mecanismos e abstrações inovadores para melhorar a separação de interesses oferecida pelo paradigma de orientação por objetos. Os termos Programação Pós-Objeto, Separação Avançada de Interesses e, mais recentemente, Desenvolvimento de Software Orientado por Aspectos são normalmente usados para caracterizar a nova geração de tecnologias de separação de interesses [Cha04].

2.2 Separação Avançada de Interesses

A separação de interesses é um princípio fundamental da Engenharia de Software que tenta resolver as limitações da cognição humana ao lidar com a complexidade do software [Dij76, Gar04]. Um interesse é uma parte do problema que queremos tratar como uma unidade conceitual única na solução do software.

O princípio da separação de interesses defende que, para superar a complexidade, deve-se resolver um interesse por vez. Na engenharia de software, esse princípio está relacionado à modularização e à decomposição de sistemas [Som00]. Os sistemas complexos devem ser decompostos em unidades modulares menores e claramente separadas, cada uma lidando com um único interesse. Um módulo é um artefato de programação que pode ser desenvolvido e compilado separadamente de outras partes que compõem o programa [Dij76, Par02]. Como principais benefícios ou vantagens dessa decomposição, pode-se destacar:

- Facilidade de manutenção - como o código fica mais bem modularizado e com menos redundâncias, o engenheiro de software não necessita procurar trechos de códigos por todas as classes da aplicação, pois ele está localizado em poucos módulos;

- Facilidade de compreensão do código - da mesma forma que o tópico anterior, a modularidade faz com que o engenheiro de software possa se concentrar em poucos módulos para entender determinada funcionalidade;

- Facilidade de evolução - como o código encontra-se bem modularizado, adicionar novos aspectos que tratam de outros interesses se torna mais fácil.

A separação dos interesses depende muito da adequabilidade das abstrações e métodos usados ao longo do processo de desenvolvimento de software. Ela também depende dos mecanismos de composição suportados pelas linguagens de programação utilizadas.

Os mecanismos de composição, os métodos e as abstrações orientados por objeto evoluíram para permitir que fosse alcançada a separação de interesses no código-fonte e nos níveis superiores. As classes, os objetos e os métodos são exemplos de abstrações clássicas na engenharia de software orientada por objetos. Como exemplo, considere um programa orientado por objetos para a manipulação de figuras geométricas simples, adaptado de [EAK+01].

Uma figura consiste em alguns elementos, que podem ser pontos ou linhas. Os objetos encapsulam dados e procedimentos. Em um objeto que representa um ponto, os dados são as coordenadas x e y, e os procedimentos, ou métodos, operam nessas coordenadas. A representação de dados está oculta, e os dados só podem ser tratados por métodos dos objetos. Novos tipos de figuras surgem pela especialização. Além disso, como os dados internos estão ocultos, e, enquanto as interfaces estiverem estáveis, a modificação de representações existentes fica localizada em classes e não requer grandes alterações no programa.

A Figura 2.2 mostra um diagrama de classes que descreve o programa orientado por objetos para a manipulação de figuras geométricas simples.

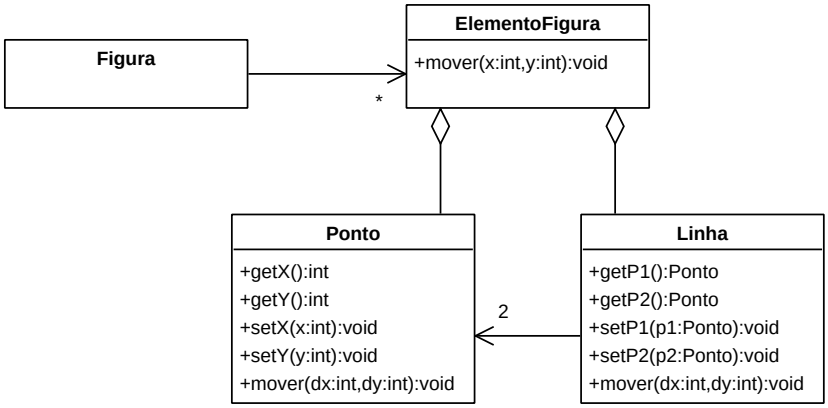

Figura 2.2: Diagrama de classes para o programa manipulador de figuras

Apesar de a programação orientada por objetos ter sido a tecnologia de programação dominante durante anos, a orientação por objetos possui algumas limitações no tratamento de interesses que se encontram naturalmente envolvidos em diversas operações e componentes do sistema.

Ainda considerando o programa manipulador de figuras, suponha que um novo interesse peça a atualização de uma janela de exibição sempre que uma figura for movida ou modificada. A solução OO direta é incluir uma chamada janela.atualizar() em todos os métodos que modificam a aparência de uma figura (como desenhar, girar ou escalonar), de forma que quando esses métodos forem chamados nos objetos das figuras, o método de atualização é chamado no objeto de exibição. A implementação do novo interesse (atualizar a exibição) fica entrelaçada com alguns métodos básicos definidos para os objetos de figuras.

A Figura 2.3 ilustra o diagrama de classes para a nova versão do programa. Observe que o nome dos métodos que implementam o comportamento que atualiza uma janela está destacado em negrito. Este novo interesse não se encaixa em nenhuma classe no diagrama, em vez disso, atravessa várias delas.

Generalizando o exemplo, a Figura 2.4 apresenta a visão de um sistema composto de múltiplos interesses como lógica de negócio, persistência e registro de operações que se tornam entrelaçados à medida que vão sendo inseridos nos módulos de implementação. Cada módulo de implementação trata diversos interesses pertencentes ao sistema. Conseqüentemente, a independência dos interesses não pode ser mantida. A separação avançada de interesses denota as técnicas cujo objetivo é oferecer suporte à modularização desses interesses especiais, chamados de interesses transversais.

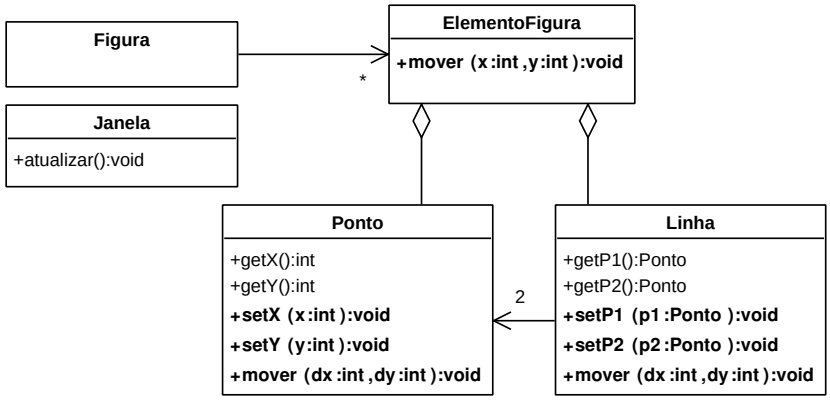

Figura 2.3: Diagrama de classes para a nova versão do programa manipulador de figuras

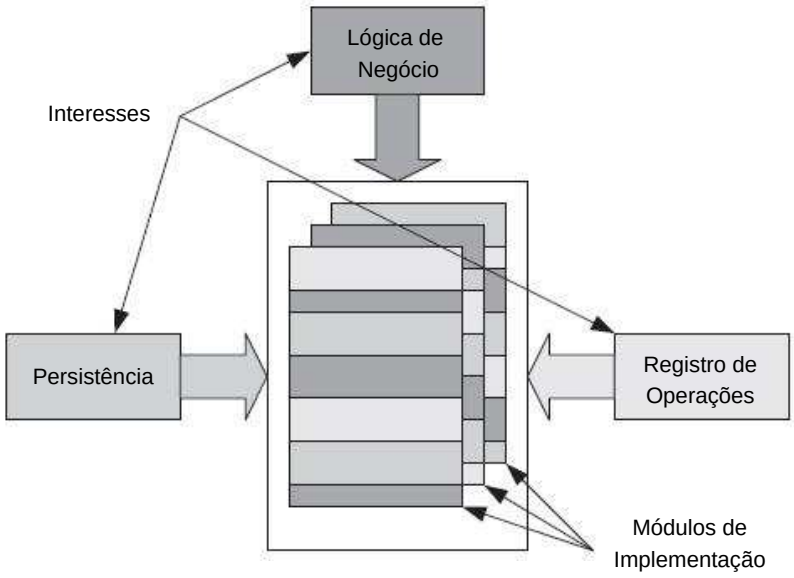

Figura 2.4: Visão de um sistema composto de múltiplos interesses

A Figura 2.5 ilustra a idéia da separação avançada de interesses aplicada às chamadas ao pacote de registro de operações no servidor Tomcat. Comparando-a com a Figura 2.1, página 8, pode-se perceber que o interesse relacionado ao registro de operações, que antes encontrava-se espalhado e entrelaçado com todo o sistema, encontra-se em apenas um módulo. Assim, uma das idéias da separação avançada de interesses é facilitar a compreensão e manutenção do sistema, uma vez que todo interesse ficará modularizado, mesmo na presença de interesses transversais.

Há várias abordagens recentes à separação avançada de interesses, dentre elas, pode-se citar a Programação Orientada por Aspectos (POA, do inglês *Aspect-Oriented Programming*) [2.3], a Programação Orientada por Assunto (*Subject-Oriented Program-*

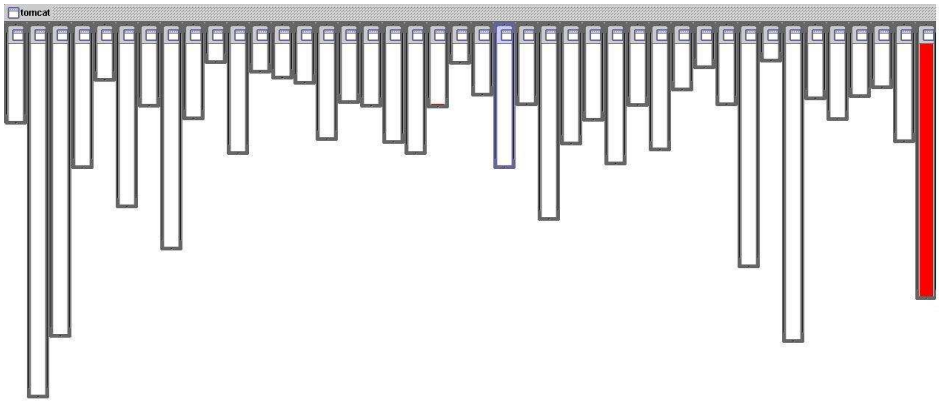

Figura 2.5: Modularização das chamada ao pacote de registro de operações do servidor Tomcat

ming) [HO93], a Programação Adaptativa (*Adaptive Programming*) [Lie96] e a Separação Multidimensional de Interesses (*Multi-Dimensional Separation of Concerns*) [MLWR01]. Essas abordagens, brevemente descritas por Chavez [Cha04], contribuíram para melhorar a programação orientada por objetos, uma vez que proporcionam a separação de interesses em outras dimensões, além das classes e dos objetos. Cada uma dessas abordagens descreve seu próprio modelo de programação para oferecer suporte à modularização dos interesses transversais.

2.3 Programação Orientada por Aspectos

Em 1997, Kiczales et al. apresentam um trabalho chamado *Aspect-Oriented Programming* [KLM+97], que forneceu o primeiro arcabouço conceitual para a POA: um conjunto inicial de termos e conceitos a fim de oferecer suporte a projetos de sistemas baseados em POA. Além disso, apresentou-se uma análise dos problemas que a POA pretendia resolver, assim como vários exemplos expressos em linguagens orientadas por aspectos específicas de domínio.

Kiczales apresenta o termo procedimento generalizado (*generalized procedure*) para denotar "qualquer abstração-chave usada para encapsular uma unidade funcional do sistema geral (objeto, método, procedimento, API etc.)". Componentes são definidos como "propriedades de um sistema, no qual a implementação pode ser encapsulada de forma limpa em um procedimento generalizado". Aspectos são definidos como "propriedades de um sistema, no qual a implementação não pode ser encapsulada em um procedimento generalizado". Além desses termos, pontos de junção (*join point*), transversalidade (*crosscutting*) e processo de combinação (*weaving*) correspondem a ou-

tros conceitos introduzidos e que coletivamente constituem o coração da programação orientada por aspectos.

Segundo o livro *Mastering AspectJ* [GLG03], a POA resolve o problema da modularização dos interesses transversais sem a complexidade e dificuldade encontradas na programação orientada por assunto e na separação multidimensional de interesses. A POA propõe um novo tipo de abstração - denominado aspecto - e novos mecanismos de composição que permitem a descrição modular e a composição de interesses, que geralmente se encontram espalhados e entrelaçados em vários pontos de um sistema orientado por objetos.

Dentre as propriedades existentes na abstração de aspecto, destacam-se a transparência (*obliviousness*) e a quantificação [CL03]. A transparência é o ato ou o efeito de deixar transparente, no sentido de poder ser esquecido ou passar despercebido. A transparência é a idéia de que os componentes não precisam ser especificamente preparados para receber as melhorias proporcionadas pelos aspectos. A quantificação é a capacidade de escrever declarações unitárias e separadas que afetam muitos lugares não-locais no sistema de programação. A quantificação proporciona a capacidade de realizar declarações como: "em programas P, sempre que ocorrer a condição C, faça a ação A" [FF00]. Quando há a propriedade de quantificação, os aspectos podem afetar um número arbitrário de componentes simultaneamente.

A POA baseia-se na idéia de que se pode melhorar o desenvolvimento de sistemas especificando-se separadamente os vários interesses e descrições de seus relacionamentos e, posteriormente, deixando-se para mecanismos existentes em ambientes POA a composição dos mesmos para a obtenção de um sistema coerente.

A POA envolve três etapas distintas [GLG03]:

- Decomposição: os vários interesses do sistema são identificados e classificados como principais ou transversais.
- Implementação: os interesses são implementados separadamente em componentes (para os interesses principais) e aspectos (para os interesses transversais).
- Recomposição: o sistema é recomposto a partir dos interesses implementados seguindo regras de recomposição. Esta etapa também é chamada de combinação.

A Figura 2.6 ilustra essas etapas supra mencionadas. Primeiramente, realiza-se a decomposição dos interesses e, logo após, a implementação dos mesmos em módulos individuais. A etapa final consiste em combinar as implementações usando os aspectos para formar o sistema final.

O objetivo da POA é, então, oferecer suporte ao programador para separar de forma limpa os componentes e aspectos uns dos outros, fornecendo mecanismos que

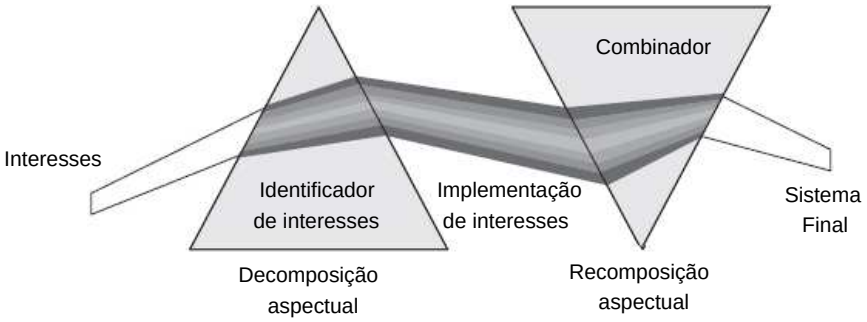

Figura 2.6: Etapas do desenvolvimento POA [Lad03]

possibilitam abstraí-los e compô-los a fim de produzir o sistema geral. Com isso, a POA pode ser vista como uma nova metodologia de programação que promove a separação avançada de interesses ao introduzir uma nova unidade modular, chamada aspecto, para a modularização dos interesses transversais.

Recentemente, muitos grupos de pesquisa começaram a discutir a função dos aspectos em outras fases do processo de desenvolvimento de software. Assim, o restante desta seção descreve alguns benefícios da programação orientada por aspectos e apresenta alguns trabalhos relacionados a aplicação da separação de interesses na fase de requisitos.

No fim desta seção, é apresentada uma teoria informal de aspectos - um arcabouço conceitual para analisar uma técnica orientada por aspecto com base em cinco conceitos principais: aspectos, componentes, pontos de junção, transversalidade e processo de combinação.

2.3.1 Benefícios da Programação Orientada por Aspectos

No livro *AspectJ in Action* [Lad03], são destacados alguns benefícios da programação orientada por aspectos, dentre esses, pode-se destacar:

- **Diminuição das responsabilidades dos módulos** - a POA permite que um módulo seja responsável somente pelo seu interesse principal, não sendo mais entrelaçado por interesses de outros módulos. Por exemplo, um módulo que acessa o banco de dados não é responsável pelo *pool* de conexões. Isto resulta em uma melhor distribuição das responsabilidades, possibilitando uma melhor rastreabilidade.

- **Modularização**: a POA fornece um mecanismo para implementar cada interesse separadamente com o mínimo de acoplamento. O resultado é uma implementação modularizada mesmo na presença de interesse transversais. Tal implementação

reduz a redundância de código, uma vez que cada interesse encontra-se separado. Desta forma, a implementação modularizada facilita o entendimento e a manutenção do sistema.

- **Facilidade na evolução do sistema**: a POA modulariza os interesses transversais em aspectos e faz com que interesses principais não tenham consciência de que esses interesses estão sendo interceptados, ou seja, os interesses principais não precisam ser especificamente preparados para receber as melhorias proporcionadas pelos aspectos. Assim, adicionar uma nova funcionalidade não requer grandes modificações. Além disso, quando um novo módulo é adicionado ao sistema, os aspectos existentes podem entrecortar o mesmo, evitando retrabalho.

- **Protelar a implementação de interesses**: com a POA o engenheiro de software pode protelar a implementação de alguns interesses, uma vez que é possível implementá-los posteriormente em aspectos separados. Novos interesses com características transversais podem ser manipulados criando novos aspectos.

- **Reusabilidade de código**: a chave para uma maior reusabilidade de código está em uma implementação com baixo acoplamento e alta coesão. A POA permite a implementação com baixo acoplamento pois implementa cada aspecto como um módulo separado e, consequentemente, uma alta coesão, possibilitando que cada módulo implementado seja desaclopado um do outro.

- **Redução do custo de implementação**: a POA permite uma redução do custo de implementação porque facilita a compreensão do código, diminuindo, assim, o tempo para realizar uma modificação. Além disso, aumenta a produtividade do desenvolvedor, porque o mesmo pode ficar focado na implementação de um único interesse sem se preocupar com os demais.

2.3.2 Separação de Interesses na Fase de Requisitos

O Desenvolvimento de Software Orientado por Aspectos [Kic02] é uma área emergente cujo objetivo é promover a separação avançada de interesses ao longo de todo o ciclo de vida do desenvolvimento de software.

Segundo Silva e Leite [SL05a], a POA trata de aspectos ou interesses transversais de baixo nível de abstração, quando os requisitos já foram congelados e muitas decisões de gerência e desenho já foram tomadas.

Com o intuito de possibilitar o tratamento de aspectos em todas as etapas do desenvolvimento, têm sido propostos métodos e linguagens de modelagem considerando

estes novos elementos e relacionamentos de desenho de maneira a permitir a separação e composição de interesses transversais no nível de desenho [SL05b, CL03].

Além disso, com o objetivo de melhorar o entendimento e, conseqüentemente, a qualidade do documento de requisitos, desta maneira antecipando a correção de possíveis erros, alguns trabalhos propõem a utilização da separação avançada de interesses na fase de requisitos [RSMA02a], cuja a idéia principal é que a especificação dos interesses é mais bem compreendida quando os interesses principais e transversais estiverem especificados separadamente. Além dessa separação de interesses, pesquisas nessa área fornecem algumas técnicas para modelagem e ainda como e onde posteriormente combinar o que foi separado [RSMA02a].

Rashid et al. [RSMA02a] propõem um modelo de processo genérico, denominado AORE (*Aspect-Oriented Requirements Engineering*), para separar interesses transversais no nível de requisitos. No modelo AORE, o termo "interesse" é utilizado em um sentido mais restrito, correspondendo a um interesse transversal de alto nível de abstração, por exemplo, segurança ou auditoria.

Diante das vantagens propostas pela aplicação de técnicas orientadas por aspectos na fase de requisitos, Ramos e Castro [RC05] propõem uma metodologia para avaliação da qualidade desta nova maneira de especificar os requisitos. Como principal conclusão, os autores afirmam que "a busca de informações sobre atributos foi facilitada, pois apenas foi necessário identificar no diagrama de casos de uso onde se encontrava o atributo procurado e posteriormente localizar a tabela que continha sua descrição. Diferente de quando o documento de requisitos não é orientado por aspectos, pois neste caso o avaliador teria de fazer uma busca em todo o documento de requisitos a fim de encontrar trechos que tratam do atributo".

2.3.3 Teoria Informal sobre Aspectos

Segundo Masuhara e Kiczales [MK03], não existe um consenso em relação ao que torna uma técnica orientada por aspectos. Diante deste contexto, Masuhara e Kiczales apresentam um arcabouço para modelar a semântica básica de quatro tecnologias de desenvolvimento de software orientado por aspectos e seus mecanismos. Na Tabela 2.1, essas tecnologias são relacionadas com suas respectivas implementações.

Masuhara e Kiczales [MK03] tentam caracterizar quais propriedades de cada mecanismo permite a modularização dos interesses transversais. Uma propriedade crítica de seu arcabouço é que ele modela os pontos de junção como pontos existentes no resultado do processo de combinação em vez de em um dos programas de entrada. Ademais, o trabalho de Masuhara e Kiczales destaca-se por outras duas razões. Primeiro, ele trata componentes e aspectos uniformemente, assumindo que não há linguagem de compo-

Tecnologia	Implementação
Composição de classes	Hyper/J
Especificações transversais	DAJ, DemeterJ, DemeterC++
Classes abertas(*open classes*)	AspectJ, MultiJava
Conjunto de Junção e Adendo	AspectJ, AspectC++, AspectWerz, Pythius

Tabela 2.1: Tecnologias do desenvolvimento de software orientado por aspectos

nentes primária nem dicotomia baseada aspecto, ou seja, não existe uma distinção clara entre componentes e aspectos. Segundo, ele permite a descrição de aspectos que afetam outros aspectos naturalmente.

Chavez e Lucena [CL03] propõem um arcabouço conceitual para a programação orientada por aspectos que oferece uma terminologia consistente e uma semântica básica para pensar sobre um problema em termos dos conceitos e propriedades que caracterizam o estilo da POA como um paradigma emergente de desenvolvimento de software. Esse arcabouço é chamado de modelo de aspectos e subdivide-se entre quatro modelos conceituais inter-relacionados: (i) o modelo de componentes, (ii) o modelo de pontos de junção, (iii) o modelo principal e (iv) o modelo de processo de combinação. A Figura 2.7 ilustra o modelo de aspectos. Assim, definiu-se que uma linguagem orientada por aspectos é uma linguagem que oferece suporte ao modelo de aspectos.

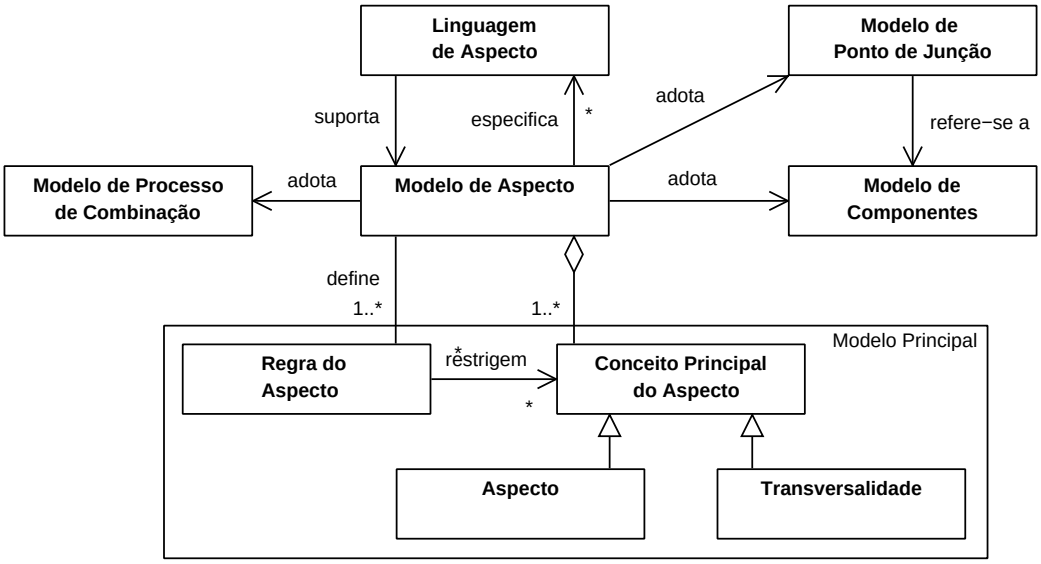

Figura 2.7: Modelo de Aspecto [CL03]

Nas próximas seções, descrevem-se os modelos conceituais. Utilizaram-se diagramas de classes para representar cada modelo conceitual em termos dos principais conceitos

envolvidos.

2.3.3.1 O Modelo de Componentes

O modelo de componentes representa um arcabouço conceitual usado para pensar sobre um problema e decompô-lo em termos de um determinado tipo de componente [CL03]. Esse arcabouço consiste em categoria de conceitos principais (mecanismos de composição e componentes), regras que restringem elementos dessas categorias e um conjunto de princípios gerais.

Por exemplo, no modelo de objetos, os principais conceitos são objetos, classes e herança. Java e C++ são linguagens de programação orientadas por objetos uma vez que seu arcabouço conceitual é o modelo de objetos, ou seja, ambas oferecem suporte ao modelo de objetos.

Um modelo de componentes pode ter suporte a uma ou mais linguagens de componentes. A Figura 2.8 ilustra um diagrama de classes para o modelo de componentes.

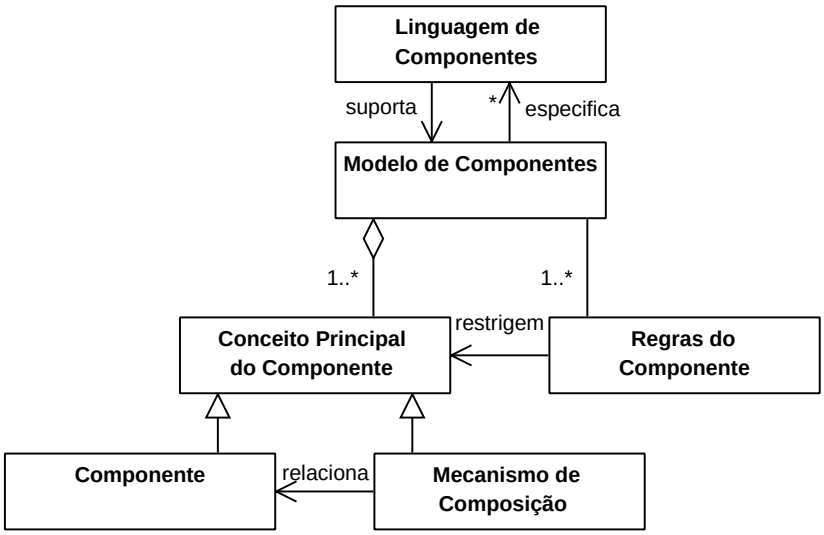

Figura 2.8: Modelo de componentes [CL03]

2.3.3.2 O Modelo de Pontos de Junção

A POA oferece mecanismos para que os aspectos possam ser construídos em módulos separados e provê meios para a definição de pontos do programa onde esses aspectos possam definir algum comportamento. A partir daí, o sistema final pode ser obtido, combinando os módulos básicos com os aspectos. Dessa forma, a POA pretende dar suporte aos interesses transversais assim como a POO tem dado suporte aos objetos [KHH+01].

O modelo de ponto de junção (MPJ) representa um arcabouço conceitual usado para descrever os tipos de pontos de junção de interesse e as restrições associadas a seu uso. Esse arcabouço é altamente dependente do modelo de componentes adotado. A Figura 2.9 apresenta um diagrama de classes para o modelo de pontos de junção.

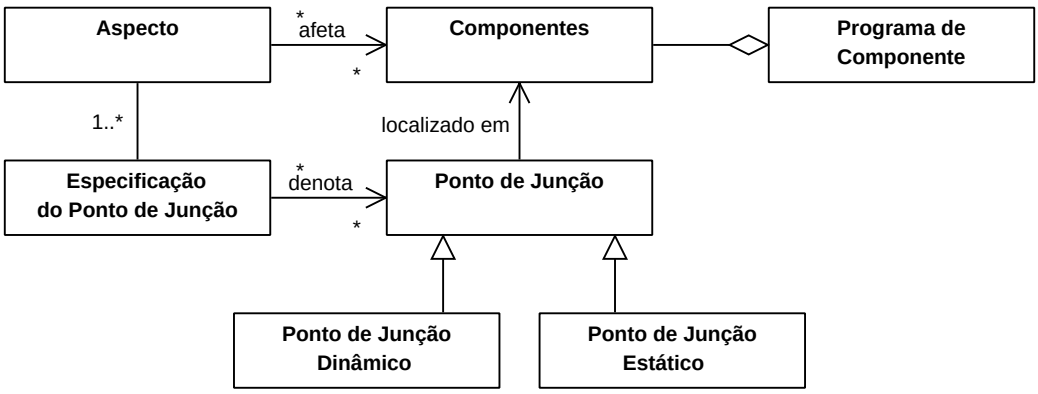

Figura 2.9: Modelo de Pontos de Junção, adaptado de [CL03]

De acordo com Masuhara et al. [MKD03] a capacidade de uma linguagem de aspectos suportar a modularização de interesses transversais é determinada pelo MPJ. Um MPJ consiste, basicamente, em três elementos: (i) pontos de junção, (ii) meios que identifiquem um ponto de junção e (iii) meios que especifiquem semântica em um ponto de junção.

Pontos de Junção (*Join Point*) são posições bem definidas na execução de um programa, como, chamadas e execuções de métodos ou leitura e escrita de campos. Os pontos de junção podem também possuir contexto associado. Por exemplo, o contexto de um ponto de junção de chamada de método contém o objeto alvo da chamada e a lista de argumentos. Masuhara et al. destacam que pontos de junção não são necessariamente construções explícitas da linguagem de componentes, são elementos claros, mas talvez implícitos, da semântica do programa de componentes. Desta forma, um ponto de junção pode denotar um elemento relacionado à estrutura estática ou dinâmica de um programa de componente. Como o exemplo do manipulador de figuras, Seção 2.2, em que o ponto de junção seria o fluxo de dados do programa de componente.

Um ponto de junção estático (*static join point*) é um local na estrutura estática de um componente, enquanto que um ponto de junção dinâmico (*dynamic join point*) é um local no comportamento dinâmico de um programa de componente. Um ponto de junção dinâmico possui uma sombra estática (*static shadow*) correspondente na parte estrutural do componente. Um sombra estática [HH04] é uma região no componente que representa um ponto de combinação dinâmico relacionado a tal componente.

Meios que identifiquem um Ponto de Junção: um conjunto de junção (*pointcut*) seleciona pontos de junção e expõe dados do contexto de execução desses pontos. Eles podem ser compostos por operações de conjunto (união, interseção e diferença) a fim de criar outros conjuntos de junção [WKD04]. Um designador de conjunto de junção é uma fórmula que especifica um conjunto de pontos de junção ao qual um adendo (*advice*) pode ser aplicado.

Há muitos locais na estrutura ou comportamento de componentes que podem ser usados como conjunto de junção, mas, na prática, somente um subconjunto é considerado útil [OT98]. As linguagens orientadas por aspectos devem definir seu conjunto de junção levando em consideração sua linguagem de componentes correspondente.

Meios que especifiquem semântica em um Ponto de Junção: um adendo compreende o comportamento de aspectos que deve ser executado em cada ponto de junção. Um adendo é composto de um conjunto de junção e um bloco de código. Quando o ponto de junção é capturado pelo conjunto de junção o bloco de código é executado.

2.3.3.3 O Modelo Principal

Segundo Chavez e Lucena [CL03], o modelo principal representa um arcabouço conceitual usado para descrever aspectos, como um mecanismo de abstração, e transversalidade, como um mecanismo de composição.

Aspectos, em [KLM+97], são definidos como "propriedades de um sistema, no qual a implementação não pode ser encapsulada em um procedimento generalizado". Assim, podem-se definir os aspectos como propriedades sistêmicas que afetam os componentes causando espalhamento e entrelaçamento de interesses. Utiliza-se, também, o termo *aspecto* para denotar uma entidade de primeira classe que oferece uma representação modular para um interesse transversal.

Transversalidade é definida como um fenômeno observado sempre que duas propriedades programadas devem compor de forma diferente e, mesmo assim, serem coordenadas. Além disso, a transversalidade é restrita a um conjunto pré-definido de pontos especificados por um MPJ e, desta forma, pode ser subdivida em transversalidade dinâmica e estática.

A transversalidade dinâmica consiste em introduzir novos comportamentos na execução dos componentes, ou seja, é um tipo de transversalidade que utiliza pontos de junção dinâmicos e afeta o comportamento dinâmico dos componentes. A transversalidade estática consiste em introduzir modificações nas estruturas estáticas dos componentes, sendo assim, um tipo de transversalidade que utiliza pontos de junção estáticos e, portanto, afeta a estrutura estática dos componentes.

Além de apresentar alguns conceitos sobre aspecto e transversalidade, Chavez e Lucena definem o conceito de interface transversal como especificações de pontos de junção que descrevem os tipos de pontos de junção de interesse para o aspecto, restrito pelo MPJ adotado.

2.3.3.4 O Modelo do Processo de Combinação

O modelo de processo de combinação representa um arcabouço conceitual usado para descrever os tipos de mecanismos de combinação (*weaving*). O termo combinador de aspectos (*aspect weaver*) denota um processador de linguagem especial que oferece suporte à composição entre os componentes e os aspectos.

O conceito de ponto de junção é essencial para o combinador de aspectos, pois representa os pontos da linguagem de componente que o aspecto pode afetar. O combinador de aspectos identifica nos componentes os pontos de junção onde os aspectos se aplicam, produzindo o código final da aplicação, que programam tanto as propriedades definidas pelos componentes quanto aquelas definidas pelos aspectos. Assim, o combinador de aspectos trabalha gerando uma representação de um ponto de junção do programa do componente e, em seguida, executa (ou compila) os programas de aspectos em relação a ele [KLM+97].

Combinadores de aspectos podem atuar em tempo de compilação ou execução. Implementações de combinadores em tempo de execução oferecem a possibilidade interessante de permitir a adição/exclusão de aspectos com a aplicação em pleno funcionamento.

2.4 Linguagens Orientadas por Aspectos

As linguagens orientadas por aspectos podem ser classificadas em linguagens de propósito específico e linguagens de propósito geral.

As linguagens orientadas por aspecto de propósito específico, como o próprio nome informa, são linguagens que tratam de comportamentos específicos dentro de um sistema, por exemplo, sincronismo e tratamento de exceções, não fornecendo suporte a qualquer outro tipo de componente [HG05].

Para garantir a utilização dos aspectos conforme foram projetadas, essas linguagens geralmente impõem alguma restrição no uso da linguagem de componente. Isto é, a linguagem de aspecto poderá impedir a utilização de determinadas palavras que são reservadas da linguagem de aspectos para evitar que a linguagem de componente implemente interesses, que deveriam ser implementados como aspectos. Como exemplo, pode-se citar a linguagem Malaj [CGMP00, AC02], uma linguagem orientada por

aspectos de domínio específico que oferece suporte a sincronização e realocação.

As linguagens orientadas por aspecto de propósito geral permitem a implementação de qualquer tipo de aspecto. Não é possível programar restrições junto à linguagem de componentes, pois, geralmente, ambas possuem o mesmo conjunto de instruções [HG05]. Como exemplo, pode-se citar a linguagem AspectJ, que possui a linguagem Java como linguagem de componente.

A linguagem proposta nesta dissertação foi baseada, praticamente, em AspectJ. Desta forma, o restante desta seção apresenta suas principais características e construções. Por fim, são apresentadas mais algumas linguagens orientadas por aspectos que contribuíram para o trabalho proposto ou acrescentam alguns conceitos importantes à separação avançada de interesses.

2.4.1 AspectJ

AspectJ é uma linguagem de programação orientada por aspectos de propósito geral baseada nos principais conceitos da POA apresentados em [KLM+97].

A linguagem AspectJ foi desenvolvida pelo grupo de POA do Xerox PARC como uma base para uma avaliação empírica da programação orientada por aspectos em Java. Segundo Kiczales et al. [KHH+01], a compatibilidade foi a principal questão no projeto da linguagem. Por compatibilidade entende-se:

- Compatibilidade superior - todos os programas válidos em Java devem ser programas válidos em AspectJ.
- Compatibilidade de plataformas - todos os programas válidos em AspectJ devem ser executados também nas máquinas virtuais Java.
- Compatibilidade de ferramentas - as ferramentas existentes, inclusive as ferramentas de projeto, documentação e ambientes de desenvolvimento integrados, podem ser usadas.
- Compatibilidade de programador - os programadores de Java devem se sentir confortáveis programando em AspectJ.

Esses requisitos de compatibilidade foram motivados por outro objetivo principal: o desenvolvimento e o suporte adequado a uma comunidade substancial de usuários. De fato, muitas características incorporadas nas versões evolutivas de AspectJ surgiram a partir de *feedbacks* recebidos de usuários e de discussões surgidas em listas de discussão de usuários de AspectJ [Cha04].

AspectJ estende Java com dois tipos de implementação de interesses transversais: a transversalidade dinâmica, que permite definir implementações adicionais em

pontos bem definidos do programa, e a transversalidade estática, que afeta as assinaturas estáticas das classes e interfaces de um programa Java [TBBV04]. Segundo Lemos [Lem05], as novas construções de AspectJ consistem em: conjunto de junção que identificam pontos de junção; adendos que definem o comportamento de um dado conjunto de junção; construções para afetar estaticamente a estrutura dos módulos básicos do programa e os aspectos que encapsulam os interesses transversais.

Nas Listagens 2.1 e 2.2, são mostradas uma implementação em AspectJ para o programa de manipulação de figuras geométricas, exemplificado na Seção 2.2. Essa implementação será utilizada ao longo desta seção. O aspecto AtualizacaoJanela se encarrega de atualizar a visualização todas as vezes que alguma figura for alterada. Como as figuras são alteradas a partir dos métodos que alteram suas coordenadas, os adendos são definidos nas chamadas a esses métodos.

```
1  public class Ponto implements ElementoFigura {
2      private int x=0, y=0;
3      int getX() { return x; }
4      int getY() { return y; }
5      void setX(int x) { this.x = x; }
6      void setY(int y) { this.y = y; }
7      public void mover(int dx, int dy) { x += dx; y += dx; }
8  }
9  public class Linha implements ElementoFigura {
10     private Ponto p1, p2;
11     Ponto getP1() { return p1; }
12     Ponto getP2() { return p2; }
13     void setP1(Ponto p1) { this.p1 = p1; }
14     void setP2(Ponto p2) { this.p2 = p2; }
15     public void mover(int dx, int dy) {
16         p1.mover(dx, dy); p2.mover(dx, dy);
17     }
18 }
```

Listagem 2.1: Implementação de algumas classes para o manipulador de figuras.

```
1  public aspect AtualizaJanela {
2      public pointcut mudancaEstado() :
3          call(void ElementoFigura+.mover(..)) ||
4          call(* Ponto.set*(*)) || call(* Linha.set*(*));
5      after() returning : mudancaEstado () {
6          Janela.atualiza();
7      }
8  }
```

Listagem 2.2: Implementação do novo requisito em AspectJ.

2.4.1.1 Transversalidade Dinâmica

A transversalidade dinâmica é restrita a um conjunto pré-definido de pontos especificados por um modelo de pontos de junção. Esse modelo concentra-se na execução dinâmica de programas em Java.

AspectJ adota um modelo no qual um ponto de junção é um ponto bem-definido na execução dinâmica de um programa em Java [KHH+01]. Os pontos de junção de AspectJ incluem a chamada ou a execução de métodos, construtores ou tratadores de exceção, acesso ou modificação do valor de um atributo, etc.

No modelo de ponto de junção de AspectJ, os pontos de junção podem ser considerados vértices de um grafo de chamadas de objetos em tempo de execução. Estes vértices incluem pontos em que um objeto recebe uma chamada de método e pontos em que um campo de um objeto é referenciado. As arestas são relações do fluxo de controle entre os vértices. O controle passa por cada ponto de junção duas vezes, por exemplo, quando uma mensagem é enviada e quando essa mensagem é respondida.

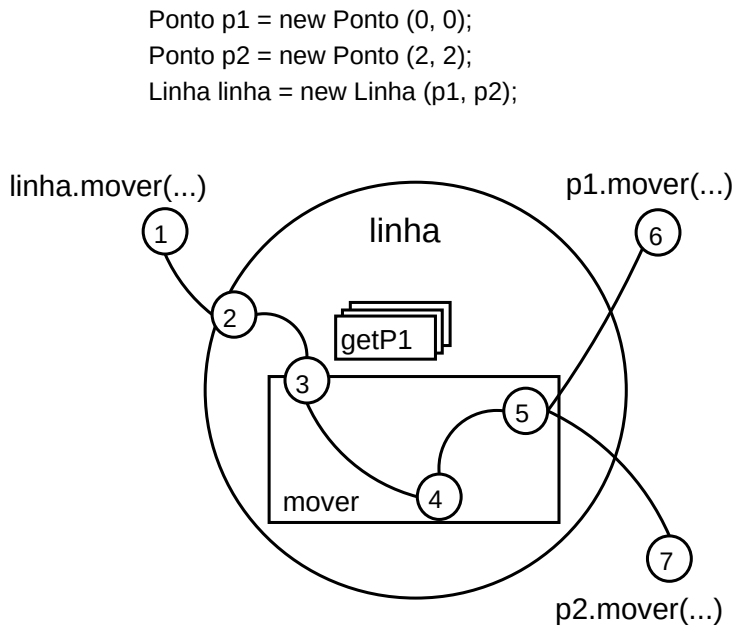

Figura 2.10: Pontos de junção dinâmico em AspectJ, adaptado de [KHH+01]

Baseado na Listagem 2.1, a Figura 2.10 ilustra alguns pontos de junção. As primeiras três linhas de código dessa figura criam um objeto da classe Linha, representado por um círculo grande. Os retângulos representam métodos e os círculos numerados representam os pontos de junção. Executando o comando linha.mover(..), inicia-se uma computação que prossegue atráves dos pontos de junção destacados abaixo:

1. Chamada de método correspondente ao método mover sendo chamado sobre o objeto linha.
2. Recepção da chamada de método que linha recebe da chamada de mover.
3. Execução do método mover definido na classe Linha.
4. Referência ao campo p1 do objeto linha.
5. Chamada de método correspondente ao método mover sendo chamado sobre o objeto p1.
   ...
6. Chamada de método correspondente ao método mover sendo chamado sobre o objeto p2.
   ...

Conjuntos de junção são utilizados para identificar pontos de junção no fluxo do programa. A partir da especificação desses pontos, adendos podem ser definidos - tal como realizar uma certa ação antes ou depois da execução dos pontos de junção. Em AspectJ, os conjuntos de junção podem conter nomes ou serem anônimos. Na Figura 2.11 é mostrado um exemplo de conjunto de junção nomeado. Um conjunto de junção pode ser definido como uma combinação de conjuntos de junção, utilizando operações de conjunto, &&(interseção), ||(união) e !(complementação). O conjunto de junção mudancaEstado, da Figura 2.11, por exemplo, captura todas as chamadas ao método mover da classe ElementoFigura , que recebe dois inteiros como parâmetros e não retorna nenhum valor.

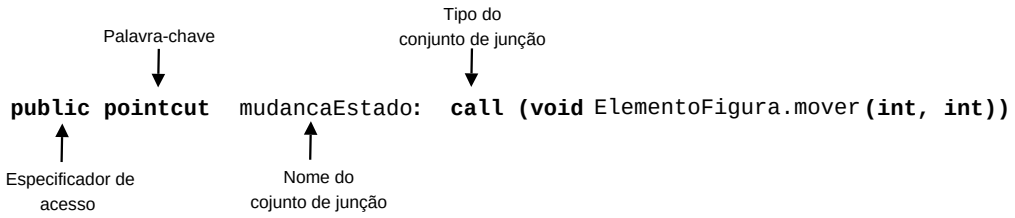

Figura 2.11: Definição de um conjunto de junção, adaptado de [Lad03]

Dado que os interesses transversais, por definição, ficam espalhados por diversos módulos, uma linguagem orientada por aspectos deve prover um modo prático para capturar os pontos de junção de interesse. AspectJ possibilita a utilização de alguns símbolos curinga (*wildcard*) na construção de um conjunto de junção, assim capturando pontos de junção que compartilhem alguma característica em comum.

O símbolo "+" denota qualquer subclasse ou subinterface de um dado tipo[1]. Já o símbolo ".." representa qualquer número de caracteres incluindo o ".". Esse símbolo pode ser utilizado para designar métodos com quaisquer números e tipos de argumentos. A construção **void** ElementoFigura+.mover(..), por exemplo, representa todos os métodos de nome mover que pertençam à classe ou subclasses de ElementoFigura e que retorne **void**.

O símbolo "*" representa qualquer número de caracteres com exceção do ".". Desta forma, pode-se utilizá-lo para representar qualquer tipo de retorno, * Ponto.getP1(), classe, int *.getP1(), ou método * Ponto.*(..). Além disso, esse símbolo pode ser utilizado em nomes parciais, como em int Ponto.get*().

Em Java, as classes, interfaces, métodos e atributos contêm assinaturas. Em AspectJ, pode-se utilizar padrões de assinatura para especificar várias assinaturas contidas em um programa. Os símbolos curingas e as operações de conjunto podem ser utilizados para especificar um padrão de assinatura. Por exemplo, private !String *.* denota todos os campos privados que não sejam do tipo String.

Além disso, AspectJ possibilita a utilização de designadores para categorizar um conjunto de pontos de junção. Um designador de conjunto de junção é uma fórmula que especifica um conjunto de pontos de junção ao qual um adendo (*advice*) pode ser aplicado.

Na Tabela 2.2 são relacionados os principais designadores implementados em AspectJ, com suas respectivas características.

Designador	**Características**
call(assinatura)	Invocação do método / construtor identificado por assinatura
execution(assinatura)	Execução do método / construtor identificado por assinatura
get(assinatura)	Acesso a atributo identificado por assinatura
set(assinatura)	Atribuição do atributo identificado por assinatura
this(padrão tipo)	Objeto em execução é instância do padrão tipo
target(padrão tipo)	Objeto de destino é instância do padrão tipo
args(padrão tipo)	Os argumentos são instância do padrão tipo
within(padrão tipo)	O código em execução está definido em padrão tipo
cflow(padrão tipo)	O código em execução está no fluxo de controle definido em padrão tipo

Tabela 2.2: Designadores em AspectJ.

Os adendos são construções similares a métodos e representam códigos que devem ser executados em pontos de junção [TBBV04]. Em AspectJ, é possível definir três tipos de adendos: (i) anterior (*before*), (ii) posterior (*after*) e (iii) de contorno (*around*).

[1] *tipo* no AspectJ refere-se a uma classe, interface, tipo primitivo ou aspecto

Um adendo anterior é executado imediatamente antes da execução do ponto de junção, ao passo que um adendo posterior é executado imediatamente após a execução do ponto de junção.

É possível, ainda, definir adendos posteriores para serem executados após o retorno normal de um método, utilizando-se para isso a construção after returning, e em caso de retorno com lançamento de exceção, utilizando-se a construção after throwing.

Por exemplo, considere o código da Listagem 2.2. O conjunto de junção mudancaEstado representa os pontos de junção referentes a modificação de uma figura. O adendo posterior indica que, após a execução de um ponto de junção pertencente ao conjunto mudancaEstado, deve-se atualizar a sua exibição; isto é feito por meio do método Janela.atualiza.

Os aspectos de AspectJ são unidades modulares de implementação de interesses transversais. Um aspecto é definido como uma classe e pode ter métodos, campos, construtores, inicializadores, conjuntos de junção e adendos. O modelo de implementação de AspectJ define aspectos como tipos e instâncias de aspectos como objetos regulares que não podem ser explicitamente instanciados. Para obter referência a um aspecto, utiliza-se o método estático aspectOf(), presente em todos os aspectos.

AspectJ fornece regras para estabelecer a precedência relativa entre aspectos, para situações em que mais de um adendo puder ser aplicado a um ponto de junção.

2.4.1.2 Transversalidade Estática

Como definido na Seção 2.3.3.3, a transversalidade estática consiste em introduzir modificações nas estruturas estáticas dos componentes. Enquanto a transversalidade dinâmica, obtida a partir do modelo de ponto de junção, permite modificar o comportamento da execução do programa, a transversalidade estática permite redefinir a estrutura estática dos tipos - classes, interfaces ou outros aspectos - e o seu comportamento em tempo de execução. Por exemplo, os aspectos podem declarar membros que atravessam várias classes ou alteram o relacionamento de herança entre classes.

Em AspectJ, é possível implementar os seguintes tipos de transversalidade estática [TBBV04]: (i) introdução de campos e métodos em classes e interfaces; (ii) modificação da hierarquia de tipos; (iii) declaração de erros e advertências de compilação; (iv) enfraquecimento de exceções.

Introdução de campos e métodos em classes e interfaces, a linguagem AspectJ possui um mecanismo denominado introdução (*introduction* ou *open classes*), que permite a inclusão de atributos e métodos em classes ou interfaces de forma transversal. É possível também adicionar campos e métodos não-abstratos em implementações de interfaces de Java, permitindo adicionar um comportamento padrão às classes que as

implementem.

Modificação da hierarquia de tipos, por meio da construção declare parents, é possível modificar a hierarquia de classes, de modo a definir superclasses e implementação de interfaces para classes e interfaces já existentes, desde que isto não viole as regras de herança de Java. Como conseqüência, obtém-se o desacoplamento dos aspectos e das classes específicas da aplicação, aumentando, assim, o grau de reúso dos aspectos.

Declaração de erros e advertências de compilação, em AspectJ, é possível declarar erros e advertências de compilação. Com este mecanismo, obtém-se comportamento similar ao obtido por meio das diretivas de compilação #error e #warning. A diretiva #error faz com que o compilador reporte um erro fatal, já a diretiva #warning faz como que seja gerado um aviso, porém o processo de compilação não é encerrado.

Enfraquecimento de exceções, por meio da construção declare soft, é possível silenciar exceções de forma seletiva e relançá-las como exceções não checadas. Esta construção é constituída por uma exceção e por um conjunto de junção. A sua semântica consiste em enfraquecer a exceção para o conjunto de junção especificado, fazendo com que o compilador Java não exija o seu tratamento.

Conclusão: o modelo de ponto de junção utilizado na linguagem proposta é similar ao da linguagem AspectJ, porém adaptado para o contexto da linguagem Machĭna. Por exemplo, é indiferente realizar uma operação antes ou depois ao acesso uma função, visto que tudo ocorre paralelamente. Além disso, utilizou-se a mesma sintaxe para definir e combinar um conjunto de junção, e para a declaração de intertipos, que no contexto da linguagem Machĭna, permite a introdução de funções, tipos e ações.

2.4.2 Outras Linguagens

Já foram propostas diversas linguagens, extensões e ferramentas que suportam a programação orientada por aspectos. Uma classificação imediata que pode ser feita entre essas propostas é pelo paradigma da linguagem de componente [Bre05]. Estes paradigmas correspondem aos principais existentes na programação de forma geral. Por isso, agrupamos as linguagens apresentadas desta forma.

2.4.2.1 Paradigma Imperativo

Atualmente, no desenvolvimento de sistemas, o paradigma imperativo tem sido o mais utilizado. Dentro desse contexto, inúmeras linguagens foram propostas baseadas neste paradigma. De forma análoga, existem diversas linguagens orientadas por aspecto que possuem com base o paradigma imperativo. Dentre elas pode-se destacar as linguagens

AspectJ (Seção 2.4.1), Eos, AspectC++, AspectWerkz, DAJ e Hyper/J.

Segundo o arcabouço conceitual, proposto por Chavez e Lucena [CL03], Eos e Hyper/J não oferecem suporte à POA. No entanto, Eos e Hyper/J introduzem novas construções à separação avançada de interesses. Além disso, o trabalho proposto baseou-se em algumas idéias apresentadas em Eos. Assim, dentre as linguagens supracitadas, ambas tiveram um maior destaque.

AspectC++ [SGSP, Asp06a] é uma implementação da programação orientada por aspectos em C++. Os conceitos básicos são bastante semelhantes aos de outras linguagens, como, AspectJ, que já se encontra bem mais difundida. Isto é uma tentativa de estimular o uso de AspectC++ por desenvolvedores já acostumados a trabalhar com AspectJ e que necessitem trabalhar com aplicações C/C++.

AspectWerkz [Asp06b] implementa transversalidades estática e dinâmica, além de permitir que o processo de combinação de código seja feito durante a compilação ou durante a execução. A sua implementação é flexível, permitindo que interesses transversais sejam implementados por meio de combinações de código Java com anotações e arquivos XML.

DAJ [OL03] é uma pequena extensão da linguagem AspectJ. DAJ estende AspectJ com os principais conceitos de Demeter, ou seja, grafo de classes, transversalidade de objetos e visitantes adaptativos (*adapter visitors*). Segundo a página do projeto [Dem06], um conceito importante de Demeter é separar um sistema em pelo menos duas partes: a primeira parte define os objetos e a segunda parte define as operações. O objetivo de Demeter é diminuir o acoplamento entre objetos e operações, de modo a possibilitar modificações a qualquer um sem impacto sério no outro. Isto reduzir significativamente o tempo de manutenção.

Similar a DAJ, há outras linguagens que utilizam os conceitos de Demeter, como, por exemplo, DemeterC++[Dem06], uma extensão para a linguagem C++, e DemeterJ [OL01], uma extensão para a linguagem Java.

Eos [RS03] é uma extensão da linguagem C# orientada por aspectos. Dentre as principais características de Eos, destacam-se (i) a unificação de classes e aspectos como *classpescts*. Um *classpesct* tem todas propriedades de uma classe, além de todas propriedades de um aspecto e (ii) o modelo unificado de Eos elimina o corpo do adendo com o objetivo de utilizar apenas métodos. Para isso, Eos provê uma nova construção para associar os pontos de junção capturados ao método que devem ser executado.

Como exemplo de utilização da linguagem Eos, a Listagem 2.3 redefine o aspecto AtualizaJanela, da Listagem 2.2. Convém destacar que AtualizaJanela é uma classe em Eos que possui um método atualizarJanela. Além disso, a linha 8 ilustra a nova construção que associa os pontos de junção coletados pelo conjunto de junção mudancaEstado ao método atualizarJanela.

```
1 class AtualizaJanela {
2     public void atualizarJanela() {
3         Janela.atualiza();
4     }
5     pointcut mudancaEstado() :
6         call(void ElementoFigura+.mover(..)) ||
7         call(* Ponto.set*(*)) || call(* Linha.set*(*));
8     after mudancaEstado () : atualizarJanela();
9 }
```

Listagem 2.3: Classe AtualizaJanela definida em Eos

Hyper/J [OT01a, OT01b] é uma ferramenta desenvolvida no Centro de Pesquisa T.J. Watson da IBM para oferecer suporte ao espaço de interesses(*Hyperspaces*). O espaço de interesses [TO00] permite: (i) a definição de quaisquer dimensões de interesses em qualquer estágio do ciclo de vida, (ii) encapsular os interesses dentro de cada uma de suas dimensões, (iii), identificar e gerenciar o relacionamento entre os interesses e (iv) a integração ou combinação dos interesses. Assim, Hyper/J oferece suporte à identificação, encapsulamento e integração de interesses a partir de programas em Java.

Um dos principais objetivos de Hyper/J era não modificar ou estender a linguagem Java. Em vez disso, Hyper/J oferece suporte à separação multidimensional de interesses para sistemas desenvolvidos em Java usando qualquer metodologia e ferramentas.

A separação multidimensional de interesses é uma evolução da Programação Orientada por Assunto. Tirelo et al. descrevem que "o objetivo desta tecnologia é quebrar a *tirania da decomposição dominante*, permitindo que cada requisito seja implementado em uma dimensão distinta, de modo que o ambiente de compilação faça a composição, de forma semelhante à feita em AspectJ" [TBBV04].

No desenvolvimento de um sistema, utilizando Hyper/J, o desenvolvedor deve fornecer três entradas [Cha04]:

- Um arquivo com o espaço de interesses.
  Este arquivo lista todas as classes Java com os quais o desenvolvedor está trabalhando, ou seja, o espaço com todos os interesses existentes. Assim, definem-se o nome do espaço de interesses e as unidades que pertencem a ele.

- Um ou mais arquivo de mapeamento de interesses.
  O(s) arquivo(s) de mapeamento associa(m) os interesses dispersos no espaço de interesses a determinadas dimensões (*hyperslices*). Por exemplo, a construção package JUnit : Teste.Unitario associa todas as classes do pacote JUnit a dimensão Teste e, nesta dimensão, todas as classes são agrupadas como um interesse nomeado Unitario.

- Um arquivo para integração das dimensões.
  O arquivo para integração das dimensões define a estratégia de composição entre as dimensões do espaço de interesses. Estas dimensões são compostas por meio de pontos de junção.

Como exemplo de uso, a Listagem 2.4 define o espaço de interesses necessário para implementar o aspecto mudancaEstado, da Listagem 2.2. As linhas 2 e 3 desta listagem definem que o espaço de interesses contém todas das classes dos pacotes com.exemplo.Visao e com.exemplo.Figura.

```
1 hyperspace FiguraHyperSpace
2     composable class com.exemplo.Visao.*;
3     composable class com.exemplo.Figura.*;
```

Listagem 2.4: Definição do espaço de interesses

Como supra explicado, tem-se ainda que definir o arquivo de mapeamento e o arquivo de integração. O arquivo de mapeamento está codificado na Listagem 2.5. Este arquivo define duas dimensões: (i)Principal, que contém o interesse ManipulaFiguras e (ii) Transversal, que contém o interesse ControleVisual. Observe que, na definição das dimensões e interesses, podem-se utilizar classes, operação entre outros tipos.

```
1     package com.exemplo.Figura : Principal.ManipulaFiguras;
2     operation com.exemplo.Visao.AtualizaJanela.atualiza
3         : Transversal.ControleVisual;
```

Listagem 2.5: Mapeamento dos interesses em dimensões

Por fim, a Listagem 2.6 define o arquivo de integração. As linhas 3 e 4 definem as dimensões que serão combinadas e nas linhas 5 a 11 são especificadas as estratégias de composição.

```
1  hypermodule AtualizaJanela
2      hyperslices:
3          Principal.ManipulaFiguras,
4          Transversal.ControleVisual;
5      relationships:
6          bracket "mover"
7              after Transversal.ControleVisual.Janela.atualiza;
8          bracket "Point"."set*"
9              after Transversal.ControleVisual.Janela.atualiza;
10         bracket "Linha"."set*"
11             after Transversal.ControleVisual.Janela.atualiza;
```

Listagem 2.6: Especificação da estratégia de composição

Conclusão: Dentre estas linguagens apresentadas, Eos influenciou o projeto da linguagem proposta. Assim, similar a Eos, a linguagem proposta utiliza o conceito de *classpect*, unificando no contexto da linguagem Machĭna, módulos e aspectos. Desta forma, pode-se concluir que a linguagem proposta não possui dicotomia baseada em aspectos, ou seja, não existe uma distinção clara entre módulos e aspectos, semelhante a Eos e Hyper/J.

Além disso, as novas construções propostas para a linguagem Machĭna são convertidas para código em AspectC++, como será apresentado no Capítulo 5.

2.4.2.2 Paradigma Funcional

Segundo Eichberg et al. [EMO04], a programação funcional permite que as declarações de conjunto de junção sejam curtas, precisas e declarativas. Além disso, destacam que esse tipo de declaração facilita a composição de conjuntos de junção. Eichberg et al. [EMO04] ainda destacam a importância de uma linguagem orientada por aspectos possuir construções que relacionam pontos de junção diferentes. Uma crítica realizada sobre as linguagens baseadas em AspectJ é que não existe este tipo de mecanismo. Por exemplo, considerando o programa manipulador de figuras, pode-se definir um conjunto de junção `p1` que captura todos os acessos as variáveis(*getters*) da classe `Ponto` que sejam realizados no fluxo de execução do método `mover` da classe `Linha`. Entretanto, não é possível combinar `p1` com um outro conjunto de junção `p2`, o qual receba `p1` como parâmetro, retornando o nome das variáveis capturadas por `p1`, e, então, selecionando um determinado conjunto de pontos de junção.

Com o objetivo de mostrar o valor e a praticabilidade dessas construções, os autores implementam um modelo de ponto de junção no qual os pontos de junção são nodos em uma árvore que representa a estrutura do programa. Tais pontos são capturados por consultas aos atributos desses nodos e podem ser passados como parâmetros para outras consultas. Para isso, utiliza-se um arcabouço que converte *bytecodes* Java em uma representação XML. Deste modo, pode-se expressar definições de conjunto de junção como consultas na linguagem XQuery. O XQuery é uma linguagem funcional, criada para permitir consultas em um meta-modelo da linguagem de marcação XML.

A meta-estrutura da representação gerada em XML para um programa é ilustrada na Figura 2.12. O nodo *all* representa o espaço global do programa. Desta forma, todas as classes são nodos filhos do nodo *all*. Os filhos do nodo *class*, por sua vez, definem a hierarquia de classes e interfaces, além dos campos e métodos. Um nodo *class* pode ter vários nodos *field* e *method*. Um nodo *field*, basicamente, define o nome e o tipo do campo. O nodo *method* representa a declaração de um método, e consiste em dois nodos, um para representar a assinatura do método e outro para representar

o código.

A listagem 2.8 mostra a representação XML gerada para o código Java da listagem 2.7. A representação XML define os mesmos métodos (linhas 9,15,16 e 22) e campos (linha 2 e 3) que o código-fonte. O construtor é representado pelo <init> do método (linha 5) com o tipo do retorno void (linha 6). O código do construtor e dos métodos foram omitidos por simplicidade.

Eichberg et al. [EMO04] destacam ainda que conjuntos de junção especificados como consultas sobre alguma representação de um programa têm alguns benefícios, dentre eles, destacam-se: (i) as consultas permitem escrever especificações precisas dos conjuntos de junção e (ii) os usuários podem estender a linguagem de definição de conjuntos de junção definindo seus próprios conjuntos de junção. Desta forma, torna-se possível criar bibliotecas de conjunção de junção de domínio-específico, por exemplo, para a sincronização, ou para otimização.

Como exemplo de uso, a Listagem 2.9 define o designador call encontrado em AspectJ. Para codificar o conjunto de junção mudancaEstado, da Listagem 2.2, tem-se ainda que definir uma função com uma funcionalidade similar ao símbolo "+". Desta forma, a Listagem 2.10 define essa função e a Listagem 2.11 representa o conjunto de junção mudancaEstado em XQuery.

Conclusão: No início do projeto da linguagem, considerou-se a existência de um compilador que convertia um programa em Machĭna para uma representação XML. Desta forma, as novas construções da linguagem relativas a aspectos seriam convertidas em consultas utilizando a linguagem XQuery. No entanto, esta idéia foi descartada com o objetivo de remover a representação XML e gerar um código executável na linguagem

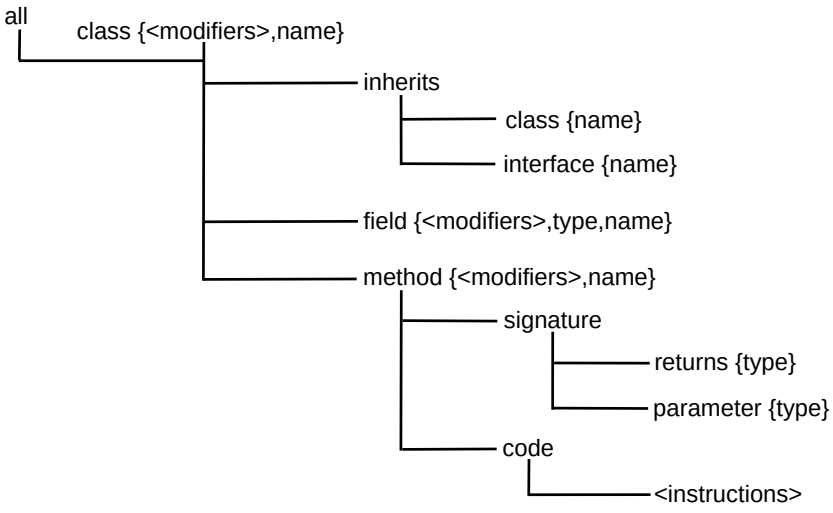

Figura 2.12: Meta-estrutura da representação gerada em XML para um programa

```
1  abstract class AbstractLine extends FigureElement {
2      Point p1 = new Point(); Point p2 = new Point();
3      FigureElement(){ super(); }
4      void setP1(Point point) {this.p1 = point;}
5      void setP2(Point point) {this.p2 = point;}
6      void moveBy(int x,int y){
7          this.p1.moveBy(x, y); this.p2.moveBy(x, y);
8      }
9      abstract void draw();
10 }
```

Listagem 2.7: Representação de um programa em Java

```
1  <class abstract="true" name="AbstractLine">
2      <inherits><class name="FigureElement"/></inherits>
3      <field type="Point" name="p1"/>
4      <field type="Point" name="p2"/>
5      <method name="<init>">
6          <signature><returns type="void"/></signature>
7          <code>...</code>
8      </method>
9      <method name="setP1">
10         <signature> <returns type="void"/>
11             <parameter type="Point"/>
12         </signature>
13         <code>...</code>
14     </method>
15     <method name="setP2"> as setP1 but setting p2 </method>
16     <method name="moveBy">
17         <signature> <returns type="void"/>
18             <parameter type="int"/><parameter type="int"/>
19         </signature>
20         <code>...</code>
21     </method>
22     <method abstract="true" name="draw">
23         <signature><returns type="void"/></signature>
24     </method>
25 </class>
```

Listagem 2.8: Representação da estrutura do programa em XML

C++ e AspectC++.

2.4.2.3 Paradigma Lógico

Diversos trabalhos foram propostos no sentido de utilizar a programação lógica como forma de especificação de conjuntos de junção. A sugestão parece surgir naturalmente da semelhança sintática e semântica entre o uso de conjuntos de junção e de fatos em uma base Prolog [Bre05].

Klose e Ostermann [KO05] propõem uma linguagem denominada Gamma e Oster-

```
1 declare function call($all element(), $meths as element()*)
2     as element()* {
3     $all//invoke[(@methodName = $meths/@name) and
4                  (@declaringClassName = $meths/../@name)]
5 }
```

Listagem 2.9: Designador call definido em XQuery

```
1 declare function subtypes($all as element()*, $types as element()*)
2     as element()* {
3     let $s := $all/class[./inherits//@name = $types/@name]
4         return  if (empty($s)) then
5                   $types
6                 else $types union subtypes($all, $s)
7 }
```

Listagem 2.10: Definição de uma função para capturar subtipos

```
1 let $subTipoFig := subtypes($all,
2     $all/class[@name="FigureElement"])/method[@name,"mover"])
3
4 let $mudancaEstado := call($all, $subTipoFig) union
5     $all/class[@name="Point"])/method[starts-with(@name,"set")]union
6     $all/class[@name="Linha"])/method[starts-with(@name,"set")]
```

Listagem 2.11: Conjunto de junção mudancaEstado definido em XQuery

mann et al. [OMB05] propõem uma linguagem denominada Alpha. Gamma e Alpha utilizam à linguagem lógica Prolog para definir o modelo de pontos de junção. O modelo de componentes é representado por uma linguagem orientada por objetos, a saber L2, um pequeno subconjunto de Java.

O modelo de pontos de junção de Gamma e Alpha baseia-se em um traçado (*trace*) da execução do programa em que cada evento está associado a um número seqüencial, chamado de *timestamp*. Todos os predicados utilizados têm como primeiro argumento um *timestamp*. Isto permite que se façam relações temporais entre pontos de junção para compor os conjuntos de junção.

A linguagem Alpha é similar à linguagem Gamma, exceto por não permitir referências a eventos no futuro, além de dispor de mais modelos de dados sobre os quais se pode compor os conjuntos de junção. Com o objetivo de prover uma linguagem de alto nível para o modelo de pontos de junção, Ostermann et al. propõem o uso de quatro diferentes e complementares fontes de informação: uma representação da árvore abstrata de sintaxe, uma representação do armazenamento de objetos (*heap*), uma representação do tipo estático de cada expressão do programa e uma representação do traçado (*trace*) da execução do programa.

Ostermann et al. [OMB05] apresentam-se diversas formas para definir o mesmo

conjunto de junção, para o exemplo do manipular de figuras geométricas. Desta forma, realiza-se uma comparação em termos da capacidade das definições resistirem a mudanças no programa-base.

Como exemplo de utilização da linguagem Alpha, a Listagem 2.12 redefine o aspecto AtualizaJanela, da Listagem 2.2.

```
1 class AtualizaJanela extends Object {
2     after now(ID), instanceof(P,'FigureElement'),
3           calls (ID, _, P, 'mover', _) , set(ID, _, _, P, _) {
4         Janela.atualiza();
5     }
6 }
```

Listagem 2.12: Aspecto AtualizaJanela definido em Alpha

Conclusão: Gamma e Alpha procuram permitir uma maior flexibilidade na definição dos conjuntos de junção. Dentro deste contexto, flexibilidade pode ser medida em duas dimensões: a granularidade e profundidade da informação a que se tem acesso na definição de um conjunto de junção, e a capacidade de combinação dessa informação.

Ao analisar estas linguagens, concluiu-se que ao aumentar a flexibilidade na definição do conjunto de junção aumenta, também, a sua complexidade. Além disso, avaliou-se que a usabilidade destes conjuntos de junção muito precisos é bem pequena. Desta forma, o modelo de ponto de junção da linguagem proposta ficou com uma flexibilidade semelhante ao modelo de AspectJ.

2.5 Considerações Finais

A programação orientada por aspectos, conforme revisada neste capítulo, é um novo paradigma de programação que tem por objetivo modularizar requisitos de um sistema, que por sua vez não podem ser modularizados por meio da programação orientada por objetos. A POA possibilita separar os interesses transversais do sistema dos interesses não transversais, melhorando, ou até mesmo eliminando, o espalhamento e o entrelaçamento de código.

Como principal ponto negativo, tem-se que a orientação por aspectos pode quebrar o encapsulamento de módulos, ao permitir acesso a detalhes de sua implementação. Se mal utilizada, pode afetar negativamente o funcionamento de módulos funcionais que já foram testados e homologados.

Conforme visto neste capítulo, não existe um consenso em relação ao que torna uma técnica orientada por aspectos. Diante deste contexto, apresentou-se uma teoria

informal sobre aspectos que será utilizada para avaliar a linguagem proposta neste trabalho.

Por fim, apresentaram-se algumas linguagens orientadas por aspectos. Dentre elas, AspectJ e Eos influenciaram diretamente o projeto da linguagem proposta. O modelo de ponto de junção utilizando é similar ao da linguagem AspectJ, porém adaptado para o contexto da linguagem Machĭna. Por exemplo, é indiferente realizar uma operação antes ou depois ao acesso uma função, visto que tudo ocorre paralelamente. Além disso, utilizou-se a mesma sintaxe para definir e combinar um conjunto de junção.

Similar a Eos, a linguagem proposta utiliza o conceito de classpect, unificando assim, no contexto da linguagem Machĭna, módulos e aspectos. Desta forma, pode-se concluir que a linguagem proposta não possui dicotomia baseada em aspectos. Assim, não existe uma distinção clara entre módulos e aspectos, semelhante a Eos e Hyper/J.

Algumas linguagens apresentadas, como Gamma e Alpha, procuram permitir uma maior flexibilidade na definição dos conjuntos de junção. No entanto, ao analisar estas linguagens, concluiu-se que ao aumentar a flexibilidade na definição do conjunto de junção aumenta, também, a sua complexidade. Além disso, avaliou-se que a usabilidade destes conjuntos de junção muito precisos é bem pequena. Desta forma, o modelo de ponto de junção da linguagem proposta ficou com uma flexibilidade semelhante ao modelo de AspectJ.

Capítulo 3

A Linguagem Machĭna

Este capítulo apresenta a linguagem Machĭna, uma linguagem de especificação formal baseada no modelo ASM. Inicialmente é apresentada a metodologia ASM, com objetivo de contextualizar o ambiente que a linguagem está inserido. Em seguida, são apresentadas as características desta linguagem bem como suas principais construções. Por fim, são apresentados dois exemplos que destacam um problema desta linguagem e que este trabalho propõe-se a solucionar.

3.1 Metodologia ASM

Börger [BS03] definiu que a metodologia ASM é constituída de três componentes: o conceito de Máquinas de Estado Abstratas(ASM), o Modelo Básico (do inglês, *Ground Model*), para especificação dos requisitos, e o método de refinamento para transformar um modelo básico em um código executável por meio de passos incrementais.

Máquinas de Estado Abstratas

ASM, introduzidas por Yuri Gurevich, provêem recursos expressivos para especificar a semântica operacional de sistemas dinâmicos discretos, em um nível de abstração natural e de uma maneira direta e essencialmente livre de codificação. Com isso, tem-se por objetivo diminuir a distância que há entre modelos formais de computação e métodos práticos de especificação. Para tal, definem-se máquinas de estado que simulem passo a passo o algoritmo.

ASM constitui um formalismo no qual um estado é atualizado em passos de tempo discretos. Diferentemente da maioria dos sistemas baseados em estado, o estado é dado por uma álgebra, que é uma coleção de funções e universos. As transições de estado são dadas por regras que atualizam as funções pontualmente e estendem os universos com novos elementos.

Modelo Básico e Método de Refinamento

Métodos padrões de engenharia de software definem linguagens de projeto de propósito geral para a descrição de componentes. Devido à sua generalidade, o uso destas linguagens requer o conhecimento técnico de engenharia de software, sendo, desse modo, inapropriadas para serem usadas por não-especialistas. Este fato impede que os especialistas do problema (domínio da aplicação) possam contribuir de forma ativa durante o processo de desenvolvimento de software.

Uma dificuldade inerente a especificação de um sistema é decidir o nível de detalhes e, por conseguinte, grau de formalidade, que é apropriado para o nível do sistema planejado. Quanto maior o grau de detalhamento, maior a dificuldade de se entender e validar a especificação. Por outro lado, a omissão de alguns detalhes pode não capturar características pertinentes ao sistema.

Dentro deste contexto, Börger destaca que o nível de classes descritas por UML [RBJ99] impõe uma baixa abstração para ser utilizado no início da estruturação dos modelos, existindo assim, o risco de uma explosão conceitual de classes [BS03]. Além disso, torna-se difícil modelar características que se relacionam as múltiplas classes e que freqüentemente estão espalhadas na hierarquia de classe, por exemplo, geração de registro de operações (*logging*), autenticação de serviços e segurança.

Segundo Börger, a metodologia ASM utiliza conceitos simples e bem conhecidos, o que facilita a leitura e a escrita de especificações de sistemas [BS03]. A liberdade que ASM oferece para modelar objetos e operações, abstraindo de detalhes irrelevantes, permite isolar a principal parte do sistema em um modelo preciso e que expõem toda complexidade do sistema, conservando a simplicidade para que o mesmo possa ser entendido e analisado facilmente. Este modelo, chamado de Modelo Básico[Bör03], pode servir como uma base para contrato entre o cliente ou especialista do problema e o desenvolvedor.

O Modelo Básico representa o primeiro nível de uma especificação. Com isso, pode-se controlar o nível de abstração utilizado, obtendo-se um modelo mais simples e preciso. Desta forma, o Modelo Básico permite uma maior participação dos especialistas do problema antecipando, para fase de projeto, alguns dos possíveis erros existentes.

Aplicando-se métodos de refinamento a um Modelo Básico pode-se obter o programa final. Um método de refinamento consiste em um processo que a partir da aplicação de regras bem formadas, que substituem construções abstratas por outras com maiores detalhes, pode-se obter um código executável. As principais características que tornam o método de refinamento bastante utilizado são: (i) o fato de a verificação estar inclusa no processo e (ii) a possibilidade de testar se o sistema realmente realiza as tarefas que lhe foram designadas.

Linguagens para especificação de Modelos Básicos

Já foram propostas algumas linguagens para especificação de Modelos Básicos. Como exemplo, pode-se citar as linguagens AsmL (Abstract State Machine Language) [GRS05], XASM (Extensible ASM) [Anl00], AsmGofer [Sch01] e Machĭna [BTI+07].

O decorrer deste capítulo explica as principais construções da Linguagem Machĭna, foco principal deste trabalho.

3.2 A Linguagem Machĭna

A linguagem Machĭna foi criada pelo grupo de pesquisa do Laboratório de Linguagens de Programação do DCC/UFMG, e atualmente se encontra em sua versão 2.0 [BTI+07]. Trata-se de uma linguagem baseada no modelo de máquinas de estado abstratas. Apesar da existência de outros ambientes baseados em ASM à época de sua criação, a linguagem Machĭna se justificou pelos seguintes motivos, entre outros:

- adaptação da linguagem às necessidades do grupo de pesquisa, fazendo-se modificações de caráter experimental para investigar algum conceito em particular;
- possibilidade de criação de novas construções de modo a facilitar a especificação
- dificuldade de obtenção da documentação completa e precisa das linguagens existentes.

As características principais de Machĭna são:

- estruturas para modularização e mecanismos de visibilidade e proteção;
- extensibilidade de tipos;
- sequenciadores de regras;
- sistema fortemente tipado, com um rico conjunto de tipos pré-definidos;
- invariantes para a execução da regra de transição da máquina abstrata;
- regras de transição de estado;
- multiagentes, com capacidade de autonomia, independência, consciência de contexto sociabilidade, introduzida na linguagem de maneira simples e direta;
- abstração de regras de transição que podem, por exemplo, ser executadas a partir de outras máquinas, criando a noção de submáquinas.

Os seguintes identificadores são declarados como palavras reservadas de Machĭna:

abstractions	**action**	**active**	**agent**	**Agent**	**algebra**
all	**anew**	**and**	**as**	**begin**	**blocked**
Bool	**case**	**Char**	**choose**	**create**	**default**
derived	**destroy**	**destroyed**	**dispatch**	**do**	**dynamic**
else	**elseif**	**end**	**ensure**	**enum**	**exists**
external	**false**	**file**	**forall**	**if**	**import**
include	**in**	**initial**	**inout**	**input**	**Int**
interface	**invariant**	**is**	**let**	**list**	**List**
loop	**machina**	**module**	**not**	**of**	**or**
old	**otherwise**	**out**	**Output**	**public**	**Real**
return	**require**	**rule**	**satisfying**	**select**	**self**
set	**Set**	**shared**	**state**	**State**	**static**
step	**stop**	**stopped**	**String**	**then**	**true**
tuple	**type**	**undef**	**with**	**xor**	**Promise**

O restante deste capítulo apresenta as construções da linguagem Machĭna, sendo que uma descrição mais detalhada pode ser encontrada em [BTI+07]. Assim, as principais construções da linguagem Machĭna foram organizadas em: unidades de especificação, sistema de tipos, funções e relações, abstrações de regras, regras de transição de estado, expressões e administração de agentes.

3.2.1 Unidades de Especificação

Basicamente, uma especificação em Machĭna consiste em um conjunto de *unidades de especificação*, as quais podem ser módulos de definição Machĭna, módulos de programa, ou definições de interface de agentes.

Módulos de Definição Machĭna

Associados a uma especificação, pode haver um ou mais *módulos de definição Machĭna*, os quais criam e disparam os principais agentes que executam de forma autônoma as regras de transição de estado dos módulos.

Um módulo de definição de Machĭna consiste na especificação de seu nome e de diretivas para criação dos agentes e disparo de sua execução. A partir deste módulo, o sistema de execução Machĭna cria um agente especial, denominado **superagente**, que executa as diretivas especificadas no módulo Machĭna, dando início ao processo de criação e execução dos agentes.

A diretiva **agent of** `M` declara um agente a partir do módulo `M`. O vocabulário do agente é formado por uma cópia própria de todas as declarações contidas no módulo **M** e nos módulos a partir dele incluídos via a diretiva **include**. A interação entre agentes é possível mediante troca de mensagens, com interveniência do superagente, e por meio de funções compartilhadas. Cabe ressaltar que funções compartilhadas, ou seja, declararadas como **shared**, não são duplicadas pela diretiva **include**. Essas funções pertencem ao vocabulário comum a todos os agentes, e consequentemente, não pode haver colisão de nomes de funções compartilhadas. Funções compartilhadas ocorrem na lista de inclusão de cláusula **include** apenas para fins de controle de visibilidade.

Após a declaração de todos os agentes especificados no módulo de definição Machĭna, esses são criados e a execução de cada um, em paralelo, é simultânea e automaticamente iniciada, e cada um passa a executar repetidamente sua regra de transição. Cada ciclo de execução da regra de transição de estado de um agente é atômico, sem qualquer tipo de interrupção ou bloqueio. Somente entre uma execução da regra da transição e a seguinte é que agentes podem ser bloqueados, por exemplo, por falta de recursos solicitados a outros agentes.

A Listagem 3.1 apresenta um exemplo de um módulo de definição Machĭna. Este exemplo cria um agente do módulo Producer e três agentes do módulo Reader, todos independentes entre si.

```
1 machina Host
2     agent of Producer;
3     agent of Reader(3);
4 end Host
```

Listagem 3.1: Exemplo de um módulo de definição Machĭna

Módulos de Programa

Um módulo de programa especifica a regra que um agente a ela associado executa, seu vocabulário, i.e., conjunto de símbolos que manipula, interpretação destes símbolos no estado inicial e o invariante da execução.

Um módulo contém um mecanismo de controle de visibilidade, permitindo organizar o vocabulário de um agente em unidades encapsuladas, reduzindo sua complexidade. Assim, um módulo pode incorporar no vocabulário de seus agentes os elementos (declarações de funções, tipos e ações) definidos em outros módulos. A primeira parte de um módulo consiste na enumeração das interfaces que ele importa. Depois vem a lista dos módulos a serem incluídos. Quando um módulo é incluído, todos os seus símbolos públicos tornam-se visíveis. A Listagem 3.2 apresenta a forma de um módulo de programa.

```
1  module module-name
2      import elementos importados
3      include elementos incluídos
4      algebra:
5          elementos declarados(funções e tipos)
6      abstractions:
7          declaração de abstrações(ações)
8      initial state:
9          inicializações de funções dinâmicas
10     transition:
11         regras de transição de estado
12     invariant:
13         invariante de execução
14 end module-name
```

Listagem 3.2: Estrutura de um módulo em Machĭna

A parte **import** coloca no escopo do módulo a interface dos agentes com os quais o agente do módulo deseja-se comunicar. A interface de um agente contém as assinaturas de abstrações de regras que o módulo do agente torna público para uso de outros agentes.

A parte **include** define os módulos secundários e seus elementos que devem ser incorporados ao módulo. Esta cláusula tem importante papel na formação do vocabulário dos agentes. A cláusula **include** também serve para controle de visibilidade de elementos declarados públicos ou compartilhados em um módulo.

A seção **algebra** define os elementos da álgebra subjacente ao modelo, contendo os *sorts* ou tipos e as funções do módulo. A seção **initial state** serve para inicializar essas funções dinâmicas. A seção **abstrations** define abstrações de regras de transição, que podem ser usadas localmente ou exportadas.

A seção **transition** define a regra de transição de estado do agente, a qual é executada repetidamente quando o agente é disparado. Esta seção representa o corpo do programa dos agentes associados ao módulo. Por fim, a seção **invariant** define a condição envolvendo os elementos de um módulo que deve ser invariante durante sua execução. Entre duas iterações da regra de transição do módulo, o invariante é verificado pelo sistema de execução Machĭna. Caso seja violado, uma mensagem de erro é emitida e a execução, interrompida.

Interface de Agentes

Interface é um recurso para se definir um canal de comunicação através do qual um agente pode interagir com outros agentes. A interface tem o mesmo nome do módulo principal que forma o agente, e nela são relacionados os nomes dos tipos, assinaturas de regras e abstrações exportados pelo módulo correspondente, isto é, aqueles elemen-

tos que podem ser usados por outros agentes para transmitir informação ao agente associado à interface ou dele solicitar execução de serviços.

As assinaturas das abstrações especificadas na interface têm informações redundantes em relação às suas respectivas declarações no módulo correspondente. Toda abstração cuja assinatura ocorre em alguma interface deve ser declarada pública e seus parâmetros devem possuir indicação explícita de sua natureza (**in**, **out** ou **inout**), o qual é usado no processo de comunicação interagentes. A Listagem 3.3 apresenta a definição do módulo Philosopher, com sua respectiva interface.

```
1  interface Philosopher
2    action setFork(in f: Int);
3  end
4
5  module Philosopher
6    ...
7    abstractions:
8      public action setFork(f: Int) is
9        ...
10     end
11   ...
12 end
```

Listagem 3.3: Definição de uma interface para o módulo *Philosopher*

Interfaces podem ser importadas, via a cláusula **import**, por outros módulos. Diferentemente do processo de inclusão, feito via **include**, o vocabulário da interface importada **não** é duplicado no módulo importador. O mecanismo de importação apenas estabelece um canal de comunicação. A Listagem 3.4 importa a ação setFork definida no módulo Philosopher.

```
1  module Host
2    import Philosopher(setFork);
3    ...
4  end
```

Listagem 3.4: Exemplo de importação de um módulo

Uma descrição mais detalhada sobre os agentes e sua comunicação é realizada na Seção 3.2.7, sendo que maiores detalhes podem ser obtidos na especificação formal da linguagem Machĭna[BTI+07].

3.2.2 Sistema de Tipos

As expressões de tipos de Machĭna são as seguintes:

- tipos básicos: **Bool**, **Char**, **Int**, **Real**, **String** e **Promise**;

- tipos compostos: listas, agentes, tuplas, conjuntos, uniões disjuntas, enumerações, funcionais e tipos definidos pelo programador;
- tipos genéricos: listas genéricas (**List**), conjuntos genéricos(**Set**), agentes genéricos (**Agent**), tipo `?` e a união disjunta de todos os tipos;
- tipos arquivos: arquivos *stream* (**Input** e **Output**) e arquivos binários (**file of** T).

Os tipos acima se excetuando **Promise** têm o significado usual e estão definidos em [BTI+07]. O tipo **Promise** denota o tipo de uma função que contém o estado de uma execução assíncrona.

3.2.3 Funções e Relações

Funções e relações são os elementos fundamentais do vocabulário de uma especificação na linguagem Machĭna. Os parâmetros de uma função podem ser de qualquer tipo, excetuando-se funções de ordem mais alta e abstrações de regras.

As funções, quanto sua natureza, podem ser:

- estáticas – são as que não podem sofrer atualizações e nem acessar funções dinâmicas, derivadas ou externas. São identificadas pelo modificador **static**;
- derivadas – são funções que não podem sofrer atualizações, mas podem acessar funções dinâmicas, derivadas ou externas. São identificadas pelo modificador **derived**.
- externas – são funções definidas e atualizadas no ambiente externo. São identificadas pelo modificador **external**.
- dinâmicas – são as funções identificadas pelo modificador **dynamic** e aquelas sem modificadores de classe, que são assumidas dinâmicas por padrão.

Na definição de uma função estática ou dinâmica, define-se sua assinatura e, se desejado, também o seu valor inicial. No exemplo abaixo, a função `f` é uma função dinâmica e `g` uma função estática, ambas com assinatura dada por **Char**×**Char** → **Int** e valor inicial $\lambda(x, y).1$.

Quando uma função estática ou dinâmica é declarada sem um valor de inicialização explícito, seu valor inicial será o valor padrão do seu tipo de saída (contradomínio). No exemplo abaixo, como a função `h` não recebe valor inicial, ela será definida por `h` $= \lambda xy.0$, pois 0 é o valor padrão de **Int**. Por outro lado, definições de funções derivadas possuem obrigatoriamente o corpo para ser avaliado. No exemplo abaixo,

```
1  module M
2      ...
3      algebra:
4          external square:Int-> Int;
5          f(a:Char, b:Char):Int:= 1;
6          static g(x:Char, y:Char):Int:= 1;
7          static h(x:Char, y:Char):Int;
8          derived mysquare(x:Int):Int:= square(x);
9      ...
10 end
```

Listagem 3.5: Exemplos da declaração de funções e relações

definimos a função `mysquare`, que retorna o quadrado de um número inteiro, usando a função externa `square`.

Um aspecto importante que deve ser modelado na especificação de um sistema é que, em geral, sistemas são afetados pelo ambiente. O modelo ASM supõe que o ambiente se manifeste por meio de funções $e_1, \ldots, e_k$, denominadas *funções externas*. Uma função externa é uma função definida externamente a qualquer módulo Machĭna. Como o seu valor é definido pelo ambiente externo, elas não possuem um corpo para ser avaliado. Desta forma, a definição de uma função externa contém somente a especificação de sua assinatura, como `square` em **external** `square`: **Int** → **Int**;.

3.2.4 Abstrações de Regras

Abstrações de regras são um recurso apropriado para se definir operações de tipos abstratos de dados. Uma ação pode receber qualquer tipo de parâmetro, inclusive outras ações. O corpo de uma ação pode ser uma execução repetitiva, se marcado por **loop:** ou execução unitária, quando marcado com **begin:**. Na ausência das marcas **loop:** ou **begin:**, procede-se com execução unitária.

Durante a execução, as eventuais atualizações têm efeito apenas local, inclusive para os parâmetros e funções dinâmicas globais que ocorrerem no corpo da abstração. As atualizações ocorridas na abstração somente são percebidas pela regra de transição do agente no passo seguinte, da mesma forma que ocorre com uma regra de atualização de módulo.

Para alcançar este efeito, a primeira providência de uma ação repetitiva, marcada com **loop:**, é fazer uma cópia local de todos os parâmetros recebidos e de todas as funções dinâmicas globais atualizadas no seu corpo. Funções dinâmicas com acesso somente de leitura no corpo da abstração não são copiadas. A regra de transição das ações repetitivas opera diretamente com estas cópias locais e com as funções globais só de leitura, e no fim da ação definida pela abstração, a lista de atualização obtida para

as cópias locais são refletidas na lista de atualização corrente do ponto de chamada da abstração. Do ponto de vista do módulo que faz a chamada, tudo se passa como se as alterações geradas pela ação houvessem ocorrido em um único passo.

No caso de ação sem a marca **loop:**, a cópia local citada acima não se faz necessária. Neste caso, executa-se uma única vez a regra de transição no corpo da abstração e retorna-se ao ponto de chamada com a lista de atualizações produzidas para parâmetros e funções globais à ação. Neste caso, diferentemente das iterações, não se fazem cópias das funções globais atualizadas na abstração.

Os modos de passagem de parâmetros para um ação são **in**, **out** e **inout**. Se omitido a especificação do modo de passagem, assume-se **inout**. A passagem de parâmetros para abstrações **out** ou **inout** é por nome (*call by name*). Desta maneira, as atualizações destes parâmetros no corpo da abstração são refletidos nos argumentos de chamada. Os parâmetros do tipo **in** são passados por valor.

Os elementos declarados locais a uma abstração sobrevivem de uma chamada à seguinte, preservando, portanto, seus valores finais entre uma chamada e a seguinte. Na primeira execução da ação, os valores iniciais dos elementos locais são os valores padrões de seus tipos. As inicializações de funções contidas na declaração, se houver, são executadas sempre que a abstração for chamada, reiniciando seus valores; caso contrário, os valores assumidos no fim da última chamada estarão disponíveis.

Ao corpo de uma ação pode ser associado um contrato, que estabelece a pré-condição e a pós-condição da ação. A pré-condição define as condições que devem existir antes de se iniciar a execução da regra de transição da ação. A pós-condição estabelece a condição a ser satisfeita quando a ação finalizar a execução de seu corpo. Pré-condição ou pós-condição não satisfeitas denotam execução inválida.

3.2.5 Regras de Transição de Estado

A transição principal de um módulo ou do corpo de abstrações definidas em um módulo pode ser uma regra de transição de um único passo ou então de uma seqüência de passos de execução. No caso de transição de um único passo, a mesma regra é executada repetidamente pelo agente até que a condição de terminação seja atingida. Na transição de múltiplos passos, cada um dos passos indicados pelas cláusulas **step** é executado em sequência, começando-se com **step := 1**, sendo **step** uma variável pré-declarada com o tipo **Int**. O valor de **step** não pode ser alterado por atribuição direta. Para sua alteração dentro da regra de transição, há uma outra variável implícita inteira pré-definida, denominada **next**, cujo valor inicial é 1.

No início da execução de cada passo, essa variável **next** é automaticamente incrementada, e no fim de toda transição, faz-se automaticamente **step** := **next**. A próxima

transição a ser executada é escolhida comparando-se o valor corrente da variável **step** com a constante inteira que rotula cada uma das regras de cada passo. Se não houver um rótulo especificado para um dado passo, nada é feito, exceto o incremento de **next** e atualização de **step**, e o passo seguinte é iniciado. O valor de **step** pode ser explicitamente alterado em um passo, atribuindo-se um valor à variável **next**, o qual então prevalece sobre o incremento automático, permitindo que se force a re-execução da regra a partir de determinado passo.

As regras de transição de Machĭna podem ser dos seguintes tipos: (i) Regras Básicas: Atualização, Blocos e Abreviaturas; (ii) Regras Condicionais: If, Case, With e Choose (iii) Regras de Universalização; (iv) Chamadas de Abstração; (v) Regra de Asserção.

3.2.5.1 Atualização de Funções

A regra de atualização é formada por um identificador, que deve ser um nome de função declarada como dinâmica, seguido de seus argumentos, os quais são usados para a determinação do endereço da atualização, e uma expressão, cujo valor é atribuído ao endereço formado. Funções estáticas, derivadas e externas não podem ser atualizadas.

Por exemplo, uma regra de atualização pode ser da forma $f(x_1, \cdots, x_n) := y$, onde f é uma função dinâmica com assinatura $f : T_1 \times \cdots \times T_n \rightarrow T$, cada x_i é uma expressão cujo tipo é T_i, $1 \leq i \leq n$, e y é uma expressão cujo tipo é T.

3.2.5.2 Bloco de Regras

A regra bloco é composta por uma seqüência de regras, separadas por ponto-e-vírgula, que devem ser executadas simultaneamente.

3.2.5.3 Abreviatura de Termos

A regra *let* é uma regra especial, que não é de transição, e tem a forma:

let $x_1 = e_1$;
let $x_2 = e_2$;
...
let $x_k = e_k$;

onde, para $1 \leq i \leq k$, x_i são identificadores e e_i são expressões. Seu efeito é avaliar cada uma das expressões e_i e continuar a execução da regra de transição em um ambiente onde cada x_i está imediatamente associado à respectiva expressão e_i.

A Listagem 3.6 declara os identificadores `a` e `b`, cujo escopo é toda a regra a partir do ponto de definição do **let**.

```
1 let a = head(z);
2 let b = head(tail(z));
3 x := a :: b;
```

Listagem 3.6: Exemplo da regra **let**

3.2.5.4 Regra if

A regra *if* tem a forma

$$\textbf{if } g_1 \textbf{ then } R_1 \textbf{ elseif } g_2 \textbf{ then } R_2 \cdots \textbf{elseif } g_k \textbf{ then } R_k \textbf{ else } R_{k+1} \textbf{ end},$$

onde g_i, $1 \leq i \leq k$, são expressões booleanas denominadas guardas e R_i, $1 \leq i \leq (k+1)$, são regras de transição. Seu efeito é avaliar as guardas g_i na ordem em que aparecem e, quando alguma delas for verdadeira, a regra correspondente é executada e o restante da regra é ignorado. Se nenhuma das guardas for verdadeira, então a regra R_{k+1} é executada.

3.2.5.5 Regra case

A regra *case* tem a forma:

case e
  of $e_1 \Rightarrow R_1$; **of** $e_2 \Rightarrow R_2$; $\cdots$ **of** $e_k \Rightarrow R_k$;
  otherwise $\Rightarrow R_{k+1}$;
end

onde e é uma expressão de algum tipo discreto (**Bool**, **Char**, **Int** ou alguma enumeração), e_i, $1 \leq i \leq k$, são expressões constantes ou manifestas do mesmo tipo de e, e R_i, $1 \leq i \leq (k+1)$, são regras de transição. Seu efeito é avaliar a expressão e e executar a primeira regra R_i para a qual $e_i = e$. Se para todo valor de i, $e_i \neq e$, então executa-se R_{k+1}. Como exemplo tem-se a Listagem 3.7.

```
1 case head(z)
2   1 => x := 11
3   3 => x := 17
4   4 => x := 19
5   otherwise => x := 23
6 end
```

Listagem 3.7: Exemplo da regra **case**

3.2.5.6 Regra *with*

Uma regra *with* é uma forma especial de *case*, em que, em vez de compararmos o valor de uma expressão com expressões manifestas, comparamos o nome do tipo de uma expressão com os nomes de tipos relacionados nas cláusulas da regra *with*.

A regra *with* tem a forma:

with e
  as $x_1 : T_1 => R_1(x_1);$
  $\cdots$
  as $[a\ b\ c\ \cdots] => R_2(a, b, c);$
  $\cdots$
  as $(a, b, c, \cdots) => R_k(a, c);$
  otherwise $=> R_{k+1};$
end

onde e é uma expressão cujo tipo normalmente é de uma união disjunta, x_i, $1 \leq i \leq k$, são identificadores, T_i, $1 \leq i \leq k$, são nomes de tipos, e R_i, $1 \leq i \leq (k+1)$, são regras de transição. Seu efeito é avaliar a expressão e e executar a primeira regra R_i em que tipo de e seja T_i. O ambiente em que R_i é executado contém o identificador x_i associado ao valor de e. Se para todo valor de i, o tipo de e for diferente de T_i, então executa-se R_{k+1}.

3.2.5.7 Regra *choose*

A regra *choose* tem a forma: **choose** $e_1, e_2, ..., e_k$ **satisfying** g **do** R **end**, onde as expressões $e_1, e_2, ..., e_k$ e R são da mesma forma que na regra *forall* e g é uma expressão booleana. O seu efeito é escolher não-deterministicamente elementos dos domínios dados que satisfaçam a guarda `g` e executar a regra dada, associando os nomes de identificadores em $e_1, e_2, ..., e_k$ aos elementos escolhidos.

Cabe mencionar que as expressões $e_1, e_2, ..., e_k$ são do tipo coleção de valores, ou seja, um conjunto ou uma lista.

3.2.5.8 Regra de Universalização

A regra *forall* tem a forma: **forall** $x_1 : U_1, \cdots, x_k : U_k$ **do** R **end**, onde x_i são identificadores, U_i são expressões que denotam coleção de valores, isto é, conjuntos ou listas, e R, uma regra de transição e g uma expressão booleana. Seu efeito é criar diversas instâncias da regra `R`, uma para cada valor possível de $e_1, e_2, ..., e_k$ nos domínios dados e executá-las paralelamente. Cada instância é executada em um ambiente em que os identificadores x_i estão associados aos valores atribuídos na instância da regra.

Por exemplo, no trecho de código abaixo, a codificação do lado esquerdo é equivalente ao lado direito.

```
f(Int,Int):Int;                 f(Int,Int):Int;
transition:                     transition:
    forall x:1..2,y:1..3 do         f(1,1) := 2; f(1,2) := 3;
        f(x,y) := x + y             f(1,3) := 4; f(2,1) := 3;
    end                             f(2,2) := 4; f(2,3) := 5;
end                             end
```

3.2.5.9 Regra de Chamada de Abstrações

Uma regra de tipo *chamada de abstração* gera um conjunto de atualizações, que é incluído no conjunto de atualizações existente no ponto de chamada.

Os argumentos de uma abstração que por ela podem ser atualizados devem ser funções dinâmicas ou expressões que avaliem em algum endereço para atualização. Os tipos dos parâmetros formais da abstração de regra devem ser compatíveis com os tipos dos respectivos argumentos.

A chamada de abstrações pode ser síncrona ou assíncrona, conforme explicado na Seção 3.2.7.

3.2.5.10 Regra de Asserção

A expressão da regra *require* é avaliado com os valores correntes das funções durante uma transição. Se o valor obtido for verdadeiro, a execução da transição procede normalmente. Do contrário, a execução é interrompida, com uma condição de erro.

A regra *ensure* é similar à *require*, exceto que os valores usados para as funções que ocorrem em sua expressão são os obtidos após o fim da transição, portanto com valores atualizados pela transição executada.

3.2.6 Expressões

As expressões de Machĭna são as seguintes: expressões sobre os tipos básicos, agregados, conversão de tipos e uma combinação de operadores e expressões mais simples. Dentre essas se destacam:

agregados – os agregados são listas, tuplas, conjuntos ou nodos. Os agregados do tipo lista ou conjunto denotam coleções de elementos de mesmo tipo.

aplicação de funções – a aplicação de função tem a forma geral *nome de função (argumentos)*, onde *argumentos* é uma lista de expressões, cujo comprimento é

igual à aridade da função da aplicação. Se a aridade da função for igual a zero, então não se utilizam os parênteses, somente o nome da função. Os tipos dos argumentos devem ser equivalentes aos tipos esperados como parâmetros.

if – na expressão **if**, as condições indicadas devem ser expressões booleanas e as expressões que ocorrem nas partes **then** e **else** devem ter tipos estruturalmente equivalentes.

case – o argumento de uma expressão **case**, isto é, a expressão após a palavra **case**, pode ser de qualquer tipo que tenha a operação de comparação por igual. A cláusula **otherwise** é obrigatória se as cláusulas da expressão não cobrirem todas as possibilidades de valores do argumento. As expressões que antecedem as setas (=>) devem ter tipo equivalente ao tipo do argumento. As expressões após as setas (=>) e a da cláusula **otherwise** devem ter tipos compatíveis entre si.

with – a expressão **with** é semelhante a expressão case. A principal diferença é que a comparação para escolha da alternativa a ser executada se baseia não no valor da expressão avaliada, mas sim no seu tipo. Assim, na avaliação da expressão **with**, o tipo do argumento é testado em seqüência contra o tipo indicado em cada uma das cláusulas até que se encontre o primeiro tipo que seja, por equivalência nominal, uma projeção do tipo do argumento. Isto ocorrendo, o valor do argumento é associado à variável da cláusula, e a expressão que segue a respectiva seta (=>) dá o valor da expressão **with**. Caso contrário, o valor é dado pela expressão da cláusula **otherwise**.

exist – a expressão **exist** permite determinar se uma coleção contém valores que satisfazem uma dada condição.

all – a expressão **all** informa se todos os elementos de uma coleção de valores satisfazem uma dada condição.

promise – uma expressão promise, definida pela sintaxe **dispatch** `call-rule`, serve para disparar a execução de uma chamada de abstração, em geral de um outro agente, que deve ser executada em paralelo em relação a execução do agente que a originou.

Os operadores de Machĭna são mostrados na tabela a seguir. A ordem de apresentação mostra a precedência dos operadores, sendo que os de precedência mais alta estão na primeira linha e os de precedência mais baixa estão no fim.

`old`	valor da função operando antes da execução da ação
`::`	construção de lista
`in`	pertinência a conjuntos
`- + not`	unário para inversão de sinal ou de valor lógico
`* / % and`	multiplicação, divisão, resto
`+ - or xor`	soma ou subtração
`..`	constrói conjunto por intervalo
`= != < > <= >= is`	operadores de relação

3.2.7 Administração de Agentes

Para administrar a execução dos agentes da máquina abstrata, são utilizadas as regras *create, dispatch, stop, return* e *destroy.*

A regra *create* tem a forma: **create** x_1**,** $\cdots$**,** x_k, onde x_i são identificadores declarado como sendo do tipo **agent of** M, onde M é o nome do módulo associado a cada um dos agentes. Seu efeito é criar k agentes, cada um devendo, quando disparados, executar repetidamente a regra de transição do módulo M_i.

A regra *create* pode criar múltiplos agentes de um mesmo módulo, como ocorre na regra `create m(10)`, onde `m:agent of M`, a qual cria 10 agentes de tipo `M`. A identificação `m` dos agentes criados é um mapeamento do tipo `Int-> agent of M`, com domínio no intervalo de 1 ao número de agentes criados, o qual é dada pela função pré-definida `numberOfAgents`. Assim, no exemplo dado, `numberOfAgents(M)` retorna o valor 10. A regra `create m` é uma abreviatura de `create m(10)`

A regra *create* não inicia a execução do agente, a qual deve ser feita explicitamente via a regra *dispatch*, mas um agente criado está apto a executar pedidos a ele encaminhados por outros agentes.

A regra *stop* termina a execução do agente que está executando no momento, assim que o passo atual for terminado. A execução do agente chega ao fim após o disparo das atualizações do passo.

A regra *return* é semelhante à regra *stop*. Ela só pode aparecer dentro do corpo de uma abstração de regra, e seu objetivo é terminar a execução da ação, voltando o controle ao ponto de chamada.

A regra *destroy* tem a forma **destroy** a, onde a é uma expressão de tipo agente. Seu efeito é avaliar a expressão a, obtendo a referência para um agente e terminar a execução deste agente. Se o agente a ser destruído for o agente que está executando no momento, então a sua execução será terminada após os disparos das atualizações do passo, da mesma forma que na execução da regra *stop*.

Todo agente tem um atributo implícito state, do tipo State que indica o estado corrente do agente. O tipo pré-definido **State** é equivalente a State = enum {anew, active, stopped, blocked, destroyed} e representa os possíveis estados de um agente, a saber:

- **anew** - agente criado, mas ainda não disparado;
- **active** - agente em execução e não bloqueado;
- **stopped** - agente que encerrou a execução de sua regra de transição, mas ainda apto a responder requisições de outros agentes e pode ser redisparado;
- **blocked** - agente em execução, mas bloqueado à espera de alguma mensagem;
- **destroyed** - agente destruído, incapaz até de responder requisições de outros agentes e de ser redisparado.

Comunicação Entre Agentes

Os agentes em execução comunicam-se via chamadas a abstrações de regras (ações) que são declaradas nas interfaces dos módulos principais dos agentes. Durante a execução de sua regra de transição, um agente a pode solicitar a execução **síncrona** ou **assíncrona** de uma abstração de regra disponibilizada via sua interface por um outro agente b.

Execução Síncrona: uma chamada a uma abstração da forma `b.g(`a_1`,...,`a_n`)` é um pedido de uma execução síncrona e deve ser entendido como o envio da mensagem `g(`a_1`,...,`a_n`)` pelo agente solicitante a ao agente solicitado b, requisitando a execução do serviço definido pela abstração chamada, cujos parâmetros servem para transmitir informações ao agente b ou dele recebê-las. Estas chamadas são ditas síncronas porque a próxima transição do agente a somente será executada quando os seus pedidos tiverem sido atendidos. Entretanto, estas chamadas a abstrações de regras de outros agentes são **assíncronas** em relação à execução da transição corrente, isto é, o envio de mensagens não causa bloqueio do agente durante a transição corrente, porém a próxima transição somente será retomada após o recebimento das informações por ventura solicitadas pelas mensagens enviadas durante a última iteração da regra de transição.

Para controle do sincronismo, todo agente possui internamente um contador, denominado *PendingAnswers* e as filas *AnswersReceived* e *RequestsReceived*. O contador *PendingAnswers* contabiliza o número de chamadas de abstrações de outros agentes que foram disparadas pelo agente na última execução de sua regra de transição. A fila *AnswersReceived* tem o formato de uma lista de atualização. O conteúdo desta fila é relativo ao valor de retorno de parâmetros de chamadas a abstrações enviadas a

outros agentes. Por fim, a fila *RequestsReceived* armazena os pedidos de execução de abstrações de regras encaminhadas ao agente.

Agentes nos estados **anew**, **active**, **stopped** e **blocked** estão habilitados a atender requisições depositadas em sua fila *RequestsReceived*. Agentes no estado **active** somente o fazem entre transições, ou seja, antes de reiterar a execução de sua regra de transição. Em qualquer caso, o agente retira uma a uma as atualizações contidas na sua fila *AnswersReceived* e realiza, na ordem da fila, as atualizações indicadas. Em seguida retira e executa, uma a uma, em ordem, todas as abstrações de regras relacionadas na fila *RequestsReceived*.

Execução Assíncrona: o operador **dispatch** (vide 7.10) deve ser usado para disparar a execução assíncrona de uma abstração. Este operador sempre retorna imediatamente com um valor do tipo **Promise**, que contém dados sobre a identificação da respectiva chamada da abstração e o estado de sua execução. Entre uma transição e outra do agente, o valor de uma **Promise** pode ser atualizado automaticamente pelo sistema de execução para indicar que a respectiva chamada da abstração foi concluída. Esta condição pode ser verificada a qualquer momento pelo predicado `completed(p)`, onde `p` é uma **Promise**.

Sincronização de Agentes

Funções declaradas **shared** podem ser acessadas simultaneamente por mais de um agente, possivelmente criando situações de disputa (*race condition*) por recursos compartilhados, normalmente com resultados imprevisíveis.

A sincronização de acesso a recursos compartilhados, seja para escrita ou leitura, pode ser obtida por meio de troca de mensagens entre agentes.

3.3 Problema das Linguagens de Especificação Formal

Não encontramos nenhum trabalho relacionado a linguagens de especificação formal orientadas por aspecto. A linguagem Machĭna, na versão atual, não possui recursos apropriados para especificar interesses transversais (do inglês, *crosscutting concern*). Interesses transversais são funcionalidades de um sistema que não são adequadamente encapsuladas em módulos, seja porque se aplicam aos diversos módulos de um sistema simultaneamente, seja porque não pertencem à natureza intrínseca dos módulos os quais se aplicam. A dificuldade de modularizar alguns tipos de funcionalidades causa um acúmulo de responsabilidade adicional, ocasionando um entrelaçamento da

responsabilidade inicial com responsabilidade extra. Na metodologia ASM, como o Modelo Básico fornece um alto nível de abstração, pode-se esconder estes problemas de especificação. Mas após alguns refinamentos, torna-se claro o entrelaçamento das responsabilidades envolvidas.

Para ilustrar melhor o problema relacionado aos interesses transversais, serão mostrados dois exemplos utilizando a linguagem Machĭna. O objetivo é mostrar, na prática, que o código dos interesses transversais fica entrelaçado com código dos interesses principais (do inglês, *core concern*). Cabe ressaltar que estes exemplos são mais bem detalhados no Capítulo 5.

3.3.1 Linguagem *Tiny*

Tiny é uma linguagem de programação imperativa de pequeno porte contendo somente comandos e expressões. Todo comando de *Tiny*, quando executado, modifica um estado. Este estado contém três elementos:

- a `memória` – correspondência entre identificadores e valores. Na memória, cada identificador está associado a algum valor ou está não-associado (*unbound*);
- a `entrada` – fornecida pelo usuário antes do início da execução do programa; consiste em uma seqüência (possivelmente vazia) de valores que podem ser lidos utilizando uma expressão;
- a `saída` – é inicialmente uma seqüência vazia de valores que armazena os resultados de um comando de saída, a saber, *output E*.

Especificação da Semântica de *Tiny*

Basicamente, o vocabulário de *Tiny* contém os seguintes nomes de função:

- *program* : função de aridade 0 que contém um programa em Tiny;;
- *memory* : função de aridade 1 que associa nomes de identificadores a valores;
- $infile$: função de aridade 0 que contém a seqüência de valores de entrada;
- *output* : função de aridade 0 que contém a seqüência de valores de saída do programa;
- *opstack* : função de aridade 0 que simula uma pilha auxiliar de execução.

Em [BTI+07], *Tiny* é definida em cinco módulos, a saber:

1. Módulo principal (`MainProgram`) – contém a regra principal da especificação e as rotinas de inicialização;

2. Módulo de Expressões (`Expressions`) – contém as regras de avaliação de expressões;

3. Módulo de Comandos (`Commands`) – contém as regras de execução de comandos;

4. Módulo de Operações (`Operations`) – contém as regras de operação na pilha.

5. Módulo de Globais (`Globals`) – contém as declarações dos principais tipos e funções da especificação de *Tiny*.

A verificação de tipos é um interesse transversal. Assim, como a versão atual da linguagem Machĭna não possui meios apropriados para realizar a separação, esses interesses tendem a ficar espalhados e entrelaçados com outros interesses. Uma das conseqüências naturais é uma menor compreensibilidade do código.

As Listagens 3.8, 3.9 e 3.10 apresentam exemplos de entrelaçamento e espalhamento de interesses. A Listagem 3.8 apresenta uma parte da codificação do módulo `Operations`. Observe como a verificação de tipos encontra-se espalhada por todo módulo, linhas 6, 19 e 27. Já a Listagem 3.9 ilustra parte do módulo `Expression`. O tratamento de erro encontra-se espalhado nas linhas 7, 8, 13 e 14. Por fim, a Listagem 3.10 apresenta a estrutura do módulo `MainProgram`, onde fica claro o entrelaçamento das regras do módulo com as regras referentes ao tratamento de erro.

3.3.2 Caldeira a Vapor

O problema da Caldeira a Vapor, proposto em [Abr94], consiste em desenvolver um programa para controlar o nível de água de uma caldeira a vapor. É importante que o programa trabalhe corretamente porque a quantidade de água presente, quando a caldeira estiver funcionando, deve sempre estar entre os valores limites permitidos; caso contrário, a caldeira ficará danificada.

Para destacar o entrelaçamento dos interesses transversais, nesta seção, optou-se por simplificar a especificação proposta em [Abr94]. O sistema comunica-se com as unidades físicas por meio de mensagens, sendo que o tempo para transmissão pode ser negligenciado. O programa segue um ciclo que, em princípio, não termina. Este ciclo acontece a cada cinco segundos e consiste nas seguintes ações: (a) receber as mensagens das unidades físicas, (b) analisar a informação recebida e (c) transmitir uma mensagem aos dispositivos.

O sistema possui as seguintes unidades:

```
1  module Operations
2    ....
3    abstractions:
4
5      public action treatValue is
6          if head(opstack) is Globals.KeyWord then
7            case Globals.KeyWord (head(opstack))
8               of Globals.TOUTPUT => treatOutput;
9               ...
10              of Globals.TEQUALS => treatEquals;
11           end
12         else error := true
13         end
14     end treatValue
15
16     public action treatAdd is
17         let x = head(program);
18         let y = head(tail(program));
19         if (x is Int) and (y is Int) then
20           program := (Int(x) + Int(y))::tail(tail(program));
21           opstack := tail(opstack)
22         else error := true
23         end
24     end treatAdd
25
26     public action treatNot is
27         if head(program) is Bool then
28           program := (not (Bool(b)))::tail(program);
29           opstack := tail(opstack)
30         else error := true
31         end
32     end treatNot
33
34 end Operations
```

Listagem 3.8: Parte da codificação do módulo `Operations`

- uma válvula;
- duas unidades medidoras;
- quatro bombas de água;
- quatro dispositivos para supervisionar as bombas.

Considerou-se que nunca acontece erro com as unidades físicas, com a exceção da unidade medidora de nível de água, e que não ocorrem perdas de mensagens. Assim, o programa, nesta especificação simplificada, pode operar nos modos inicialização, normal ou recuperação. O modo de inicialização dá início ao funcionamento da caldeira. O modo normal tenta manter um nível satisfatório de água. Porém, caso a unidade medidora de nível de água fique danificada, o sistema entra no modo recuperação. O

```
module Expressions
  ...
  abstractions:

    public action readMemory is
        let id = String (head(program));
        if memory(id) is Globals.Undefined then
          error := true
        else program := memory(id) :: tail(program)
        end
    end
    public action treatRead is
        if infile = nil then
          error := true
        else
          program := head(infile)::tail(program);
          infile := tail(infile)
        end
    end
    ...
end Expressions
```

Listagem 3.9: Parte da codificação do módulo `Expression`

```
module MainProgram
  ...
  transition:
    if program != nil and not error then
       //regras do módulo
    else
       //regras para tratar o erro
    end;
end MainProgram
```

Listagem 3.10: Parte da codificação do módulo `MainProgram`

sistema permanece no modo recuperação até ajustar a unidade medidora de nível de água, voltando, assim, ao modo normal. A Figura 3.1 ilustra o fluxo possível entre os modos.

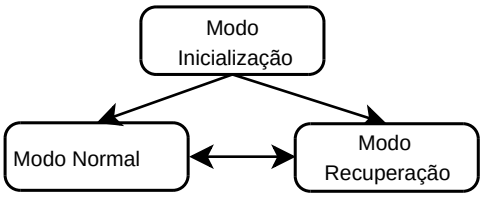

Figura 3.1: Fluxo de execução entre os modos

3.3.2.1 Especificação Formal

Diferente da solução apresentada em [BBD+96], o sistema foi decomposto em módulos, sendo que cada módulo implementa um interesse bem definido. Com isso, o sistema ficou divido nos seguintes módulos:

- **Inicialização, Normal e Recuperação** contêm as funções relativas a cada modo da caldeira, por exemplo, o modo de inicialização possui funções que inicializam o sistema e verificam o nível da água.
- **Sincronização** responsável pela sincronização das ações dos ciclos. Pela especificação, um ciclo está dividido em três ações: leitura, execução e escrita. O módulo Sincronização deve garantir que todo o processamento ocorra apenas na execução.

Os módulos Inicialização, Normal e Recuperação implementam os interesses principais do sistema. Já o módulo Sincronização implementa um interesse transversal. A Figura 3.2 mostra o entrelaçamento dos interesses principais e transversais. Por exemplo, todas as regras dos módulos Inicialização, Normal e Recuperação devem ter como precondição um teste que identifique que a ação atual do ciclo é execução.

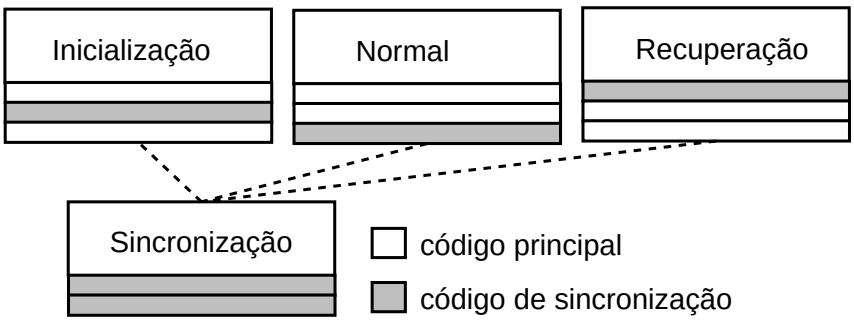

Figura 3.2: Entrelaçamento dos interesses no problema na caldeira a vapor

As Listagens 3.11, 3.12 e 3.13 ilustram uma possível codificação dos módulos Sincronização, Inicialização e Normal. O código do módulo Sincronização, interesse transversal, fica entrelaçado com o código dos demais módulos, interesses principais. Nas Listagens 3.12 e 3.13 o entrelaçamento está representado nas linhas 2 e 6. Cabe ressaltar que o objetivo do exemplo é apenas demonstrar que as regras dos interesses transversais

ficam espalhadas e entrelaçadas. Desta forma, caso seja especificado mais alguns interesses transversais, como demonstrado no Capítulo 5, o espalhamento e entrelaçamento desses dificultará o entendimento e compreensão de toda especificação.

```
1  module Sincronizacao
2    ...
3    ...
4    transition:
5      if phase = READING and inNextCycle then
6        phase := EXECUTING;
7      elseif phase = EXECUTING then
8        phase := WRITING
9      elseif acao = WRITING
10       phase := READING;
11     end
12 end Sincronizacao
```

Listagem 3.11: Codificação do módulo de sincronização em Machĭna

```
1  module Inicializacao
2    include Sincronizacao;
3    ...
4    ...
5    transition:
6      if Sincronizacao.phase = EXECUTING and modo = inicializacao then
7        if (precondições para entrar no modo normal) then
8          modo := normal
9        else
10         modo := recuperacao
11     end
12   endend Inicializacao
```

Listagem 3.12: Codificação do módulo de inicialização em Machĭna

```
1  module Normal
2    include Sincronizacao;
3    ...
4    ...
5    transition:
6      if Sincronizacao.phase = EXECUTING and modo = normal then
7        if (precondições para entrar no modo recuperação) then
8          modo := recuperacao
9        end
10     end
11   endend Normal
```

Listagem 3.13: Codificação do módulo normal em Machĭna

3.4 Considerações Finais

Máquinas de estado abstratas fornecem uma metodologia precisa para especificação de algoritmos. Esta metodologia está baseada em sólidos conceitos matemáticos, mas ao mesmo tempo apresenta semelhanças com linguagens de programação, facilitando o aprendizado por não-especialistas. A linguagem Machĭna, com seu rico conjunto de instruções, e as pesquisas relacionadas a programação orientada por aspecto formam o contexto para a elaboração do trabalho proposto.

Como a linguagem Machĭna é baseada no modelo ASM, um tipo de semântica operacional, uma especificação em Machĭna pode ser executada. Este trabalho apresenta como a compilação de uma especificação em Machĭna e suas novas construções podem ser mapeadas em C++ e AspectC++.

Conforme visto neste capítulo, existem problemas na especificação de interesses transversais, como pode ser observado nas especificações da linguagem *Tiny* e da cadeira a valor. Cabe ressaltar que o entrelaçamento causado pelo código dos interesses transversais, como destacado no Capítulo 1, ocasiona os seguintes problemas:

- o entendimento do código fica mais difícil, uma vez que temos códigos com propósitos distintos misturados;
- como o código referente a um interesse transversal não é apropriadamente encapsulado, ele encontra-se espalhado por todo o sistema, desta forma, uma alteração neste código provocará modificações em vários lugares, tornando com isso difícil a manutenção/evolução do sistema, que é a parte responsável pela maior parcela de custos no processo de desenvolvimento de software;
- visto que o código do interesse principal está misturado com o código do interesse transversal, a reusabilidade do componente fica prejudicada caso desejarmos fazer uso da sua funcionalidade básica em um contexto diferente;

Por isso, novas construções fazem-se necessárias. O trabalho proposto oferece novos recursos para a linguagem Machĭna com o objetivo de sanar estes problemas apresentados.

Capítulo 4

A Linguagem Aspect$\mathcal{M}$

Este capítulo apresenta a linguagem Aspect$\mathcal{M}$, uma linguagem de especificação formal baseada no modelo ASM e orientada por aspectos. Aspect$\mathcal{M}$ representa uma nova versão da linguagem Machĭna. Desta forma, além de possuir todas as características da linguagem Machĭna, como, construções de alto nível e criação de novos tipos, possui suporte à modularização dos interesses transversais. Assim, esta nova versão acrescenta à linguagem Machĭna alguns benefícios, dentre eles, a diminuição das responsabilidades dos módulos e a melhor modularização dos interesses.

4.1 Visão Geral

Aspect$\mathcal{M}$ é uma linguagem orientada por aspectos e constitui-se de uma nova versão da linguagem de especificação formal Machĭna [BTI+07]. Esta versão acrescenta novas construções à linguagem, permitindo a implementação de interesses transversais de forma modular. Por meio de Aspect$\mathcal{M}$, o usuário pode definir pontos específicos no fluxo de execução das regras de transição, como a chamada de ações ou o acesso a funções, e a partir destes pontos, pode-se definir regras de transição. Além disso, pode-se também alterar a álgebra dos módulos de forma não intrusiva e transparente para os módulos afetados.

No projeto de Aspect$\mathcal{M}$, mantiveram-se as seguintes compatibilidades:

- compatibilidade superior - todas as especificações válidas na versão antiga devem ser especificações válidas em Aspect$\mathcal{M}$.
- compatibilidade de plataformas - todas as especificações válidas em Aspect$\mathcal{M}$ devem ser executadas de forma análoga a especificações na versão antiga.
- compatibilidade de ferramentas - as ferramentas existentes, como [Ste07], podem ser usadas.

- compatibilidade de programador - os programadores de Machĭna devem se sentir confortáveis programando em Aspect$\mathcal{M}$.

Baseado em Eos [RS03], Aspect$\mathcal{M}$ não possui uma nova unidade de modularização para implementação dos interesses transversais. Utilizando a idéia de *Classpects* proposta por Rajan e Sullivan, em [RS05b], as novas construções desta versão são definidas no próprio módulo, em uma seção denominada **aspect**. Assim, os módulos são compilados por um compilador especial, que une os elementos e gera um código em C++ e AspectC++.

O compilador desenvolvido pode compilar especificações escritas puramente em Machĭna. Quando existir alguma definição de aspecto, esses são compilados junto ao código Machĭna, e uma etapa adicional é realizada durante a compilação, chamada costura de código. Nessa etapa, realizada pelo que se chama um combinador (weaver), as especificações de comportamentos transversais, definidas na seção `aspect`, são costuradas, ou combinadas, ao código gerado.

Simplificando o processo de compilação, pode-se dizer que o compilador *front-end* lê o código-fonte dos módulos e gera uma AST e um conjunto de informações para a costura de código. O *back-end* então usa a AST do programa e as informações de costura para gerar um código em C++ e AspectC++. No Capítulo 5, o processo de compilação é mais bem detalhado.

Análogo a AspectJ, esta versão acrescenta à linguagem Machĭna dois tipos de implementação de interesses transversais: transversalidade dinâmica e transversalidade estática.

A transversalidade dinâmica é restrita a um conjunto pré-definido de pontos especificados por um modelo de pontos de junção. Esse modelo concentra-se na execução das regras de transição de uma especificação em Machĭna.

O modelo de ponto de junção é o principal componente de uma linguagem orientada por aspectos e representa um arcabouço conceitual para descrever os tipos de pontos de junção de interesse e as restrições associadas a seu uso. Ele é altamente dependente da linguagem de componentes adotada, neste caso, a linguagem Machĭna.

A transversalidade dinâmica, em Aspect$\mathcal{M}$, permite definir regras adicionais em pontos específicos no fluxo de execução das regras de transição. Por exemplo, pode-se especificar que uma certa regra seja executada paralelamente à execução de outras regras ou ações de um conjunto de módulos. Para isso, basta especificar separadamente qual regra deverá ser utilizada e os pontos de junção onde a mesma será inserida.

A transversalidade estática afeta as estruturas dos módulos de uma especificação em Machĭna. Tem-se, por exemplo, a introdução de algumas funções ou tipos em um conjunto de módulos previamente especificados.

Desta forma, Aspect$\mathcal{M}$ permite a modularização adequada de uma grande variedade de interesses transversais, como verificação e tratamento de erros, sincronização, distribuição, monitoramento e auditoria.

Palavras Reservadas

Os seguintes identificadores são declarados como palavras reservadas de Aspect$\mathcal{M}$, além dos já existente em Machĭna:

addition	**pointcut**	**around**
introduction	**proceed**	**execution**
get	**after**	**initialization**
function	**transition**	**args**
within	**withincode**	**before**
aspect	**target**	

Notação para Sintaxe

Para descrição da gramática, utiliza-se a BNF estendida [Wir77], ou XBNF. As construções utilizadas são semelhantes à BNF tradicional, com as seguintes extensões:

|	Opções
{ X }	Zero ou mais ocorrências de X
[X]	Zero ou uma ocorrência de X

Os caracteres que representam meta-símbolos, se utilizados na gramática, são escritos em aspas, por exemplo, "{".

4.2 Novas Construções de Machĭna

Basicamente, as novas construções de Machĭna consistem em: uma nova seção, **aspect**, para a estrutura dos módulos existentes em Machĭna, e um mecanismo de composição, a transversalidade, constituída de conjuntos de junção que identificam conjuntos de pontos de junção; adendos que definem regras de um dado conjunto de junção e construções para afetar estaticamente a estrutura dos módulos.

4.2.1 Seção Aspect

No desenvolvimento de AspectJ, alguns princípios de projeto, por exemplo, generalidade e ortogonalidade, foram prejudicados em favor da adaptabilidade, que, entre outras

coisas, foi visto como o principal ponto para uma avaliação empírica dos principais conceitos da programação orientada por aspectos [RS05b].

Segundo Rajan e Sullivan [RS05b], Kiczales relatou que a decisão de separar as classes dos aspectos foi baseada nas necessidades dos usuários, que gostariam de ver e controlar os novos mecanismos introduzidos.

Assim, as linguagens baseadas em AspectJ suportam duas construções relacionadas, mas distintas, para modularização dos interesses: classes e aspectos. As classes possuem construções para encapsular os interesses principais, enquanto os aspectos fornecem as construções necessárias para modularizar os interesses transversais. Desta forma, o projeto dos sistemas implementados com essas linguagens são estruturados em duas camadas: uma camada base, tipicamente código orientado por objetos, é entrecortada por uma camada de aspectos.

Em [RS05b], os autores descrevem que a separação de classes e aspectos reduz a integridade conceitual do modelo de programação. Além disso, a assimetria entre classes e aspectos dificulta a composição do sistema. Esta assimetria ocorre em duas situações. Primeiro, enquanto aspectos podem entrecortar classes, classes não podem entrecortar aspectos, e aspectos não podem entrecortar aspectos com a mesma flexibilidade com que entrecortam classes. Segundo, as instâncias de aspectos não podem ser manipuladas no fluxo de execução do programa do mesmo modo que os objetos. As novas versões de AspectJ provêem novos conjuntos de junção para tentar amenizar estes problemas.

Desta forma, baseado na idéia de unificar classes e aspectos, como as *Classpects* de Rajan e Sullivan [RS05b], as novas construções de Aspect$\mathcal{M}$ são definidas no próprio módulo, em uma seção denominada **aspect**. A Listagem 4.1 ilustra a estrutura de um módulo em Aspect$\mathcal{M}$.

Esse módulo possui todas as construções de um módulo na versão antiga, além de um conjunto de construções usuais de uma linguagem orientada por aspectos, como AspectJ. Assim, um módulo pode entrecortar outro módulo naturalmente. Rajan e Sullivan [RS05a] apresentam uma validação formal para provar que o modelo de unificação de classes e aspectos, implementado na linguagem Eos, é mais expressivo que o modelo de ponto de junção e adendo (*Poincut and Advice Model*), no qual a linguagem AspectJ é implementada.

A sintaxe de um módulo de programa em Aspect$\mathcal{M}$ é a seguinte:

```
1  module module-name
2       import elementos importados
3       include elementos incluídos
4       algebra:
5            elementos declarados(funções e tipos)
6       abstractions:
7            declaração de abstrações(ações)
8       initial state:
9            inicializações de funções dinâmicas
10      transition:
11           regras de transição de estado
12      aspect:
13           declaração de intertipos, conjuntos de junção e adendos
14      invariant:
15           invariante de execução
16 end module-name
```

Listagem 4.1: Estrutura de um módulo em Aspect$\mathcal{M}$

program-module	::=	**module** module-name module-body **end** [module-name]
module-body	::=	[declaration-part] [initial-state-part] [transition-part] [aspect-part] [invariant-part]
declaration-part	::=	{ foreign-part } [algebra-part] [abstraction-part]
foreign-part	::=	import-part | include-part
import-part	::=	**import** importation-list [;]
include-part	::=	**include** inclusion-list [;]
algebra-part	::=	**algebra:** { algebra-section }
abstraction-part	::=	**abstractions:** { abstraction-section }
initial-state-part	::=	**initial state:** single-transition
transition-part	::=	**transition:** main-transition
aspect-part	::=	**aspect:** { aspect-section }
aspect-section	::=	pointcut-part | advice-part | inter-type-part
invariant-part	::=	**invariant:** expression

O significado das novas estruturas pointcut-part, advice-part e inter-type-part são definidos a seguir e o significado das demais estruturas encontram-se em [BTI+07].

4.2.2 Transversalidade Dinâmica

Aspect$\mathcal{M}$ é uma nova versão da linguagem Machĭna que permite a inclusão de comportamento dinâmico em pontos definidos no fluxo de execução das regras de transição. O modelo de pontos de junção (MPJ) da linguagem é léxico e identifica entidades e configurações da especificação em Machĭna por seus nomes. Entidades, como módulos e ações, são artefatos estáticos de uma especificação. Configurações são artefatos

dinâmicos da simulação desta especificação, como a chamada de ações.

Como destacado no Capítulo 2, a capacidade de uma linguagem de aspectos suportar a modularização de interesses transversais é determinada pelo MPJ. Um MPJ consiste, basicamente, em três elementos: (i) pontos de junção, (ii) conjunto de junção, meios que identifiquem um ponto de junção e (iii) adendos, meios que especifiquem semântica em um ponto de junção.

Exemplo Base

A Listagem 4.2 apresenta um trecho de código, parte de um módulo para o tratamento de operações escrito em Machĭna, extraído de uma especificação de uma linguagem de programação imperativa, a saber, *Tiny* apresentada no Capítulo 3. Essa listagem será utilizada ao longo desta seção para ilustrar a utilização dos conjuntos de junção; o programa é representado por uma lista denominada program e as operações são empilhadas em opstack. As linhas de 5 a 11 definem qual operação será realizada, com base na pilha opstack. As linhas de 13 a 16 definem um tratamento para o operador de soma, enquanto que as linhas de 18 a 21 definem um tratamento para a operação de negação. Observe que o interesse relacionado à verificação de tipos foi removido da especificação, tornando-a bem mais clara e concisa.

Ponto de Junção

Aspect$\mathcal{M}$ adota um modelo no qual um ponto de junção é um ponto bem definido no fluxo de execução das regras de transição de uma especificação em Machĭna. Os pontos de junção de Aspect$\mathcal{M}$ incluem a chamada ou execução de ações, inicialização de funções, acesso a funções, escopo de execução, entre outros.

Como exemplo de ponto de junção em uma especificação Machĭna, na Listagem 4.2, tem-se o acesso e atribuição de valores a funções, nas linhas 6, 14, 15, dentre outras. Como exemplo de chamada a abstrações, tem-se a chamada a ação treatOutput, linha 7. Por fim, como exemplo de escopo de execução, tem-se o acesso a função program dentro da ação treatAdd.

Conjunto de Junção

Um conjunto de junção captura ou identifica pontos de junção no fluxo de execução das regras de transição. Uma vez capturado um ponto de junção, pode-se especificar adendos (regras adicionais) envolvendo este ponto, por exemplo, executar uma outra regra ao mesmo tempo. Além disso, alguns conjuntos de junção expõem dados do contexto de execução do ponto de junção capturado, como, seu tipo ou assinatura. Assim, os adendos podem utilizar este contexto para implementar novas funcionalidades.

```
1  module Operations
2    include Globals(program, opstack)
3    abstractions:
4
5      public action treatValue is
6        case Globals.KeyWord (head(opstack))
7          of Globals.TOUTPUT => treatOutput;
8          ...
9          of Globals.TEQUALS => treatEquals;
10       end
11     end treatValue
12
13     public action treatAdd is
14       let x = head(program);
15       let y = head(head(program));
16       program := (Int(x) + Int(y))::tail(tail(program));
17       opstack := tail(opstack)
18     end treatAdd
19
20     public action treatNot is
21       let b = head(program);
22       program := (not (Bool(b)))::tail(program);
23       opstack := tail(opstack)
24     end treatNot
25
26 end Operations
```

Listagem 4.2: Parte da codificação do módulo `Operations` sem verificação de tipos

Em Aspect$\mathcal{M}$, os conjuntos de junção podem ser anônimos ou nomeados. Conjuntos de junção anônimos são definidos no lugar onde são utilizados, ou seja, como uma parte de um adendo ou na definição de outro conjunto de junção. Todos os conjuntos de junção devem ser declarados dentro da seção **aspect** de um módulo em Aspect$\mathcal{M}$. Os conjuntos de junção nomeados podem ser referenciados em qualquer lugar, dentro dessa seção, **aspect**, aumentando o grau de reusabilidade.

De forma análoga a uma função, um conjunto de junção pode utilizar um modificador de acesso para restringir o seu uso. Além disso, os conjuntos de junção podem ser compostos pelos operadores **and** (interseção), **or** (união) e **not** (diferença), a fim de criar outros conjuntos de junção. A ordem de precedência desses operadores é mostrada na tabela a seguir, sendo que os de precedência mais alta estão na primeira linha e os de precedência mais baixa estão no fim.

not	precedência mais alta
and	
or	precedência mais baixa

A Listagem 4.3 ilustra algumas definições de conjuntos de junção. As linhas 2 e 3 definem o conjunto de junção executionTreatAdd que captura a execução da ação

treatAdd do módulo Operations. Nas linhas de 5 a 7 é definido o conjunto de junção addCheck. Observe que esse conjunto de junção é definido por um conjunto de junção nomeado, linha 6, e um conjunto de junção anônimo, linha 7. Este conjunto de junção captura o agente que executará a ação treatAdd e o associa ao identificador agente.

```
1    aspect:
2        pointcut executionTreatAdd :
3            execution(action Operations.treatAdd)
4
5        pointcut addCheck(agente:Operations) :
6            executionTreatAdd and
7            target(agente)
```

Listagem 4.3: Exemplo de conjunto de junção anônimo e nomeado em AspectM.

Similar a AspectJ, AspectM possibilita a utilização de alguns símbolos curinga (*wildcard*) na construção de um conjunto de junção, provendo assim, um modo prático para capturar os pontos de junção que compartilhem algumas características em comum.

O símbolo "..." representa qualquer número de caracteres incluindo o ".". Esse símbolo pode ser utilizado para designar funções e/ou ações com quaisquer números e tipos de argumentos. A construção **dynamic** Globals.memory(...), por exemplo, representa todas as funções dinâmicas de nome memory que pertençam ao módulo Globals.

O símbolo "*" representa qualquer número de caracteres com exceção do ".". Desta forma, pode-se utilizá-lo para representar:

- Qualquer tipo de módulo. Por exemplo, a construção **dynamic** *.treatNot que especifica uma função dinâmica de nome treatNot pertencente a qualquer módulo.
- Qualquer função. Por exemplo, a construção * Expression.*(...) que especifica todas as funções do módulo Expression.

Além disso, esse símbolo pode ser utilizado em nomes parciais, como na construção **dynamic** Expression.treat*, que especifica todas as funções do módulo Expression iniciadas com a palavra treat.

Em Machĭna, os módulos, interfaces, ações e funções contêm assinaturas. Em AspectM, pode-se utilizar padrões de assinatura para especificar várias assinaturas contidas em uma especificação Machĭna. Os símbolos curingas e os operadores de conjunto podem ser utilizados para especificar um padrão de assinatura. A Tabela 5.1 apresenta alguns exemplos.

A seguir, é apresentada a sintaxe dos principais padrões utilizados no decorrer da seção:

Padrão de Assinatura	Significado
Globals	módulo de nome Globals
Exp*	módulos com o nome iniciado com o prefixo Exp
not dynamic *.*	todas as funções que não sejam dinâmicas
derived Globals.*	todas as funções derivadas do módulo Globals
action Expression.* **or** **static** Expression.*	todas as ações ou funções estáticas do módulo Expression

Tabela 4.1: Exemplo de alguns padrões de assinatura

module-pattern	::=	name-pattern
action-pattern	::=	**action** name-pattern.name-pattern [(parameters)]
function-pattern	::=	class-modifier name-pattern.name-pattern [(parameters)]
class-modifier	::=	**static**| **derived**| **dynamic**| **external**| *
parameters	::=	type-expression { , type-expression }
name-pattern	::=	[*] name [*] | *

Designadores de Conjunto de Junção

Aspect$\mathcal{M}$ provê uma coleção de designadores para categorizar um conjunto de pontos de junção. Um designador de conjunto de junção é uma fórmula que especifica um conjunto de pontos de junção ao qual um adendo pode ser aplicado. A Listagem 4.4 ilustra a utilização dos designadores *execution* e *get*, além da composição de conjuntos de junção. O código desta listagem define o conjunto de junção unaryOpCheck que captura todos os acessos à função program do módulo Globals, na execução da ação treatNot do módulo Operations.

```
1  aspect:
2      pointcut unaryOpCheck :
3          execution(action Operations.treatNot) and
4          get(dynamic Globals.program)
```

Listagem 4.4: Codificação de um conjunto de junção em Aspect$\mathcal{M}$

Convém ressaltar que, basicamente, todos os designadores utilizados para capturar pontos de junção atuam no estado corrente. Como exemplo, considere uma máquina abstrata que se encontra no estado S_n, qualquer conjunto de junção irá capturar regras que forem executadas em S_n. Assim, todos os pontos são capturados antes da execução da lista de atualizações, o que causaria a mudança do estado S_n para S_{n+1}. O único

designador que permite capturar pontos após a execução da lista de atualizações é o **transition**, que será explicado no decorrer deste capítulo.

Na Tabela 4.2, estão relacionados os principais designadores implementados em Aspect$\mathcal{M}$, com suas respectivas características.

Designador	Características
execution(action-pattern)	Execução de ações
get(function-pattern)	Referência a funções
initialization(module-pattern)	Inicialização de estado
transition(module-pattern)	Transição entre os estados
target(identificador ou module-pattern)	O agente receptor(alvo) é do tipo do módulo especificado ou associado ao identificador
args(identificador ou tipo)	Os argumentos são instância do tipo informado ou são associados ao identificador
within(module-pattern)	O código em execução está definido no(s) módulo(s) especificado(s)
withincode(action-pattern)	O código em execução está definido na(s) ação(es) especificada(s)

Tabela 4.2: Designadores em Aspect$\mathcal{M}$

A sintaxe para declaração dos conjuntos de junção em Aspect$\mathcal{M}$ é a seguinte:

pointcut-part	::=	[**public**] **pointcut** pointcut-declaration
pointcut-declaration	::=	pointcut-name [(pointcut-parameters)] : pointcut-designators
pointcut-parameters	::=	pointcut-parameter { , pointcut-parameter }
pointcut-parameter	::=	parameter-name : type-expression
pointcut-designators	::=	pointcut-designator | **not** pointcut-designators | pointcut-designators set-op pointcut-designator | (pointcut-designators)
set-op	::=	**and** | **or**
pointcut-designator	::=	inicialization-designator | execution-designator | get-designator | transition-designator | arg-designator | target-designator | within-designator | withincode-designator | name-designator[(name-parameters)]
name-parameters	::=	name-parameter { , name-parameter }

O restante desta seção explica cada designador, agrupando-os com base nos tipos de pontos de junção capturados e apresenta-se um pequeno exemplo para cada designador

seguido de sua sintaxe. Por fim, outros exemplos, que envolvem um maior número de designadores, são apresentados.

Execução de Ações

O designador de execução de ações captura a execução das ações. Para definir este tipo de designador, utiliza-se o operador **execution**, parametrizado com a assinatura da ação. Por exemplo, para representar todas as execuções das ações treatAdd e treatNot do módulo Operations, a Listagem 4.5 define o conjunto de junção operationsActionExecute.

```
1  aspect:
2    public pointcut operationsActionExecute :
3      execution(action Operations.treatAdd) or
4      execution(action Operations.treatNot)
```

Listagem 4.5: Exemplo do designador para execução de ações

A definição do designador de execução de ações é semelhante à definição do designador de chamada. No entanto, em linguagens como AspectJ, pode-se destacar como principais diferenças:

- o contexto no qual estão inseridos: por exemplo, a chamada de uma ação f no corpo da ação g leva como informação de contexto para o **call** os elementos de g, ao passo que, para **execution**, leva os elementos de f;
- o local onde as regras do aspecto são inseridas: as regras são espalhadas no ponto de chamada para o designador **call** e no ponto de definição para o designador **execution**.

Dentro do contexto da linguagem Machĭna, considerou-se que os designadores de execução e chamada de ações são iguais, porque como as regras executam em paralelo em um mesmo estado não faz diferença onde os códigos serão inseridos. Além disso, o superagente é quem executa as ações, a pedido dos agentes, provendo assim o mesmo contexto.

Sintaxe de uso do designador:

execution-designator ::= **execution** (action-pattern)

Acesso a Funções

O designador referente ao acesso a funções captura a leitura de qualquer tipo de função de um módulo. Para definir este tipo de designador, utiliza-se o operador

get, parametrizado com a assinatura da função. Por exemplo, a Listagem 4.6 define o conjunto de junção **globalFunctionAcess**, que representa os acessos de leitura à função program do módulo Globals.

```
1  aspect:
2    pointcut globalFunctionAcess :
3      get (dynamic Globals.program)
```

Listagem 4.6: Exemplo dos designadores para acessar funções

Cabe ressaltar que o designador **set**, comum a linguagens orientadas por aspectos baseadas em AspectJ, não foi integrado ao modelo de ponto de junção da linguagem proposta. O designador **set** captura as atualizações das funções, e dentro do contexto do modelo ASM, onde todas as regras executam em paralelo, a semântica desse designador é igual ao designador **transition**, que captura as mudanças de estado. Tais mudanças são causadas pela atualização de qualquer função. Cabe ressaltar que o designador **transition** será explicado no decorrer deste capítulo.

Sintaxe de uso do designador:

get-designator ::= **get** (function-pattern)

Inicialização de funções

O designador referente a inicialização de funções captura a execução do código especificado no bloco initial state dentro das definições dos módulos. A identificação é realizada por meio do operador **initialization**, parametrizado pelo nome do módulo. Como exemplo, tem-se a definição do conjunto de junção operationsInitialize, Listagem 4.7, que representa todos os pontos de junção que ocorrem no bloco initial state do módulo Globals ou dos módulos cujo nome inicia-se com a palavra Ope.

```
1  aspect:
2    public pointcut operationsInitialize :
3      initialization(Globals) or
4      initialization(Ope*)
```

Listagem 4.7: Exemplo do designador para capturar a inicialização de funções

Sintaxe de uso do designador:

inicialization-designator ::= **inicialization** (module-pattern)

Transição entre estados

O designador referente a transição entre os estados de um módulo descreve o ponto de execução antes, durante ou depois da atualização de todas as funções de um estado, ou seja, captura a transição de um estado S_n para um estado S_{n+1}. A definição de antes ou depois fica especificada no adendo, como será demonstrado na Seção 4.2.2.1.

A identificação é realizada por meio do operador **transition**, parametrizado pelo nome do módulo. Como exemplo, o conjunto de junção intercalateEverything, definido na Listagem 4.8, captura as transições de estado que ocorrem durante a execução.

```
1  aspect:
2    public pointcut intercalateEverything :
3      transition(*)
```

Listagem 4.8: Exemplo do designador para capturar as transições dos estados

Sintaxe de uso do designador:

transition-designator ::= **transition** (module-pattern)

Designadores Baseados no Contexto

Contextos podem ser utilizados na definição de conjuntos de junção. Em Aspect$\mathcal{M}$, pode-se utilizar designadores para definir conjuntos de junção nos seguintes elementos de contexto: tipo do agente receptor de uma chamada a uma função e argumentos de funções ou ações.

O designador **target**, parametrizado com a assinatura de módulo, é utilizado para definir pontos de junção que ocorrem durante a execução de uma ação cujo agente receptor é do tipo do módulo passado como parâmetro. Além disso, esse designador pode ser utilizado para associar um identificador a um agente do tipo capturado. A Listagem 4.9 tem por objetivo apenas ilustrar a sintaxe e uso. Nessa listagem, define-se o conjunto de junção useTarget, que captura todas as execuções de ações que ocorrem quando o agente receptor for do tipo Operations.

```
1  aspect:
2    public pointcut useTarget :
3      target("Operations") and
4      execution(action *.*)
```

Listagem 4.9: Exemplo do designador que utiliza o agente alvo

O designador **args** é utilizado para definir pontos de junção que utilizam os valores dos argumentos de uma função ou chamada de ação. Além disso, esse designador pode

ser utilizado para associar o parâmetro formal ao parâmetro real. Como exemplo, tem-se a definição do conjunto de junção argBinding, Listagem 4.10, que captura todas os acessos a função Globals.memory e associa o valor do parâmetro inteiro ao identificador id.

```
1  aspect:
2    public pointcut argBinding(id:String):
3      execution(dynamic Globals.memory(String)) and
4      args(id)
```

Listagem 4.10: Exemplo do designador que utiliza os valores dos argumentos

Sintaxe de uso dos designadores:

target-designator	::=	**target** (context-parameter)
arg-designator	::=	**args** (context-parameter { , context-parameter })
context-parameter	::=	" type-expression "| name

Designadores Baseados na Estrutura dos Módulos

A estrutura do módulo pode ser utilizada na definição dos conjuntos de junção. Em AspectM, pode-se utilizar designadores para especificar pontos de junção que ocorram dentro de módulos ou dentro de ações. O designador **within** é utilizado para definir pontos de junção que ocorrem no escopo do módulo cujo nome é dado como parâmetro. Por exemplo, o conjunto de junção moduleScope, definido na Listagem 4.11, linhas de 2 a 4, captura todos os acessos a função program do módulo Globals que ocorrem dentro do módulo Expression.

O designador **withincode** permite capturar pontos de junção no escopo de ações de módulos, passando no lugar de um nome de módulo a assinatura de uma ação. Por exemplo, o conjunto de junção actionScope, definido na Listagem 4.11, linhas de 6 a 8, captura todas as execuções da ação treatOutput que ocorrem dentro da acão treatValue do módulo Operations.

```
1  aspect:
2    public pointcut moduleScope :
3      get(dynamic Globals.program) and
4      within(Operations)
5
6    public pointcut actionScope :
7      execution(action *.treatOutput)  and
8      withincode(action Operations.treatValue)
```

Listagem 4.11: Exemplo dos designadores que utilizam a estrutura dos módulos

Sintaxe de uso dos designadores:

within-designator	::=	**within** (module-pattern)
withincode-designator	::=	**withincode** (action-pattern)

Outros Exemplos

Para melhor exemplificar a utilização dos designadores de conjunto de junção de Aspect$\mathcal{M}$, esta seção apresenta mais alguns exemplos.

A Listagem 4.12 ilustra a utilização de alguns designadores de conjunto de junção combinados. O conjunto de junção memoryAcess captura todos os acessos a função memory do módulo Globals que sejam realizadas em um módulo diferente do Operations. Além disso, associa o identificador x ao parâmetro real.

```
1  aspect:
2      public pointcut memoryAcess(x:String):
3          within(not Operations) and
4          get(public Globals.memory(String)) and
5          args(x)
```

Listagem 4.12: Combinando os designadores **within, get e args**

A Listagem 4.13 define o conjunto de junção simpleRef que captura todos os acessos às funções do módulo Globals que forem realizadas dentro do próprio módulo ou a partir de alguma ação iniciada por treat do módulo Operations

```
1  aspect:
2      public pointcut simpleRef:
3          (within(Globals) or
4           withincode(Operations.treat*)) and
5          get(* Globals.*)
```

Listagem 4.13: Combinando alguns padrões de assinatura e operadores booleanos

Por fim, a Listagem 4.14 define o conjunto de getReceiveAgent que captura todos os acessos à funções do módulo Global que sejam realizadas na ação treatNot do módulo Operations. Além disso, este conjunto de junção associa o agente receptor da chamada da ação ao identificador agente.

4.2.2.1 Adendo

O adendo compreende as ações que devem ser executadas em cada ponto de junção. Em Aspect$\mathcal{M}$ existem quatro tipos de adendo: (i) anterior, (ii) posterior, (iii) de adição e (iv) de contorno (do inglês *before, after, addition* e *around*).

```
1  aspect:
2      public pointcut getReceiveAgent(agente:Operations):
3          withincode(action Operations.treatNot) and
4          get(dynamic Globals.*) and
5          target(agente)
```

Listagem 4.14: Combinando os designadores **withincode, get e target**

O adendo anterior é executado antes do ponto de junção, enquanto que o adendo posterior é executado depois do ponto de junção. Diferente de linguagens como AspectJ, esses adendos, anterior e posterior, só podem estar associados a um designador, a saber **transition**, que permite capturar a transição de um estado S_n para um estado S_{n+1}. Por exemplo, pode-se especificar pré e pós condições para executar uma determinada regra de transição. Já o adendo de adição, criado para o contexto de ASM, executa paralelamente ao momento em que o ponto de junção é alcançado. Por fim, o adendo de contorno executa quando o ponto de junção é alcançado e tem controle explícito se o próprio ponto afetado deve ou não ser executado.

Essas declarações associam uma regra de transição a um conjunto de junção, indicando como uma regra entrecortará uma especificação em Aspect$\mathcal{M}$. Basicamente, quando o ponto de junção é capturado pelo conjunto de junção, a regra de transição é inserida para que posteriormente seja executada.

A definição de um adendo pode ser dividida em três partes: (i) a declaração do adendo; (ii) a especificação do conjunto de junção; (iii) e a declaração da regra de transição.

Para ilustrar essa definição será utilizado o conjunto de junção typeCheckAdd, definido na Listagem 4.15. Este conjunto de junção captura as execuções da ação treatAdd do módulo Operations e associa o identificador op ao agente que está executando a ação.

```
1  aspect:
2      pointcut typeCheckAdd(op:Operations):
3          execution(action Operations.treatAdd)
4          target(op)
```

Listagem 4.15: Conjunto de junção utilizado na definição do adendo da Listagem 4.16

Agora, a Listagem 4.16 apresenta a declaração de um adendo de contorno. Este adendo acrescenta uma verificação de tipos para ação treatAdd definida na Listagem 4.2, página 71.

A linha 2 representa a declaração do adendo e especifica quando o adendo vai executar a ação nos pontos capturados. Além disso, na declaração do adendo, podem-se

especificar as informações de contexto disponíveis para a ação, como o agente receptor de uma chamada de ação.

```
1  aspect:
2    around(op:Operations):
3      typeCheckAdd(op)
4       do
5         let x = head(op.program);
6         let y = head(head(op.program));
7         if x is Int and y is Int then
8           proceed
9         else op.error := true
10        end
11      end
```

Listagem 4.16: Definição de um adendo

Já a linha 3, parte entre os dois pontos e a palavra **do**, especifica os conjuntos de junção responsáveis pela coleta dos pontos de junção, ou seja, define os pontos onde a regra de transição, especificada pelo adendo, vai ser inserida para que posteriormente seja executada. Por fim, entre as linhas 4 e 11, é especifica a regra de transição que será inserida nos pontos coletados pelos conjuntos de junção especificados. A Listagem 4.17 apresenta um exemplo mais completo. Basicamente, essa listagem acrescenta uma verificação de tipos para a parte do módulo Operations definido na Listagem 4.2, página 71.

Por fim, apresenta-se a sintaxe para declaração de adendos em Aspect$\mathcal{M}$:

advice-part	::=	advice-declaration : pointcut-designators
		do single-transition **end**
advice-declaration	::=	advice-type [(advice-parameters)]
advice-type	::=	**around**| **addition**| **after**| **before**
advice-parameters	::=	advice-parameter { , advice-parameter }
advice-parameter	::=	parameter-name : type-expression

4.2.3 Transversalidade Estática

A transversalidade dinâmica, obtida a partir do modelo de ponto de junção, permite modificar o comportamento da execução do programa, ao passo que a transversalidade estática permite redefinir a estrutura estática dos tipos.

Em Aspect$\mathcal{M}$, é possível implementar a introdução de novos elementos à álgebra de um módulo, como tipos e funções. Além disso, é possível adicionar novas ações, ou seja, abstrações de regras de transição. Esse mecanismo de introdução é denominado

```
1  module TypeCheck
2   import Operations, Globals;
3
4   aspect:
5     around(op:Operations):
6       execution(action Operations.treatValue) and target(op) do
7         if head(op.opstack) is Globals.KeyWord then
8           proceed
9         else op.error := true
10        end
11    end
12
13    around(op:Operations):
14      execution(action Operations.treatAdd) and target(op) do
15        let x = head(op.program);
16        let y = head(head(op.program));
17        if x is Int and y is Int then
18          proceed
19        else op.error := true
20        end
21    end
22
23    around(op:Operations):
24      execution(action Operations.treatNot) and target(op) do
25        let x = head(op.program);
26        if x is Bool then
27          proceed
28        else op.error := true
29        end
30    end
31  end
```

Listagem 4.17: Codificação do aspecto para verificação de tipos

declaração de intertipos (do inglês, *inter-type declaration*). Desta forma, diferente dos adendos, que operam primariamente de forma dinâmica, as declarações de intertipos operam de forma estática, em tempo de compilação.

Os novos elementos introduzidos possuem as mesmas permissões de acesso referentes ao ponto de junção capturado, ou seja, uma ação introduzida em um módulo tem acesso a todos os membros daquele módulo, inclusive os privados.

A sintaxe para definição das declarações de intertipos em AspectM é a seguinte:

inter-type-part ::= type-section-intertype | function-section-intertype | external-section-intertype | abstraction-section-intertype

O restante desta seção exemplifica a utilização da transversalidade estática, além de destacar a sintaxe utilizada, definindo o significado e estrutura de type-section-intertype, function-section-intertype, external-section-intertype e abstraction-section-intertype. Desta forma, são apresentados alguns exemplos que modificam a álgebra de alguns módulos

e, posteriormente, outros exemplos que acrescentam novas ações.

Introdução de Tipos e Funções

Em Aspect$\mathcal{M}$, é possível implementar a introdução de novos elementos à álgebra de um módulo. Basicamente, a seção **algebra** define os elementos da álgebra subjacente ao modelo, contendo os *sorts* ou tipos e as funções do módulo [BTI+07]. Em uma definição de tipos, pode haver uma cláusula **default**, que permite estabelecer o valor padrão para inicialização automática de funções do tipo dado.

A Listagem 4.18 ilustra alguns exemplos de introdução de tipos em Aspect$\mathcal{M}$. Basicamente, basta definir o tipo precedido do nome do módulo que receberá o tipo. Neste exemplo, entre as linhas 2 e 5, definem-se que os tipos Undefined, Id, Value e Memory serão introduzidos ao módulo Globals. Observe que, na definição de Memory, considerou-se que os tipos Undefined, Id e Value já pertencem ao módulo Globals.

```
1  aspect:
2    type Globals.Undefined = public enum {UNDEFINED};
3    type Globals.Id = String;
4    type Globals.Value = Bool | Int;
5    public type Globals.Memory = Id -> Value | Undefined;
```

Listagem 4.18: Exemplo de introdução de tipos

A seguir, é apresenta a sintaxe da introdução de tipo de Aspect$\mathcal{M}$:

type-section-intertype ::= [**public**] **type** module-name.type-declarator
[= type-denotation]

Na seção **aspect**, de um módulo em Aspect$\mathcal{M}$, é possível declarar a introdução de funções. Como exemplo, tem-se a Listagem 4.19. Nessa listagem, define-se que as funções error e input serão introduzidas no módulo Globals, enquanto que a função array será introduzida no módulo Structures. Observe que a listagem demonstra a definição de uma função externa, input. Além disso, observe que as funções error e input são de aridade 0, enquanto que a função array possui aridade 1.

```
1  aspect:
2    external Globals.input : list of Int;
3    function dynamic Structures.array : Int -> Int;
4    function dynamic Globals.error : Bool := false;
```

Listagem 4.19: Exemplo de introdução de funções

Assim, em Aspect$\mathcal{M}$, a introdução de funções tem a seguinte sintaxe:

external-section-intertype	::=	**external** module-name.declared-element : type-expression
function-section-intertype	::=	[**public**] [**shared**] **function** function-declar-intertype
function-declar-intertype	::=	[class-modifier] module-name.declared-element : type-expression [:= expression]
declared-element	::=	function-name { , function-name }
	|	function-name (function-parameters)
function-parameters	::=	function-parameter { , function-parameter }
function-parameter	::=	parameter-name : type-expression

Introdução de Ações

Em Machĭna, a seção **abstrations** é destinada a definição de ações, que representam abstrações de regras de transição. Uma ação pode ser usada localmente ou exportada. Além disso, uma ação pode receber qualquer tipo de parâmetro, inclusive outras ações [BTI+07].

Em Aspect$\mathcal{M}$, além de introduzir tipos e funções, é possível introduzir ações. Na Listagem 4.20, define-se que as ações treatEquals e treatRead serão introduzidas no módulo Operations.

```
1  aspect:
2    public action Operations.treatEquals is
3      let x = head(program);
4      let y = head(tail(program));
5      program := (Int(x) = Int(y))::tail(tail(program));
6      opstack := tail(opstack)
7    end
8    public action Operations.treatRead is
9      program := head(input)::tail(program);
10     infile := tail(input)
11   end
```

Listagem 4.20: Exemplo de introdução de ações

Basicamente, basta definir a ação, de forma análoga à Machĭna, porém o nome da ação deverá ser precedido do nome do módulo que receberá a ação. A seguir, é apresentada a sintaxe que define a introdução de ações em Aspect$\mathcal{M}$.

abstraction-section-intertype	::=	[**public**] action-abstraction-intertype [;]
action-abstraction-intertype	::=	**action** action-declaration-intertype
action-declaration-intertype	::=	module-name.action-name [(action-parameters)] **is** action-body **end** [action-name]

4.3 Considerações Finais

A linguagem Machĭna, proposta por Bigonha el al. [BTI+07], e aqui estendida, provê a expressividade necessária à especificação de programas segundo o paradigma de máquinas de estado abstratas(ASM). No entanto, em uma especificação, utilizando a linguagem Machĭna, ainda existia o problema do espalhamento e entrelaçamento de interesses causado pela especificação dos interesses transversais.

A nossa contribuição à linguagem Machĭna consiste no fato desta nova versão prover a capacidade de modularizar esses interesses transversais. Assim, a linguagem proposta acrescenta as vantagens da programação orientada por aspecto (AOP) à linguagem Machĭna. Desta forma, uma das contribuições deste trabalho é estender os conceitos da AOP para ASM.

Basicamente, as novas construções de Machĭna foram baseadas em Eos e AspectJ. Similar a Eos, esta versão utiliza o conceito de classpect, unificando, assim, no contexto da linguagem Machĭna, módulos e aspectos. Semelhante a AspectJ, utilizou-se um modelo de ponto de junção léxico com uma flexibilidade parecida. Além disso, a sintaxe utilizada para definir os conjuntos de junção nesta versão também foi baseada em AspectJ.

Do ponto de vista de usabilidade, no projeto da linguagem, houve uma grande preocupação em deixar as novas construções similares à linguagem Machĭna, de forma que um programador Machĭna não tenha grandes problemas com a nova versão.

Capítulo 5

Implementação de Aspect$\mathcal{M}$

Uma das vantagens do uso do modelo ASM para a especificação da semântica de um algoritmo é que esta especificação pode ser executada. Neste trabalho, utiliza-se a linguagem Aspect$\mathcal{M}$ para definir as especificações formais, possibilitando assim, uma separação avançada de interesses no nível de especificação. Para que tais especificações sejam executadas, faz-se utilização da programação orientada por aspectos através do uso do arcabouço ACOA [Lob06] e algumas classes do arcabouço kℓar [San06],

Desta forma, este capítulo tem por objetivo detalhar como as construções da linguagem Machĭna e suas novas construções são compiladas para código C++ e AspectC++, respectivamente.

5.1 Arcabouços Utilizados

5.1.1 ACOA

O Arcabouço de Compilação Orientado por Aspectos (ACOA) tem por objetivo facilitar a geração de compiladores para novas linguagens, e permitir fácil manutenção dos compiladores quando ocorrem mudanças na definição da linguagem.

Segundo Lobato [Lob06], a decisão para a construção deste arcabouço se deve a dois fatos: (i) obter os benefícios dos *compiler compilers* possibilitando a automatização de etapas de desenvolvimento de um compilador; (ii) obter maior extensibilidade, flexibilidade e modularização nas etapas implementadas pelo usuário, possibilitando maior evolução dos compiladores desenvolvidos.

O ACOA gera compiladores escritos em C++ utilizando a programação orientada por aspectos, via a linguagem AspectC++ [SGSP02], para a implementação da análise semântica e geração de código.

O uso do ACOA para a construção de um compilador é feito do seguinte modo: inicialmente, o usuário constrói um arquivo de especificação que possui em alto nível de

abstração as definições do analisador léxico, sintático, nodos da AST e as declarações dos passos de compilação. Este arquivo é similar aos arquivos usados para os *compiler compilers*. Em seguida, o usuário usa este arquivo de especificação como entrada para o gerador de *front-end*, denominado *FrEG*, para que as partes automáticas sejam geradas. Por fim, o usuário implementa os passos de compilação em métodos das classes de nodos da AST gerados. Esses métodos são implementados nos aspectos e são inseridos nas classes no momento da costura dos códigos.

Assim, inicialmente, definiu-se este arquivo referente à linguagem proposta. Este arquivo de especificação encontra-se dividido em três partes: (i) definições para o analisador léxico (*lexer*) e definições dos nodos da AST para os *Tokens*; (ii) definições para o analisador sintático (*parser*) e dos nodos da AST para as regras gramaticais; (iii) declarações dos passos. O Apêndice A mostra parte do arquivo de especificação para a linguagem proposta. Basicamente, este apêndice apresenta, resumidamente, as novas construções propostas. Maiores detalhes sobre a sintaxe deste arquivo podem ser obtidas em [Lob06];

Dentre as vantagens da utilização do ACOA, destaca-se a modularização do sistema obtida. Desta forma, alterações na definição da linguagem tiveram um impacto pequeno no código não gerado pelo arcabouço. Assim, no desenvolvimento do compilador para a linguagem proposta, observaram-se os seguintes benefícios:

- a criação de um novo passo de compilação, ou alteração na ordem de execução dos passos, ou a remoção de algum passo pode ser realizada sem a necessidade de qualquer alteração do código de passos já implementados. Ao criar um novo passo, foi apenas necessário implementar o novo método para cada nodo da AST.

- a introdução de novas produções na gramática da linguagem que não afetam semanticamente outras produções da mesma pode ser feito sem nenhuma alteração do código anteriormente implementado. Apenas foi necessário implementar os passos de compilação para os novos nodos da AST. Nos casos onde as novas produções da gramática alteraram a semântica de outras já existentes, foi possível saber exatamente os locais que deveriam sofrer essas alterações, pois as implementações estão modularmente separadas.

- alteração de algum passo de compilação para algum nodo da AST demandou alteração apenas no método que possui a implementação deste passo para este nodo.

- existia um compilador para a versão antiga da linguagem Machĭna desenvolvido com este arcabouço. No entanto, devido a algumas limitações, além de gerar uma representação em XML, esta implementação foi, praticamente, descartada. Mas

foi possível reaproveitar o arquivo de entrada para o *FrEG*, ou seja, o arquivo que contém a descrição da gramática, além de algumas classes e passos de compilação, agilizando assim o desenvolvimento.

O ACOA utiliza a linguagem AspectC++ na inserção estática dos métodos responsáveis pela implementação dos passos de compilação e geração de código. Infelizmente, no desenvolvimento do compilador para a linguagem proposta, AspectC++ apresentou algumas falhas de implementação (*bugs*) diante de uma quantidade de métodos a serem inseridos em uma mesma classe. Este problema foi reportado por meio da lista de discussão, aspectc-user@aspectc.org, e deverá ser sanado em uma próxima versão. No entanto, até a data do presente trabalho, não havia uma data estipulada para a liberação de uma nova versão.

5.1.2 Kℓar

Kℓar é um arcabouço otimizador. Como entrada, esse arcabouço recebe uma especificação ASM em um formato conhecido como *Machĭna Intermediate Representation*, abreviadamente MIR. O objetivo do kℓar é realizar transformações na MIR originalmente recebida, e então fornecer como saída uma MIR otimizada. As otimizações a serem aplicadas sobre a especificação ASM são desenvolvidas separadas, e então acopladas ao kℓar.

Segundo Santos [San06], o arcabouço implementa uma coleção de classes para dar suporte à utilização da linguagem MIR. Por meio destas classes, é possível realizar as seguintes tarefas:

Serialização a implementação de MIR permite a sua serialização por meio de arquivos XML em um formato específico. Esta serialização ocorre nos dois sentidos: tanto é possível obter um objeto representativo de uma especificação MIR a partir de sua representação XML, quanto é possível obter a serialização XML a partir de um objeto representativo de uma especificação MIR.

Conversão para código C++ um objeto MIR pode ser convertido em código C++ padrão por meio da simples chamada de um método. O código C++ gerado segue o padrão *visitor* [GHJV95], de modo que o mesmo pode ser compilado em várias plataformas diferentes. Além disso, a linguagem C++ como linguagem alvo torna possível uma execução rápida e eficiente.

Desta forma, uma das funcionalidades do arcabouço kℓar é prover a conversão da linguagem MIR em C++. O trabalho proposto também converte uma linguagem baseada

no modelo ASM, a saber, Machĭna, em C++. Assim, como existe uma grande semelhança entre Machĭna e MIR, algumas classes que o arcabouço kℓar utiliza para realizar as traduções para C++ puderam ser reaproveitadas. Convém ressaltar que algumas se encontravam com problemas e foram refeitas. Além da reutilização das classes, foram utilizadas também algumas idéias de tradução, como a tradução da regra **case** em um conjunto de `if`'s e `else`'s aninhados.

5.1.3 Integração dos Arcabouços

Tanto o arcabouço ACOA quanto o kℓar não constituem projetos isolados. Conforme ilustrado na Figura 5.1, o ACOA compila uma especificação em Machĭna para uma linguagem intermediária e o kℓar compila esta linguagem para C++.

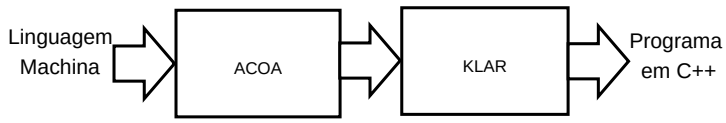

Figura 5.1: Integração dos arcabouços

Desta forma, uma das diferenças com o trabalho proposto, além da modularização dos interesses transversais, é que não existe uma linguagem intermediária no processo de tradução de Aspect$\mathcal{M}$ para C++ e AspectC++. A Figura 5.2 ilustra o processo de tradução deste trabalho.

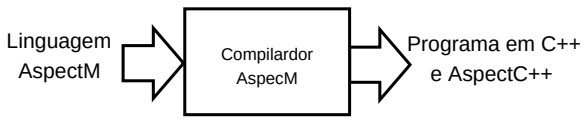

Figura 5.2: Processo de tradução de Aspect$\mathcal{M}$

5.2 Implementação de Machĭna

Considerando-se que o gerador de código produz um programa em C++ para refletir a semântica de uma especificação ASM, é preciso deixar explícito a abordagem utilizada para o mapeamento dos elementos da linguagem Machĭna para entidades da programação orientada por objetos.

Os primeiros elementos a se considerar da linguagem Machĭna são os módulos e os agentes. Assim, no processo de tradução para C++, a compilação de um módulo dá origem a uma classe. Os componentes do módulo, ou seja, as funções, tipos e

abstrações de regras são traduzidos nos membros desta classe. A regra de transição do módulo é um método de nome especial, executeTransitionRule. A Listagem 5.1 mostra um exemplo de interface da classe resultante da compilação de um módulo.

```
1  #include "klar__runtime.h"
2  using namespace std;
3  // Some needed includes
4  #include "Module2.h"
5  #include "Module3.h"
6  ...
7  #include "ModuleN.h"
8  // the base classe of all modules
9  #include "Module.h"
10 class Module1 : public Module
11 {
12   public:
13       // the module constructor
14       Module1(string agName);
15       // the imports
16       Module2* __Module2;
17       Module3* __Module3;
18       ...
19       ModuleN* __ModuleN;
20       // functions, types and actions
21       // start the agent
22       virtual void start();
23   private:
24       // the initial rule
25       void executeInitialState();
26       // the transition rule
27       virtual void executeTransitionRule();
28       // the update list and some auxiliary methods
29 };
```

Listagem 5.1: Exemplo de uma classe criada para representar um módulo genérico.

Toda classe que representa um módulo deve herdar de **Module**, apresentado na Listagem 5.2. Esta classe define o método executeTransitionRule como um método puramente virtual. Desta forma, os agentes podem ser uniformemente tratados como objetos da classe Module, e cada agente executará sua regra de transição corretamente.

Em seguida, é definida uma função de assinatura e tipo de retorno void * runModule(void* param) que cria um agente e então o dispara. Esta função é necessária porque agentes são executados concorrentemente, por meio de *threads*, e uma *threads* na biblioteca de concorrência utilizada é uma função com tal assinatura e tipo de retorno.[1] A implementação desta função pode ser vista na Listagem 5.3.

[1] A biblioteca utilizada é *pthread*, que é a biblioteca de threads padrão de sistemas operacionais *unix-like*.

```
1  #include "klar__runtime.h"
2  class Module : public Klar__Any
3  {
4    public:
5      pthread_t thread;
6      virtual void start() = 0;
7      // some getter's and setter's methods
8
9    protected:
10     string __agentName;
11     int __id;
12     AgentState __state;
13     SuperAgent* __superAgent;
14     Klar__Global *__global;
15
16   private:
17     virtual void executeTransitionRule() = 0;
18 };
19 //------------------------------------------------------------------------
20 void* runModule(void* param);
```

Listagem 5.2: Classe base de um módulo.

```
1  // The function that starts a specific agent
2  void *runModule(void* param)
3  {
4    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
5    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
6    Module* agent = (Module*) param;
7    agent->start();
8    return 0;
9  }
```

Listagem 5.3: A função runModule, que dispara um agente de um módulo.

Um agente, por sua vez, é um objeto do módulo do seu tipo. O disparo de um agente é a execução de seu método start por meio de uma *thread*. A compilação de um módulo de definição Machĭna, o qual cria e dispara os principais agentes, dá origem a uma classe que cria as instâncias dos agentes especificados de acordo com seu tipo e então dispara estes agentes. Todas as operações sobre os agentes são realizadas por um superagente, sendo que sua interface é apresentada na Listagem 5.4. A classe segue o padrão *Singleton* [GHJV95], e por isso nenhum construtor público está disponível. No lugar disto, a referência a um objeto desta classe é obtida por meio do método estático getInstance().

A compilação de um arquivo que representa um módulo de definição Machĭna resulta em um par de arquivos de extensão h e cc de mesmo nome, conforme apresentados nas Listagens 5.5 e 5.6, respectivamente. Após a inclusão dos cabeçalhos dos módulos utilizados, o arquivo de extensão h define a classe Start, cujo papel é instanciar e

```
1  #include "klar__runtime.h"
2
3  class SuperAgent
4  {
5    public:
6      static SuperAgent* getInstance();
7      void createAgent(string, Module*, bool);
8      void dispatchAgent(int agentId, string agentName);
9      void destroyAgent(int agentId, string agentName);
10     void stopAgent(int agentId, string agentName);
11     void waitForPendingAnswers(int, string);
12     void addAnswer(int, string, pair < Klar__Any**, Klar__Any* > );
13     void executeAnswersReceived(int, string);
14     void addRequest(int, string, Klar__Abstraction*);
15     void executeRequestsReceived(int, string);
16     Module* getAgent(int, string);
17     int getNumberOfAgent(string);
18
19   private:
20     // some private atributes and auxiliary methods
21
22 };
```

Listagem 5.4: Classe criada para representar o superagente.

disparar os agentes especificados no disparador de agentes. A classe segue o padrão *Singleton*, similar ao superagente. No momento em que um objeto desta classe é instanciado, é obtida uma referência para o superagente. Assim, por meio da chamada ao método run() desta classe, os agentes são criados e disparados. O arquivo de extensão cc é o arquivo principal que dará origem ao executável, pois é nele que é definida a função main. Esta função obtém uma instância do contexto dos agentes e então dispara estes agentes. O método run(), por meio do superagente, dispara a execução da regra de transição dos agentes criados como *threads* independentes.

```
1  #include "klar__runtime.h"
2  #include "Module1.h"
3  class Start {
4    public:
5      static Start* getInstance();
6      void run();
7    private:
8      Start();
9      static Start* start;
10     static SuperAgent* superAgent;
11 };
```

Listagem 5.5: O arquivo de extensão h gerado a partir de um módulo de definição Machĭna.

```
1  #include "Start.h"
2  //------------------------------------------------------------------------
3  Start* Start::start = 0;
4  SuperAgent* Start::superAgent = 0;
5  //------------------------------------------------------------------------
6  Start* Start::getInstance() {
7    superAgent = SuperAgent::getInstance();
8    if (start == 0) start = new Start();
9    return start;
10 }
11 //------------------------------------------------------------------------
12 void Start::run() {
13   Module1* module_1 = new Module1("module_1");
14   superAgent->createAgent(module_1);
15   superAgent->dispatchAgent(module_1);
16   superAgent->joinAgent(module_1);
17   return;
18 }
19 //------------------------------------------------------------------------
20 int main(int argc, char *argv[]) {
21   Start* start = Start::getInstance();
22   start->run();
23   return 0;
24 }
```

Listagem 5.6: O arquivo de extensão cc gerado a partir de um módulo de definição Machĭna.

A compilação de um módulo, também, dá origem a um par de arquivos de extensões h e cc, cujas estruturas são semelhantes àquelas apresentadas nas Listagens 5.1, página 91 e 5.7, respectivamente. No arquivo com extensão h, a definição da classe é cercada apropriadamente por expressões de compilação condicional, de modo a permitir definições de classes mutuamente recursivas. No entanto, estes detalhes de implementação foram retirados das listagens para simplificar o seu entendimento. Assim, os primeiros elementos da definição são os *includes* necessários à compilação deste módulo. Estes *includes* consistem nos arquivos de cabeçalho dos módulos referenciados ou importados por este módulo, e esta informação é dada pelas listas de módulos referenciados e importados que cada módulo possui em sua definição.

A seguir é iniciada a definição da classe em si que representa o módulo. O nome desta classe é o nome do módulo, e esta herda da classe abstrata Module, presente no ambiente de execução, que provê comportamentos comuns a qualquer módulo. Além disso, esta estratégia também fornece ao módulo uma interface mínima comun a todos os módulos. Agentes são objetos desta classe. O primeiro método a ser definido na interface da classe é o construtor, que recebe como argumento um nome para identificar o agente em um contexto global. Convém destacar que um agente é identificado por um número e um nome, sendo que este número é um seqüencial controlado pelo

```
#include "Module1.h"
//---------------------------------------------------------------------
Module1::Module1(std::string agName) {
  __Module2 = new Module2(agName);
  __Module3 = new Module3(agName);
  srand(time(NULL));
  __agentName = agName;
  // iniciaizações
}
//---------------------------------------------------------------------
void Module1::executeInitialState() {
  // regras de iniciaização
}
//---------------------------------------------------------------------
void Module1::executeTransitionRule() {
  // regras de transição
}
//---------------------------------------------------------------------
// alguns métodos auxiliares
//---------------------------------------------------------------------
void Module1::start(){
  executeInitialState();
  while (getAgentState() != DESTROYED) {
    if (getAgentState() == ACTIVE) {
      executeTransitionRule();
      executeAgentRules();
      executeUpdateList();
    }
    // código para sincronização ...
  }
}
```

Listagem 5.7: Exemplo de uma classe criada para representar um módulo.

superagente. Este contexto possibilita que um agente acesse as funções dinâmicas de outro agente, o que caracteriza a memória compartilhada.

A composição com outros módulos é feita por meio da agregação. Exemplificando, seja *Module1* um módulo que inclua os módulos *Module2* e *Module3*. Então a classe gerada a partir da definição de *Module1* possui um campo cujo tipo é a classe gerada a partir da definição do módulo *Module2* e um campo cujo tipo é a classe gerada a partir da definição do módulo *Module3*.

O ambiente do módulo é então declarado. Abstrações de regras são declaradas como métodos que retornam void, enquanto funções são declaradas como métodos de retorno diferentes de void. Os detalhes de como cada um destes métodos é implementado de fato para cada tipo de elemento do ambiente são apresentado mais à frente.

5.2.1 Sistema de Tipos

Baseado no conjunto de classes oferecidas pelo arcabouço kℓar, as classes utilizadas para representação dos tipos são organizadas em uma hierarquia onde existe um classe abstrata básica, Type, a partir da qual todas as classes de tipos devem derivar.

O mapeamento entre os tipos da linguagem Machĭna e os tipos em C++ é dado na Tabela 5.1. Apesar dos ajustes e modificações na estrutura das classes, optou-se por manter os nomes advindos do arcabouço kℓar. Convém destacar que as classes Klar__Enum, Klar__Input e Klar__Output não existiam no arcabouço, mas manteve-se o prefixo Klar para uma padronização de nomes.

Linguagem Machĭna	**C++**
Bool	Klar__Boolean
Char	Klar__Character
Int	Klar__Integer
Real	Klar__Real
String	Klar__String
Enum	Klar__Enum
Disjoint Union	Klar__Any
$(T_1, ..., T_n)$	Klar__Tuple
Set	Klar__Set
List	Klar__List
Node	Klar__Node
Input	Klar__Input
Output	Klar__Output
Agent	Module

Tabela 5.1: Mapeamento de tipos na compilação da linguagem Machĭna para C++

5.2.2 Funções e Relações

Funções são compiladas diferentemente de acordo com seu tipo. Funções derivadas são compiladas em métodos segundo o exemplo apresentado na Listagem 5.8. A passagem de parâmetros, do ponto de vista da função, é sempre por cópia, pois funções não possuem efeitos colaterais.

A compilação das funções dinâmica ou estática produz como resultado uma tabela hash que associa os pontos do domínio da função aos valores da função nestes pontos e também um método acessor para esta tabela, conforme apresentado na Listagem 5.9. Cabe ressaltar que, desta forma, para cada função, dinâmica ou estática, definida no módulo, existe uma tabela hash correspondente, que é definida na classe do módulo e inicializada em seu construtor. A exceção acontece para funções 0-árias, que fazem uso

```
1 tipoRetorno derived_nomeFuncao(tipoParam &p1,...,tipoParamN &pN){
2     // código equivalente a expressão que define a função
3     tipoRetorno v = expValor;
4     return v;
5 }
```

Listagem 5.8: Resultado da compilação de uma função derivada

de uma variável membro cujo tipo é o tipo de retorno da função. Existe também um método acessor para cada função dinâmica ou estática. A função do método acessor no lugar do acesso direto se justifica pelo fato que tais funções podem ter um valor padrão associado. Assim, o método acessor, antes de retornar o valor em um ponto da tabela hash, verifica se o ponto já foi definido por uma atribuição. Caso esta definição não tenha acontecido, o método acessor retorna o valor padrão definido na declaração da função ou o valor padrão do tipo de retorno da função.

```
1  // declaração da função dinâmica ou estática
2  map<tipoParam1,...,map<tipoParamN, tipoRetorno>...>> __nomeFuncao;
3
4  // método acessor para a função dinâmica
5  tipoRetorno dynamic_nomeFuncao(tipoParam &p1,...,tipoParamN &pN){
6      if (nomeFuncao.find(p1,..,pN)) return nomeFuncao[p1]..[pN];
7      else return valorPadrao;
8  }
9
10 // método acessor para a função estática
11 tipoRetorno static_nomeFuncao(tipoParam &p1,...,tipoParamN &pN){
12     if (nomeFuncao.find(p1,..,pN)) return nomeFuncao[p1]..[pN];
13     else return valorPadrao;
14 }
```

Listagem 5.9: Resultado da compilação de uma função dinâmica ou estática.

Funções externas são compiladas para métodos que apenas fazem a chamada a funções definidas externamente por meio de uma interface.

```
1 extern tipoC++ MNI__nomeFuncao(tipoParam &p1,...,tipoParamN &pN);
2
3 tipoRetorno external_nomeFuncao(tipoParam &p1,...,tipoParamN &pN){
4   tipoRetorno value = MNI__nomeFuncao(p1,...,pN);
5   return value;
6 }
```

Listagem 5.10: Resultado da compilação de uma função externa

Por fim, funções podem ser declaradas como **shared** para serem compartilhada por diversos agentes. Desta forma, uma função declarada com **shared** não pertence

ao agente e sim ao módulo. Assim, as funções desse tipo são traduzidas para funções estáticas de C++ de acordo com as traduções supra mencionadas.

5.2.3 Abstrações de Regras

Ações são traduzidas para métodos da classe correspondente ao módulo. Uma ação unitária é compilada para um método à semelhança daquele apresentado na Listagem 5.11, com tipo de retorno void. Os parâmetros podem ser de entrada ou de entrada e saída, o que altera a forma como os parâmetros são passados, se por cópia ou por referência. Esta classificação dos parâmetros é dada pela declaração da abstração da linguagem Machĭna, Seção 3.2.4.

```
1  void nomeAcao(tipoParam1 &p1, ..., tipoParamN &pN){
2      // código equivalente à regra da ação
3      executeAgentRules();
4      executeUpdateList();
5      // código para sincronização ...
6  }
```

Listagem 5.11: Resultado da compilação de uma ação unitária.

Ações repetitivas são convertidas em métodos e possuem uma estrutura semelhante àquela apresentada na Listagem 5.12, também com tipo de retorno void. Diferentemente, uma chamada a uma ação repetitiva dispara uma regra de transição que é executada repetidamente até que uma regra *return* seja encontrada, cuja semântica é fazer __return__submachine = true, o que causa o término da execução da ação.

```
1  void nomeAcao(tipoParam1 &p1, ..., tipoParamN &pN){
2      bool __return__submachine = false;
3      while (!__return__submachine){
4          // código equivalente à regra da ação
5          executeAgentRules();
6          executeUpdateList();
7          // código para sincronização ...
8          }
9      }
10 }
```

Listagem 5.12: Resultado da compilação de uma ação repetitiva

5.2.4 Regras de Transição de Estado

Uma das utilidades do arcabouço kℓar é converter um programa escrito na linguagem MIR para código C++. A linguagem MIR é uma linguagem intermediária de uso

geral para compiladores de especificações ASM. Devido à semelhança com a linguagem Machĭna, as traduções realizadas para as regras de transição foram baseadas nas traduções realizadas pelo arcabouço kℓar.

```
1  void Module1::executeTransitionRule()
2  {
3    if (__step == 1) {
4      // regra de transição ...
5    }
6    if (__step == 2) {
7      // regra de transição ...
8    }
9    if (__step == 3) {
10     // regra de transição ...
11   }
12   if (__step == 4) {
13     // regra de transição ...
14   }
15 }
```

Listagem 5.13: Tradução de uma regra de transição de múltiplos passos

A transição principal de um módulo ou do corpo de abstrações definidas em um módulo pode ser uma regra de transição de um único passo ou então de uma seqüência de passos de execução. A tradução de uma regra de transição de um único passo é semelhante ao código apresentado na Listagem 5.7, linhas 15 a 17, página 95. No caso da transição de múltiplos passos, cada um dos passos indicados pelas cláusulas **step** é traduzido para um comando if de C++. Além disso, as variáveis **step** e **next** são traduzidas para váriaveis inteiras de C++, como apresentado na Listagem 5.13.

Nas Listagens 5.14, 5.15 e 5.16 são apresentados exemplos de tradução das regras básicas. Uma regra de atualização de função, Listagem 5.14, é traduzida em uma inserção na lista de atualizações de um par constituído de uma localização e o valor a ser atribuído a esta localização. Esta lista é processada ao final da transição, por meio do comando executeUpdateList(), e é este artifício que permite simular o paralelismo intrínseco de uma regra de transição em um programa seqüencial.

A tradução de um bloco de regras, $R_1; R_2$, resulta da tradução seqüencial das duas regras que o compõem. O efeito de paralelismo é percebido por meio da realização de atualizações via inserções na lista de atualizações a ser processada no final da iteração. A tradução de uma regra de abreviação de termos(*let*) não produz nenhum código, apenas cria a associação de um identificador a sua expressão. Quando essa variável for utilizada o código referente a sua expressão será gerado.

Uma regra *if* é traduzida para um comando if de C++, onde a guarda é avaliada como a expressão do if e os códigos para as regras correspondentes aos braços then e else são traduzidos dentro dos blocos correspondentes ao then e else do if. A tradução da

```
1  //atualização de função
2  this->addNewUpdate((Klar__Any**)&__function, value) ;
3
4  //bloco de regras
5  /*código equivalente a regra 1*/;
6  /*código equivalente a regra 2*/;
7
8  //abreviatura de termos
9
10 //if
11 if (exp) {... /*código equivalente ao then*/}
12 else {... /*código equivalente ao else*/}
13
14 //case
15 tipoExp caseExp = ... /* avaliação da expressão */
16 if (caseExp == exp_1){
17   // regra correspondente
18 }else if (caseExp == exp_2){
19   // regra correspondente
20 }else{
21   // caso exista, código correspondente a regra padrão
22 }
```

Listagem 5.14: Resultado da compilação para algumas regras

regra *case* resulta primeiramente na avaliação da expressão do *case* e sua associação a um identificador. Em seguida, cada alternativa do *case* é traduzida em um comando if que verifica se a expressão avaliada é igual ao valor correspondente àquela alternativa. Todos os if são devidamente entremeados por um else, de modo a garantir a exclusão mútua da execução das alternativas. Por fim, a regra padrão, caso exista, é traduzida dentro de um bloco else sem guarda, que será executado caso todas as guardas anteriores falhem.

A tradução da regra *with*, Listagem 5.15, é semelhante à tradução da regra *case*. A principal diferença é que a comparação para escolha da alternativa a ser executada se baseia não no valor da expressão avaliada, mas sim no seu tipo. Esta comparação é feita por meio de uma chamada à função typeEquivalent, disponibilizada pela biblioteca de *runtime* do arcabouço kℓar.

Para a tradução da regra *choose*, inicialmente, é gerada uma variável que indica se foi encontrado um elemento que satisfaça a condição dada. Em seguida, a expressão que denota o conjunto é avaliada e associada a um identificador. É também criado um conjunto com uma seqüência aleatória de números que indica quais elementos já foram examinados. Dentro deste contexto, inicia-se a escolha, que acontece dentro de um comando while que se repete enquanto não for achado um elemento que atenda à condição especificada e enquanto todos os elementos não forem inspecionados. Cada elemento escolhido é associado a uma variável e a condição é avaliada. Caso esta

```
1  //with
2  tipoExp withExp = ... /* avaliação da expressão */
3  if (typeEquivalent(withExp,tipo1)){
4    // faz o type cast apropriado seguido da regra correspondente
5  }else if (typeEquivalent(withExp,tipo2)){
6    // faz o type cast apropriado seguido da regra correspondente
7  }else{
8    // caso exista, código correspondente a regra padrão
9  }
10
11 //choose
12 bool __found = false;
13 /* exp que define o conjunto */
14 tipoConjuntoOuLista name__domain = ...
15 /* ordem em que os elementos vão ser escolhidos */
16 vector<int> name__ordem = ...
17 /* índice para acessar o vetor */
18 int i__name = 0;
19 while (!__found && i__name < name__domain.size()) {
20     // pega o elemento da posição name__ordem[i__name]
21     // associa o elemento escolhido a um alias
22     // verifica se a condição é satisfeita
23     if (.../* expressão da condição */) {
24         __found = true;
25         // regra do choose
26     }else {
27         // incrementa o indice
28     }
29 }
```

Listagem 5.15: Resultado da compilação para algumas regras

condição seja verdadeira, o indicador de encontrado é marcado como verdadeiro, e a regra do *choose* é executada uma única vez. Caso contrário, o índice do elemento inspecionado é incrementado.

A tradução da regra de universalização(*forall*), Listagem 5.16, resulta primeiramente em uma expressão que denota o conjunto do *forall*. Essa expressão é associada a um identificador. Basicamente, cada elemento do conjunto é associado a um *alias* e, em seguida, a regra de transição é executada. O *forall* tem a semântica de disparar em paralelo a sua regra de transição para cada elemento do conjunto. Apesar da execução ser seqüencial, o efeito de simultaneidade aparente é conseguido porque as modificações no ambiente só são percebidas na próxima iteração, quando a lista de atualizações já tiver sido processada.

A chamada de abstração é encapsulada em um objeto da classe Klar__Abstraction. Convém ressaltar que a classe fornecida pelo arcabouço kℓar não atendia aos critérios existente na linguagem Machĭna para sincronização dos agentes. Desta forma, esta classe foi totalmente refeita, conservando o nome por questões de padronização. Assim,

```
1  //universalização
2  /* expressão que define o conjunto */
3  tipoConjuntoOuLista name__domain = ...
4  for (int i__name=0; i__name<name__domain.size(); i__name++) {
5      // pega o elemento da posição i__name
6      // associa o elemento a um alias
7      // regra do forall
8  }
9
10 //chamada de abstrações
11 Klar__Abstraction call;
12 /*define qual agente solicitou o pedido */
13 call.setModule(...);
14 /*define qual agente irá atender o pedido*/
15 call.setAgent(...);
16 /*define o nome da abstração(método) a ser chamada*/
17 call.setFunction(...);
18 /*caso exista, define os parâmetros para a abstração*/
19 call.addParameter(...);
20 superAgent->addRequest(__id, __agentName, call);
```

Listagem 5.16: Resultado da compilação para algumas regras

um objeto dessa classe armazena o agente que solicitou o pedido, o agente que irá atender ao pedido, o nome da abstração e seus respectivos parâmetros, caso existam. O superagente fica responsável por controlar a execução das abstrações. Os detalhes de como este controle é realizado de fato é apresentado na Seção 5.2.6.

5.2.5 Expressões

Semelhante a tradução do sistema de tipos e das regras de transição, na tradução das expressões foram utilizadas as classes e/ou idéia fornecidas pelo arcabouço kℓar, quando aplicadas. Deste modo, nas Listagens 5.17, 5.18, 5.19 e 5.20 são apresentados exemplos de tradução das expressões.

A tradução de um literal, Listagem 5.17, resulta na declaração de um valor do tipo do literal. A expressão *self* é traduzida para uma chamada ao método que retorna a instância do agente, ao passo que *undef* é mapeado para o valor NULL. Um operador unário segue o seguinte padrão na tradução: primeiro o seu operando é avaliado, e em seguida, o operador propriamente dito é aplicado ao operando. A tradução de um operador binário resulta na avaliação do primeiro operando, seguido pela avaliação do segundo operando. De posse dos valores dos dois operandos, o operador é aplicado. Uma aplicação de função é traduzida para a avaliação de cada parâmetro atual, seguida da chamada ao método que está associado à função.

Um agregado, Listagem 5.18, é traduzido para a seguinte seqüência de comandos: primeiramente, a expressão de cada componente é avaliada. Em seguida, o literal do

```
1  // literais
2  Klar__Integer t1(0);
3
4  // self
5  tipoDoAgente* t1 = self();
6
7  // undef
8  Klar__Any *t1 = NULL;
9
10 // unary op (neste exemplo, uma inversão de sinal)
11 Klar__Integer* t1 = ... ; // avalia a expressão do operando
12 Klar__Integer t2 = ( - (*t1)); // o resultado está disponível em t2
13
14 // binary op (neste exemplo, uma soma)
15 Klar__Real* t1 = ... ; // avalia a expressão do primeiro operando
16 Klar__Real* t2 = ... ; // avalia a expressão do segundo operando
17 Klar__Real t3 = ((*t1) + (*t2)); // o resultado está disponível em t3
18
19 // aplicação de funções
20 tipoParametro1 t1 = ... ; // avalia expressão do primeiro parâmetro
21 ...
22 tipoParametro1 tN = ... ; // avalia expressão do ultimo parâmetro
23 // chama a função, disponibilizando o resultado em tN+1
24 Klar__Real tN+1 = funcao((t1), ..., (tN));
```

Listagem 5.17: Resultado da compilação para algumas expressões

agregado é declarado, e cada componente é adicionado. Uma expressão *if* é traduzida de forma semelhante à tradução de uma regra if. A diferença é que neste caso existe um cuidado extra com a obtenção do valor da expressão if. Inicialmente, é criada uma variável com o tipo de retorno da expressão. Em seguida, a guarda é avaliada, e então esta guarda é testada em um comando if de C++. A expressão correspondente ao braço then da expressão é avaliada dentro do bloco correspondente ao then do comando if. No final da avaliação, o resultado se torna disponível externamente por meio da atribuição deste à variável declarada externamente com o tipo da expressão. A geração de código para o braço else é análoga.

Na tradução de uma expressão *case*, Listagem 5.19, inicialmente, é criada uma variável com o tipo de retorno da expressão para conter o resultado final. Em seguida, a guarda é avaliada, e o resultado de sua avaliação é utilizado para determinar qual é a alternativa escolhida. As alternativas são traduzidas para comandos if em C++, devidamente entremeados com comandos else que garantem a exclusão mútua na escolha das alternativas. Dentro do bloco correspondente a cada alternativa, o código para a avaliação da expressão é gerado, e o resultado desta avaliação é disponibilizado externamente por meio da atribuição à variável declarada para conter o resultado final. Além disso, caso exista uma expressão padrão(*otherwise*), é gerado um bloco else que corresponde ao caso em que nenhuma alternativa foi escolhida.

```
1  // agregados (neste exemplo, uma lista)
2  tipoElemento1 t1 = ... ; // avalia expressao do primeiro elemento
3  ...
4  tipoElemento1 tN = ... ; // avalia expressao do ultimo elemento
5  Klar__List tN+1();       // declara a lista, diponivel em tN+1
6  tN+1.push_back(t1);      // adiciona o primeiro elemento
7  ...
8  tN+1.push_back(tN);      // adiciona o ultimo elemento
9
10 // expressão if
11 tipoIfExp* t1;           // tipo de retorno da expressão if
12 Klar__Boolean t2 = ... ; // avalia a guarda
13 if (t2.getValue()) {
14      t3 = ... ;          // avalia a parte "then" da expressao
15      t1 = t3;            // atribui resultado da avaliacao
16 } else {
17      t4 = ... ;          // avalia a parte "else" da expressao
18      t1 = t4;            // atribui resultado da avaliacao
19 }
20 ... // neste ponto, o valor da expressao esta disponivel em t1
```

Listagem 5.18: Resultado da compilação para algumas expressões

A tradução da expressão *with*, Listagem 5.20, é semelhante à tradução da expressão *case*. A principal diferença é que a comparação para escolha da alternativa a ser executada se baseia não no valor da expressão avaliada, mas sim no seu tipo. Esta comparação é feita por meio de uma chamada à função typeEquivalent, disponibilizada pela biblioteca de *runtime* do arcabouço kℓar.

Na tradução da expressão *all*, Listagem 5.21, inicialmente, é criada uma variável com o tipo booleano, inicializada com o valor verdadeiro, para conter o resultado final. Em seguida, a expressão que denota o conjunto é avaliada e associada a um identificador. Dentro deste contexto, todos os elementos do conjunto são testados. Caso seja encontrado um elemento que não satisfaça a propriedade, a variável booleana receberá o valor falso e o laço será encerado. A tradução da expressão *exist* é semelhante a essa. As únicas diferenças são que a variável booleana é inicializada com o valor falso e caso seja encontrado um elemento que satisfaça a condição, essa variável receberá o valor verdadeiro.

A tradução da expressão *promise*, Listagem 5.22, é semelhante a tradução da regra de chamada de abstração (vide 5.2.4) . A única diferença é que se configura o nome da função de tipo **Promisse** que deverá ser sinalizada após o termino da execução da chamada. Cabe ressaltar que a declaração de uma função do tipo **Promisse**, cria uma entrada em uma tabela que mapea nomes de função em valores booleanos. Assim, inicialmente, toda função deste tipo é adicionada a uma tabela mapeando seu nome no valor `false`. Quando uma chamada de abstração de uma expressão *promise* termina a

```
1  // expressão case
2  tipoCaseExp* t1;                // tipo de retorno da expressão case
3  tipoExpressaoGuarda t2 = ... ; // avalia expressao a ser comparada
4  if (t2 == ... /* expressao 1*/) {
5      // avalia a expressao correspondente
6      t3 = ... ;
7      // torna o resultado da avaliacao disponivel em t1
8      t1 = t3;
9  } else if (t2 == ... /* expressao 2*/) {
10     // avalia a expressao correspondente
11     t3 = ... ;
12     // torna o resultado da avaliacao disponivel em t1
13     t1 = t3;
14 }
15 ...
16 } else {
17     // expressao padrao caso exista
18     t3 = ... ;
19     // torna o resultado da avaliacao disponivel em t1
20     t1 = t3;
21 }
22 ... // neste ponto, o valor da expressao esta disponivel em t1
```

Listagem 5.19: Resultado da compilação para expressão case

sua execução, este valor é modificado para `true`. A função `complete(functionName)` permine verificar este valor.

5.2.6 Administração de Agentes

Em Machĭna, os agentes podem ser criados, disparados e mesmo destruídos por outros agentes. Para administrar a execução dos agentes da máquina abstrata, são utilizadas as regras *stop, return, create, dispatch* e *destroy*. A Listagem 5.23 apresenta a tradução dessas regras.

Todo agente tem um atributo implícito state, do tipo State que indica o estado corrente do agente. Assim, os possíveis estados de um agente são: **anew, active, stopped, blocked** e **destroyed**.

Com exceção da regra *return*, todas estas regras são traduzidas em entradas em uma lista de atualizações, exclusiva para agentes, de um par constituído de um agente e um código que identifica a regra. Esta lista é processada ao final da transição, por meio do comando executeAgentRules(), semelhante a execução de uma regra de transição. O superagente controla a execução de cada uma destas regras, para evitar problemas decorrentes de disputa por recursos (*race condition*).

A execução da regra *create* cria o agente com o tipo adequado, a saber, o seu módulo, recebendo o seu nome e um seqüencial fornecido pelo superagente. Em seguida, o agente

```
1  // expressão with
2  tipoWithExp* t1;                // tipo de retorno da expressão with
3  tipoExpressaoGuarda t2 = ... ; // avalia expressao a ser comparada
4  if (typeEquivalent(t2, type1)) {
5      // avalia a expressao correspondente
6      t3 = ... ;
7      // torna o resultado da avaliacao disponivel em t1
8      t1 = t3;
9  } else if (typeEquivalent(withExpression, type2)) {
10     // avalia a expressao correspondente
11     t3 = ... ;
12     // torna o resultado da avaliacao disponivel em t1
13     t1 = t3;
14 }
15 ...
16 } else {
17     // expressao padrao caso exista
18     t3 = ... ;
19     // torna o resultado da avaliacao disponivel em t1
20     t1 = t3;
21 }
22 ... // neste ponto, o valor da expressao esta disponivel em t1
```

Listagem 5.20: Resultado da compilação para expressão with

é colocado em um contexto comum a todos os agentes. Convém ressaltar que todo agente criado possui estado anew. Note que o agente é criado e situado no contexto, mas ainda não é disparado. Para tanto, a execução da regra *dispatch* coloca o agente no estado active, o que dispara a execução do mesmo. Um agente pode ser interrompido por meio da execução da regra *stop*, o que torna seu estado igual a stopped. Por fim, a execução da regra *destroy* muda o estado do agente para destroyed. Ambas as regras *stop* e *destroy* interrompem a execução do agente, porém um agente uma vez destruído é incapaz de responder requisições de outros agentes e de ser redisparado.

A execução de uma ação iterativa pode ser finalizada ao se executar a regra *return*, cuja tradução resulta em uma atribuição de *true* à variável reservada __return__submachine. Esta variável é consultada no início de cada iteração para verificar se a ação deve ou não ser executada mais uma vez.

Comunicação entre Agentes

Como mencionado na Seção 3.2.7, os agentes podem se comunicar via chamadas a abstrações de regras(ações) que são anunciadas na interface dos módulos principais dos agentes. Durante a execução de sua regra de transição, um agente a pode solicitar a execução de uma abstração de regra disponibilizada pela sua interface por um outro agente b, sendo que esta execução pode ser síncrona ou assíncrona.

```
1  // expressão all
2  Klar__Boolean* t1 = new Klar__Boolean(true);
3  tipoConjuntoOuLista name__domain = ... // exp que define o conjunto
4  for (int i__name=0; i__name<name__domain.size(); i__name++) {
5      // pega o elemento da posição i__name
6      // associa o elemento a um alias
7      if (alias não satisfaz a propriedade)
8        t1 = new Klar__Boolean(false);
9  }
10 ... // neste ponto, o valor da expressao esta disponivel em t1
11
12
13 // expressão exist
14 Klar__Boolean* t1 = new Klar__Boolean(false);
15 tipoConjuntoOuLista name__domain = ... // exp que define o conjunto
16 for (int i__name=0; i__name<name__domain.size(); i__name++) {
17     // pega o elemento da posição i__name
18     // associa o elemento a um alias
19     if (alias satisfaz a propriedade)
20       t1 = new Klar__Boolean(true);
21 }
22 ... // neste ponto, o valor da expressao esta disponivel em t1
```

Listagem 5.21: Resultado da compilação para as expressões exist e all

```
1  //expressão promisse
2  Klar__Abstraction call;
3  /*define qual agente solicitou o pedido */
4  call.setModule(...);
5  /*define qual agente irá atender o pedido*/
6  call.setAgent(...);
7  /*define o nome da abstração(método) a ser chamada*/
8  call.setFunction(...);
9  /*caso exista, define os parâmetros para a abstração*/
10 call.addParameter(...);
11 /*define que é uma chamada de expressão promisse*/
12 call.setFunctionPromisse(functionName);
13 superAgent->addRequest(__id, __agentName, call);
```

Listagem 5.22: Resultado da compilação para expressão promise

Execução Síncrona: o protocolo de comunicação entre os agentes impõe que se durante a execução de sua regra de transição, um agente *a* faz uma chamada a uma abstração de um outro agente *b*, os dados relativos a este ato são capturados pelo superagente, e o agente a prossegue imediatamente a execução de sua regra de transição. O efeito desta chamada somente se fará sentir na próxima transição, que somente será executada se as requisições produzidas pela chamada da abstração tiverem sido atendidas.

Seguindo a especificação da linguagem Machĭna [BTI+07] para comunicação dos agentes, todo agente possui internamente um contador, denominado *PendingAnswers*

```
1  //create
2  this->addNewAgentRule(someAgent,CREATE);
3
4  //dispath
5  this->addNewAgentRule(someAgent,DISPATH);
6
7  //destroy
8  this->addNewAgentRule(someAgent,DESTROY);
9
10 //stop
11 this->addNewAgentRule(someAgent,STOP);
12
13 //return
14  __return__submachine = true;
```

Listagem 5.23: Resultado da compilação para regras aplicadas a agentes

e as filas *RequestsReceived* e *AnswersReceived*. Estas estruturas encontram-se armazenadas em uma tabela *hash* e podem ser recuperadas pelo identificador do agente, a saber, seu nome e código seqüencial.

A Listagem 5.24 apresenta um agente a realizando uma chamada síncrona a uma abstração de um outro agente b. Por questões de simplicidade, omitiram-se alguns detalhes.

```
1 Klar__Abstraction call;
2 call.setModule("A");
3 call.setAgent("b");
4 call.setFunction("teste");
5 superAgent->addRequest(1, "a", call);
```

Listagem 5.24: Exemplo de chamada síncrona de abstração

Primeiramente, a tradução da chamada de abstração é encapsulada em um objeto da classe Klar__Abstraction. Em segundo, este objeto é passado para o superagente. Agora, a partir destas informações, o superagente insere um pedido de execução da abstração na fila *RequestsReceived* do agente b e o contador *PendingAnswers* do agente a é incrementado de uma unidade pelo superagente. O contador será decrementado quando a resposta ao pedido do agente a for instalado em sua fila de *ReceivedAnswers*.

O código que realiza este sincronismo é apresentado na Listagem 5.25. Desta forma, é verificado se o agente em execução possui alguma resposta pendente, ou seja, uma lista de atualizações a serem realizadas e/ou uma lista que requisições a serem executadas. Assim, antes de reiterar a execução de sua regra de transição, o agente retira uma a uma as atualizações contidas na sua fila *ReceivedAnswers* e realiza, na ordem da fila, as atualizações indicadas. Em seguida retira e executa, uma a uma, em ordem, todas as abstrações relacionadas na fila *RequestsReceived*.

```
1 // código para sincronização
2 while (__superAgente->getPendingAnswers(__id, __agentName)) {
3     __superAgente->executeAnswersReceived(__id, __agentName);
4     __superAgente->executeRequestsReceived(__id, __agentName);
5     sched_yield();
6 }
7 sched_yield();
```

Listagem 5.25: Código para sincronizar a comunicação entre os agentes

Execução Assíncrona: estas chamadas são ditas asssíncronas porque a próxima transição do agente que requisitou a chamada não precisa aguardar até que seu pedido tenha sido atendido. A Listagem 5.26 apresenta um agente *a* realizando uma chamada asssíncrona a uma abstração de um outro agente *b*, sendo *readyCall* uma função de tipo **Promise**.

Semelhante a uma chamada síncrona, a chamada assíncrona de uma abstração é encapsulada em um objeto da classe Klar__Abstraction e este objeto é passado para o superagente. Assim, a partir destas informações, o superagente insere um pedido de execução da abstração na fila *RequestsReceived* do agente *b*. Agora, ao em vez de incrementar o contador *PendingAnswers* do agente *a*, o superagente cria uma entrada em uma tabela que mapea o nome da função de tipo **Promise** em um valor booleano, inicialmente configurado com `false`. Desta forma, quando o agente *b* terminar a execução da abstração, este valor será modificado para `true`.

```
1 Klar__Abstraction call;
2 call.setModule("A");
3 call.setAgent("b");
4 call.setFunction("teste");
5 call.setFunctionPromisse("readyCall");
6 superAgent->addRequest(1, "a", call);
```

Listagem 5.26: Exemplo de chamada assíncrona de abstração

5.3 Implementação das Novas Construções

Como destacado no Capítulo 4, as novas construções da linguagem Machĭna consistem em: uma nova seção, denominada **aspect**, e um mecanismo de composição, a transversalidade, constituída de conjuntos de junção, adendos e construções para afetar estaticamente a estrutura dos módulos.

Para implementar a nova seção, modificou-se o arquivo de entrada do arcabouço ACOA, o qual gera a partir de outros arcabouços as ASTs referentes aos módulos.

Deste modo, esta seção tem por objetivo detalhar a implementação do mecanismo de composição utilizado na linguagem, ou seja, a transversalidade dinâmica e estática.

5.3.1 Transversalidade Dinâmica

Considerando-se que o gerador de código produz um aspecto em AspectC++, para refletir a semântica das novas construções, é preciso deixar explícito a abordagem utilizada para o mapeamento dos novos elementos, e. g., conjuntos de junção e adendos, para construções da linguagem AspectC++.

A Listagem 5.27 apresenta o corpo de um aspecto gerado para representar as declarações de conjuntos de junção e adendos. Cabe ressaltar que o código dos aspectos são cercados apropriadamente por expressões de compilação condicional. No entanto, estes detalhes de implementação foram retirados das listagens para simplificar o seu entendimento.

```
1  #include "klar__runtime.h"
2
3   //Alguns includes
4  #include "Module2.h"
5  #include "Module3.h"
6  ...
7  #include "ModuleN.h"
8
9   //the aspect itself
10 aspect ModuleAspect {
11
12   public:
13      // declaração de conjuntos de junção e adendos
14
15
16 };
```

Listagem 5.27: Exemplo de um aspecto gerado

5.3.1.1 Conjuntos de Junção

Os primeiros elementos a se considerar da nova versão são os conjuntos de junção. Esses podem ser compostos pelos operadores, **and**(interseção), **or**(união) e **not**(diferença), a fim de criar outros conjuntos de junção. A tabela a seguir apresenta a correspondência entre os operadores da linguagem Aspect$\mathcal{M}$ e AspectC++.

Aspect$\mathcal{M}$	**AspectC++**
not	!
and	&&
or	||

Além dos operadores de conjuntos, Aspect$\mathcal{M}$ possibilita a utilização de alguns símbolos curinga (*wildcard*) na construção de um conjunto de junção, provendo assim, um modo prático para capturar os pontos de junção que compartilhem características em comum. Sendo assim, a próxima tabela mostra a correspondência entre os símbolos curinga da linguagem Aspect$\mathcal{M}$ e AspectC++. Observe que o símbolo $*$ não pode ser utilizado como símbolo curinga em AspectC++, visto que já possui outro significado.

Aspect$\mathcal{M}$	**AspectC++**
*	%
...	...

Em Aspect$\mathcal{M}$, pode-se utilizar padrões de assinatura para especificar várias assinaturas contidas em uma especificação Machĭna. Os símbolos curingas e os operadores de conjunto podem ser utilizados para especificar um padrão de assinatura. A tabela abaixo apresenta a tradução de alguns padrões apresentados na Seção 4.2.

Aspect$\mathcal{M}$	**AspectC++**
Globals	Globals
Exp*	Exp%
not dynamic *.*	**!**(% %::dynamic_%(...))
derived Globals.*	% Globals::derived_%(...)
action Expression.* **and** **static** Expression.*	**void** Expression::%(...) **&&** % Expression::static_%(...)

Com o objetivo de possibilitar a categorização de um conjunto de ponto de junção, Aspect$\mathcal{M}$ possui um conjunto de designadores. Um designador de conjunto de junção é uma fórmula que especifica um conjunto de pontos de junção ao qual um adendo pode ser aplicado. Os designadores podem ser agrupados com base nos tipos de pontos de junção capturados, assim, pode-se citar: execução de ações, acesso e inicialização de funções, transição entre estados, baseados no contexto e na estrutura dos módulos.

Cabe ressaltar que, para simplificar o entendimento das traduções, optou-se por utilizar todos os exemplos apresentados no Capítulo 4, seguidos de sua respectiva tradução, durante o decorrer desta seção.

Execução de Ações

Todas as ações de um módulo são traduzidas para métodos, com tipo de retorno void, da classe que representa o módulo. Desta forma, para capturar a execução de uma ação, basta capturar a execução do método correspondente. A Listagem 5.28 apresenta o conjunto de junção operationsActionExecute em Aspect$\mathcal{M}$, enquanto que a Listagem

5.29 ilustra a tradução deste conjunto de junção para AspectC++. Analisando o código das listagens, percebe-se que a tradução é, praticamente, direta.

```
1 aspect:
2   public pointcut operationsActionExecute :
3     execution(action Operations.treatAdd) or
4     execution(action Operations.treatNot)
```

Listagem 5.28: Designador para execução de ações em AspectM

```
1 pointcut operationsActionExecute() =
2   execution("void Operations::treatAdd()") ||
3   execution("void Operations::treatNot()");
```

Listagem 5.29: Tradução do designador **execution** para AspectC++

Acesso a Funções

Basicamente, todas as funções (dinâmicas, estáticas, derivadas e externas), quando traduzidas, possuem um método para acessar seus valores (vide Seção 5.2.2). Desta forma, para capturar o acesso às funções, basta capturar a chamada ao método que realiza o acesso. Além disso, para diferenciar as funções depois de traduzidas, os métodos acessores são inicializados com o tipo da função. Outra observação importante é que um método acessor sempre possui um retorno, diferente de um método que representa uma ação. A Listagem 5.30 ilustra a declaração de um conjunto de junção, a saber `globalFunctionAcess`, enquanto que a Listagem 5.31 ilustra a sua respectiva tradução.

```
1 aspect:
2   pointcut globalFunctionAcess :
3     get (dynamic Globals.program)
```

Listagem 5.30: Designador para acessar funções em AspectM

```
1 pointcut globalFunctionAcess() =
2   call("% Globals::dynamic_program()");
```

Listagem 5.31: Tradução do designador **get** para AspectC++

Inicialização de funções

A inicialização de funções é realizada no bloco initial state de um módulo. Assim, para possibilitar a captura dos pontos de junção no bloco initial state basta capturar a chamada do método correspondente. Como ilustrado anteriormente, este bloco é traduzido para um método de assinatura `void nome_module::executeInitialState()`. Sendo assim, a Listagem 5.32 apresenta o conjunto de junção operationsInitialize, que representa todos os pontos de junção que ocorrem no bloco initial state do módulo Globals ou dos módulos cujo nome inicia-se com a palavra Ope, e a Listagem 5.33 apresenta sua tradução para AspectC++.

```
1  aspect:
2    public pointcut operationsInitialize :
3      initialization(Globals) or
4      initialization(Ope*)
```

Listagem 5.32: Exemplo do designador para capturar a inicialização de funções

```
1  pointcut operationsInitialize() =
2    call("void Globals::executeInitialState()") ||
3    call("void Ope%::executeInitialState()");
```

Listagem 5.33: Tradução do designador **initialization** para AspectC++

Transição entre estados

A tradução do designador **transition** é semelhante a tradução do designador **initialization**. Primeiramente, as regras que causam a transição de estado ficam no bloco transition de um módulo. Desta forma, para possibilitar a captura dos pontos de junção no bloco transition, basta capturar a chamada do método correspondente, que, como já ilustrado, este bloco é traduzido para um método de assinatura `void nome_module::executeTransitionRule()`. Como exemplo de tradução, a Listagem 5.34 apresenta o conjunto de junção intercalateEverything e a Listagem 5.35 sua respectiva tradução.

```
1  aspect:
2    public pointcut intercalateEverything :
3      transition(*)
```

Listagem 5.34: Exemplo do designador para capturar as transições dos estados

```
1  pointcut intercalateEverything() =
2    call("void %::executeTransitionRule()")
```

Listagem 5.35: Tradução do designador **transition** para AspectC++

Designadores Baseados no Contexto

Os designadores **target** e **args** são traduzidos diretamente para construções em AspectC++. O designador **target** possibilita capturar pontos de junção que ocorrem durante a execução de uma ação cujo agente receptor é do tipo do módulo passado como parâmetro. Toda chamada de abstração é encapsulada em um objeto da classe Klar__Abstraction. Assim, o superagente a partir deste objeto realiza a chamada utilizando o agente receptor armazenado para invocar a respectiva ação. Para capturar este agente receptor em AspectC++, basta utilizar a construção `that`. Desta forma, na Listagem 5.36, define-se o conjunto de junção useTarget e na Listagem 5.37 sua tradução para AspectC++.

```
1  aspect:
2    public pointcut useTarget :
3      target("Operations") and
4      execution(action *.*)
```

Listagem 5.36: Exemplo do designador que utiliza o agente alvo

```
1  pointcut useTarget() =
2    that("Operations") ||
3    execution("void %::%()");
```

Listagem 5.37: Tradução do designador **target** para AspectC++

O designador **args** é utilizado para definir pontos de junção que utilizam os valores dos argumentos de uma função ou chamada de ação. A construção correspondente em AspectC++ também é denominada **args**. Assim, a Listagem 5.38 define o conjunto de junção argBinding e a Listagem 5.39 sua respectiva tradução.

Convém destacar que ambos designadores podem ser utilizados para restringir pontos por meio do tipo, como o exemplo de **target**, ou para associar valores, como o exemplo de **args**. Assim, **target** poderia ser utilizado para associar o agente alvo a uma variável, por exemplo.

Designadores Baseados na Estrutura dos Módulos

Em AspectC++ o designador **within** é utilizado para filtrar pontos tanto em classes quanto em métodos. Desta forma, ambos os designadores de Machĭna, a saber **within**

```
1  aspect:
2    public pointcut argBinding(id:String):
3      execution(dynamic Globals.memory(String)) and
4      args(id)
```

Listagem 5.38: Exemplo do designador que utiliza os valores dos argumentos

```
1  pointcut argBinding(Klar__String *id) =
2    execution("% Globals::dynamic_memory(Klar__String *)") &&
3    args(id);
```

Listagem 5.39: Tradução do designador **args** para AspectC++

e **withincode**, são traduzidos para o designador **within** de AspectC++. Sendo assim, a Listagem 5.40 define os conjuntos de junção moduleScope e actionScope, enquanto que a Listagem 5.41 apresenta sua tradução para AspectC++.

```
1  aspect:
2    public pointcut moduleScope :
3      get(dynamic Globals.program) and
4      within(Operations)
5
6    public pointcut actionScope :
7      execution(action *.treatOutput)  and
8      withincode(action Operations.treatValue)
```

Listagem 5.40: Exemplo dos designadores que utilizam a estrutura dos módulos

```
1  pointcut moduleScope() =
2    call("% Globals::dynamic_program()") &&
3    within("Operations");
4
5  pointcut actionScope() =
6    execution("void %::treatOutput()") &&
7    within("void Operations::treatValue()");
```

Listagem 5.41: Tradução dos designadoress **within** e **withincode** para AspectC++

5.3.1.2 Adendo

Conforme descrito no Capítulo 4, a definição de um adendo pode ser dividida em três partes: (i) a declaração do adendo; (ii) a especificação do conjunto de junção; (iii) e a declaração da regra de transição.

A declaração do adendo é responsável por definir o tipo do adendo. Em Aspect$\mathcal{M}$, existem quatro tipos de adendo: (i) anterior, (ii) posterior, (iii) de adição e (iv) de contorno. Os adendos anterior e posterior são traduzidos para as construções **before** e

after de AspectC++. Além disso, no fim da execução do corpo do adendo, é executada a lista de atualizações, causando a mudança de estado. Por exemplo, a transição de um estado S_n para S_{n+1} é possível ser interceptada, e assim, executar algumas regras antes do estado S_{n+1}. Desta forma, seria criado um estado intermediário, como $S_{n'}$. Já os adendos de adição e contorno são traduzidos para a construção **before** de AspectC++. No entando, como não é executada a lista de atualizações, o ponto interceptado permanece no mesmo estado, no caso do exemplo anterior, S_n.

A tradução dos conjuntos de junção já foi explicada na seção anterior. Agora, para declarar a regra de transição, primeiramente, é realizada uma inversão de controle do aspecto para o módulo. A Listagem 5.42 apresenta este código. Basicamente, é criada uma instância do módulo que possui o código do aspecto e, em seguida, é chamado o método que corresponde ao adendo. A linha 2 é utilizada apenas quando o adendo é do tipo de contorno para possibilitar a invocação de `proceed`.

```
1  ModuleName *ioc = new ModuleName();
2  tjp->actions();
3  ioc->advice_N(tjp,param_1,...,param_N);
4  delete ioc;
```

Listagem 5.42: Inversão de controle do aspecto para o módulo

Todo adendo declarado na seção **aspect** possui um método na classe correspondente ao módulo de sua declaração. Para diferenciar os métodos correspondentes aos adendos, esses são sufixados com um número seqüencial. A Listagem 5.43 apresenta a assinatura de um método correspondente a um adendo. Como o método utilizada *templates*, sua declaração deve ficar no `.h` de sua respectiva classe. Além disso, as linhas 2 e 3 não são geradas para adendos dos tipos de adição e contorno.

```
1 template <class TJP> void advice_N(TJP tjp, param_1,...,param_N){
2   executeAgentRules();
3   executeUpdateList();
4 }
```

Listagem 5.43: Assinatura de um método correspondente a um adendo

Para ilustrar a tradução completa de um adendo, será utilizada a declaração da Listagem 5.44. A Listagem 5.45 apresenta o código em AspectC++ responsável por interceptar o ponto correto e transferir o controle para o módulo `Exemplo` e, finalmente, a Listagem 5.46 apresenta o método utilizado. Por questões de simplicidade, optou-se por omitir a tradução das regras.

```
module Exemplo
  ...
  aspect:
    pointcut typeCheckAdd(op:Operations):
      execution(action Operations.treatAdd)
      target(op)
    around(op:Operations): typeCheckAdd(op) do
      let x = head(op.program);
      let y = head(head(op.program));
      if x is Int and y is Int then
        proceed
      else op.error := true
      end
    end
end Exemplo
```

Listagem 5.44: Conjunto de junção utilizado na definição do adendo da Listagem 4.16

```
//alguns includes
aspect ExemploAspect {

  public:

    pointcut typeCheckAdd(Operations *op) =
      execution("void Operations::treatAdd()") && target(op);

    advice typeCheckAdd(op) : around(Operations *op) {
      Exemplo *ioc = new Exemplo();
      tjp->actions();
      ioc->advice_N(tjp,op);
      delete ioc;
    }

};
```

Listagem 5.45: Código em AspectC++ responsável por interceptar um ponto e transferir o controle para um módulo

5.3.2 Transversalidade Estática

Diferente da transversalidade dinâmica, o processo de combinação da transversalidade estática foi todo implementado, com o objetivo de antecipar para a etapa de compilação alguns erros, como a introdução de funções com nomes conflitantes. Além disso, o próprio combinador de AspectC++, a saber ag++, apresentou alguns erros na introdução de métodos, durante a sua utilização no decorrer deste trabalho. Assim, aumentando a motivação para a implementação desta parte.

Basicamente, na compilação de uma especificação Machĭna, o usuário apenas indica o nome do arquivo que contém o módulo de definição Machĭna. Então, a partir deste arquivo, são geradas todas as classes e aspectos necessários.

```
1  //Alguns includes
2  class ExemploAspect {
3
4     //Some attributes and methods
5
6    template <class TJP> void advice_1(TJP tjp, Operations *op){
7       //tradução para a regra if
8
9       //exemplo da tradução da regra proceed
10     tjp->proceed();
11   }
12
13 };
```

Listagem 5.46: Método correspondente ao corpo de um adendo

Durante a etapa de compilação, como os aspectos possuem a propriedade de transparência (*obliviousness*), ou seja, os módulos não precisam ser especificamente preparados para receber as melhorias proporcionadas pelos aspectos, não se sabe antecipadamente todos os módulos que serão envolvidos no processo de combinação. Desta forma, similar ao combinador de AspectC++, todos os módulos existentes no diretório corrente são lidos e a sua respectiva AST é armazenada na memória.

Assim, para implementar a transversalidade estática, percorre-se cada AST, identificando módulos que possuem uma seção **aspect** com declarações de inter-tipos, ou seja, introdução de funções, tipos ou ações. Uma vez encontrados, os nodos referentes à declaração de inter-tipos contêm informações necessários para criar um novo nodo, semelhante a uma declaração normal, e em qual AST este novo nodo deve ser inserido. Desta forma, um novo nodo é criado e inserido na AST determinada pela sua declaração. Após percorrer as ASTs, inicia-se a compilação a partir do módulo de definição Machĭna.

```
1  module Aspecto
2   ...
3   ...
4   aspect:
5     Comum.funcoes_tipos_N+1
6     ..
7     OutroModulo.funcoes_tipos_N+N
8
9
10    Comum.acao_N+1
11    ..
12    OutroModulo.acao_N+N
13  ...
14  ...
15 end
```

Listagem 5.47: Módulo Aspecto.

```
1  module Comum
2   ...
3   ...
4   algebra:
5     funcoes_tipos_1;
6     ..
7     funcoes_tipos_1;
8
9   abstractions:
10    acao_1;
11    ..
12    acao_2;
13  ...
14  ...
15 end
```

Listagem 5.48: Módulo Comum.

As Listagens 5.47 e 5.48, e a Figura 5.3 ilustram este processo supra mencionado. Inicialmente, é identificado o módulo que contém uma seção **aspect**, Listagem 5.47. Em seguida, analisando os nodos referentes à declaração de inter-tipos, obtém-se o nome do módulo que receberá as novas declarações. Assim, após ter recuperado a AST referente ao módulo, Figura 5.3, os novos nodos são inseridos.

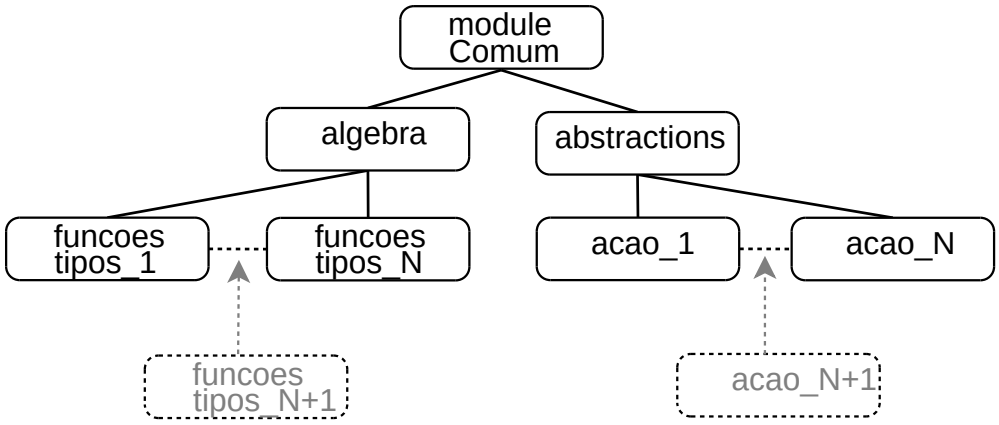

Figura 5.3: Introdução de tipos, funções e ações na AST de um módulo

5.4 Considerações Finais

Neste capítulo, foram apresentadas três questões principias: os arcabouços utilizados no desenvolvimento do compilador para a nova versão da linguagem Machĭna, como

as construções da linguagem Machĭna são compiladas para código C++ e como as novas construções são compiladas para código AspectC++. A primeira informação é importante caso se deseje alterar a gramática da linguagem. A segunda e a terceira questão têm duas funções básicas. Primeiramente, mostrar como as construções da linguagem Machĭna são convertidas em C++ e AspectC++ ajuda a tornar mais clara a semântica da linguagem. Em segundo lugar, o registro destas traduções constitui um manual básico a ser seguido em caso de se desejar alterar a forma como as traduções são realizadas.

No desenvolvimento do compilador, a idéia inicial era realizar a integração dos arcabouços ACOA e kℓar, conforme apresentado na Seção 5.1.3, e estendê-los para compilarem as novas construções propostas. Esta primeira idéia foi descartada com o objetivo de simplificar o processo de tradução, removendo-se a linguagem intermediária gerada, a saber, MIR.

A segunda abordagem era modificar o passo de compilação responsável pela geração de código do antigo compilador de Machĭna. Assim, em vez de ser gerado código MIR, este passo armazenaria os dados em classes providas pelo kℓar para que posteriormente fosse gerado código C++. Devido a diversos conflitos entre nomes de classes, advindos da falta de utilização de *namespaces* pelos arcabouços, esta idéia não pôde ser implementada.

A terceira, e atual abordagem utilizada, é a utilização do arquivo de entrada para o ACOA e algumas classes do antigo compilador de Machĭna. Deste modo, as classes referentes ao processo de tradução para C++, realizado pelo kℓar, não foram utilizadas. No entanto, algumas classes que o kℓar utiliza para possibilitar a execução do código C++ foram utilizadas, como exemplo, têm-se as classes que representam os tipos da linguagem. Assim, a utilização do ACOA bem como algumas classes do antigo compilador e do kℓar agilizaram o desenvolvimento do compilador para a nova versão, possibilitando concentrar maiores esforços na definição e implementação das novas construções para a linguagem Machĭna.

A estratégia inicial era implementar apenas alguns elementos da linguagem Machĭna, possibilitando testar a implementação das novas construções propostas. No entanto, foi possível implementar, praticamente, todas construções da linguagem. A atual versão do compilador apenas não provê suporte para os tipos genéricos, como exemplo, tem-se `type Stack(T):tuple(a:T,b:Int)`. A principal dificuldade encontrada para esta implementação, como foi deixanda para o final, são os problemas encontrados no costurador de AspectC++. Conforme mencionado na Seção 5.1.1, o costurador de AspectC++ apresenta alguns problemas na inserção estática de métodos. Assim, ao acrescentar alguns códigos referentes ao tipo genérico, acontecem alguns problemas de costura de código. Desta forma, para incorporar este tipo seria melhor aguardar a

liberação da nova versão do costurador de AspectC++, a saber, Versão 1.0pre4-1.0.

Capítulo 6

Avaliação e Validação dos Resultados

Para fins de avaliação e validação da linguagem proposta, apresenta-se neste capítulo uma pequena avaliação conceitual, utilizando alguns arcabouços apresentados no Capítulo 3. Além disso, com o objetivo de ilustrar a modularização e a clareza da especificação obtida, especificou-se uma linguagem de programação imperativa, a saber *Tiny*, e um sistema para controle de uma caldeira a vapor. Por fim, são apresentados mais alguns exemplos que procuram validar algumas construções.

6.1 Avaliação Conceitual da Linguagem

Segundo Chavez [CL03], o principal benefício de se ter um arcabouço conceitual como o apresentado na Seção 2.3.2 é fornecer suporte para a avaliação das abordagens existentes, bem como para o desenvolvimento de novos métodos e ferramentas com base em uma terminologia unificada e consistente. A Tabela 6.1 resume os conceitos que constituem o arcabouço.

Além de definir o arcabouço, duas propriedades, quantificação e transparência (*obliviousness*), foram propostas como propriedades necessárias para a POA. Porém, além destas duas propriedades, Chavez considerou uma nova propriedade a qual foi chamada de *dicotomia aspecto-base*.

A quantificação é definida como a capacidade de escrever declarações unitárias e separadas que melhoram muitos lugares não-locais no sistema. A transparência é a idéia de que os componentes não precisam ser especificamente preparados para receber as melhorias proporcionadas pelos aspectos. Por fim, a *dicotomia aspecto-base* significa a adoção de uma distinção clara entre componentes e aspectos.

A Tabela 6.2 apresenta um resumo das interpretações utilizando o arcabouço a-

Modelo	**Resumo**
Componentes	componentes, mecanismo de composição, regras de composição, linguagem de componente
Pontos de Junção	ponto de junção, ponto de junção estático, ponto de junção dinâmico, contexto estático, contexto dinâmico
Principal	aspecto, transversalidade estática, transversalidade dinâmica, interface transversal
Processo de Combinação	combinador de aspectos, processo de combinação, processo de combinação estático, processo de combinação dinâmico

Tabela 6.1: Conceitos do arcabouço apresentado

	AspectJ	Eos	**Aspect$\mathcal{M}$**
Modelo de Componentes	modelo de objetos	modelo de objetos	modelo ASM
Linguagem de Componentes	Java	C#	Machĭna
Aspecto	aspecto	classpects	módulo
Transversalidade Dinâmica	pontos na execução	pontos na execução	pontos na execução
Transversalidade Estática	classe	classpects	módulo
Interface transversal	conjunto de junção	conjunto de junção	conjunto de junção
Modelo de combinação	estático	estático	estático
Quantificação	sim	sim	sim
Transparência	sim	sim	sim
Dicotomia aspecto-base	sim	não	não

Tabela 6.2: Interpretações utilizando o arcabouço apresentado

presentado. Esta tabela apresenta as linguagens AspectJ, Eos e Aspect$\mathcal{M}$. AspectJ e Eos, como mencionado no Capítulo 2, influenciaram diretamente o projeto da linguagem Aspect$\mathcal{M}$. Por exemplo, o modelo de ponto de junção utilizado é similar ao da linguagem AspectJ e a linguagem proposta utiliza o conceito de *classpect*, unificando assim, no contexto da linguagem Machĭna, módulos e aspectos. Cabe ressaltar que apesar de utilizar a idéia *classpect*, na tabela, manteve-se a terminologia padrão, a saber, módulo.

Das linguagens apresentadas na tabela, somente AspectJ, segundo o arcabouço apresentado, oferece suporte à POA. A *dicotomia aspecto-base* é colocada com uma propriedade fundamental e necessária para à POA. Desta forma, Eos, e conseqüentemente, Aspect$\mathcal{M}$ não oferecem suporte à POA. Outra linguagem que não oferece suporte à POA, de acordo com este arcabouço, é Hyper/J.

Agora, como o arcabouço proposto Masuhara e Kiczales [MK03] trata componentes e aspectos uniformemente, assumindo que não há linguagem de componentes primária nem *dicotomia aspecto-base*, as linguagens, Eos, Aspect$\mathcal{M}$ e Hyper/J, podem ser classificadas com linguagens orientadas por aspectos.

A principal motivação em utilizar o arcabouço de Chavez foram as diretrizes destacadas em [CL03]. O arcabouço agrega os conceitos e algumas propriedades essenciais que dão suporte ao projeto de linguagens orientadas por aspectos. Basicamente, segundo Chavez, o projetista de uma nova linguagem deve:

- adotar um modelo de componentes;
- adotar um modelo de pontos de combinação adequado;
- adotar um modelo de processo de combinação;
- oferecer representações adequadas para os elementos dos modelos adotados;
- fornecer uma semântica clara para transversalidade;
- fornecer algum meio de oferecer suporte à quantificação.

6.2 Descrição dos Exemplos

Nesta seção, os exemplos descritos na Seção 3.3 são mais bem detalhados. Além disto, esta seção tem por objetivo demonstrar que a nova versão da linguagem Machĭna não possui os problemas apresentados, ou seja, entrelaçamento e espalhamento de regras.

6.2.1 Linguagem *Tiny*

6.2.1.1 Descrição Informal de *Tiny*

A descrição informal da linguagem *Tiny* realizada nesta seção foi baseada em [TMDB99]. *Tiny* é uma linguagem de programação imperativa de pequeno porte que possui dois tipos de construções: expressões e comandos. Ambos podem conter identificadores, que são cadeias de caracteres contendo letras e dígitos, começando com uma letra, e números inteiros e os valores lógicos **true** e **false**. Utilizaremos meta-variáveis

$I, I_1, I_2, I', ...$ para identificadores, $E, E_1, E_2, E', ...$ para expressões e $C, C_1, C_2, C', ...$ para comandos.

A sintaxe abstrata de *Tiny* é:

$$\begin{array}{lll} \text{E} & ::= & 0 \mid 1 \mid true \mid false \mid read \mid I \mid not\ E \mid E_1 = E_2 \mid E_1 + E_2 \mid (E) \\ \text{C} & ::= & I := E \mid output\ E \mid if\ E\ then\ C_1\ else\ C_2 \mid while\ E\ do\ C \mid C_1; C_2 \mid (C) \end{array}$$

Cabe ressaltar que a gramática dada é ambígua, mas isto não constitui problema, porque, nesta especificação, questões relativas a análise sintática não são tratadas. Desta forma, a semântica é baseada em sintaxe abstrata.

Todo comando de *Tiny*, quando executado, modifica um estado. Este estado contém três elementos:

- a `memória` – correspondência entre identificadores e valores. Na memória, cada identificador está associado a algum valor ou está não-associado (*unbound*);
- a `entrada` – fornecida pelo usuário antes do início da execução do programa; consiste em uma seqüência (possivelmente vazia) de valores que podem ser lidos utilizando uma expressão;
- a `saída` – é inicialmente uma seqüência vazia de valores que armazena os resultados de um comando de saída, a saber, *output E*.

O valor das expressões e o efeito dos comandos são apresentados a seguir:

0 ou 1 - o valor de 0 é 0 e o valor de 1 é 1;

true ou false - o valor de *true* é o valor lógico *true* e o valor de $false$ é o valor lógico $false$;

read - o valor de *read* é o valor do próximo elemento na entrada (ocorre um erro se a entrada for vazia); a expressão *read* possui o efeito colateral de remover o primeiro item da entrada, de modo que, após a avaliação da expressão, a entrada possui um elemento a menos;

I - o valor de uma expressão I é o valor associado a I na memória (ocorre um erro se I está não-associado);

not E - se o valor de E for *true* então o valor de *not E* será $false$; se o valor de E for $false$ então o valor de *not E* será *true*; em todos os outros casos, ocorre um erro;

E_1 = E_2 - o valor de $E_1 = E_2$ é *true* se os valores de E_1 e E_2 forem iguais; caso contrário, a expressão avalia em $false$; se E_1 ou E_2 não forem números inteiros, ocorre um erro;

E_1 + E_2 - o valor de $E_1 + E_2$ é a soma numérica entre os valores de E_1 e E_2; se algum destes valores não for um número, ocorre um erro.

I := E - o identificador I é associado, na memória, ao valor da expressão E, sobrescrevendo o que estava anteriormente associado a I;

output E - o valor de E é colocado na saída;

if E then C_1 else C_2 - se o valor de E for *true*, C_1 é executado; se o valor de E for $false$, C_2 é executado; em todos os outros casos, ocorre um erro;

while E do C - se o valor de E for *true*, então C é executado e, em seguida, *while E do C* é executado novamente no estado resultante da execução de C; se o valor de E for $false$, então nada é feito; se o valor de E não for nem *true* nem $false$, ocorre um erro;

C_1;C_2 - os comandos C_1 e C_2 são executados nesta ordem; C_2 é executado no estado resultante da execução de C_1

Observe nessa descrição informal o número de situações que podem ocorrer erros. Sem os meios apropriados, o tratamento destes erros ficaria entrelaçado e espalhado em toda especificação.

Um Programa em *Tiny*

O exemplo da Listagem 6.1 escreve na saída a soma dos números contidos na entrada. O fim da entrada é marcado com 0.

```
1 sum := 0;
2 x := read;
3 while not (x = 0) do (
4   sum := sum + x;
5   x := read
6 );
7 output sum
```

Listagem 6.1: Programa escrito em *Tiny*

6.2.1.2 Descrição Formal de *Tiny* em Aspect$\mathcal{M}$

Nesta seção mostraremos uma definição formal de *Tiny* utilizando a nova versão da linguagem Machĭna. Baseada na metodologia de definição de ASM, a linguagem *Tiny* pode ser descrita por $Machĭna_{tiny} = (\Upsilon, \mathcal{A}, S_0, \mathcal{P}_{tiny})$.

6.2.1.3 Convenção

A árvore sintática abstrata de um programa em *Tiny* será armazenada em uma lista. As regras serão definidas de forma a consultar sempre o primeiro elemento desta lista para decidir o que será feito. Desta forma, os comandos e expressões são traduzidos conforme apresentado nas Tabelas 6.3 e 6.4.

Comandos	Representação
$I := E$	[ASSIGN, I,E]
output E	[OUTPUT,E]
$if\ E\ then\ C_1\ else\ C_2$	[IF,E, C_1, C_2]
$while\ E\ do\ C$	[WHILE,E, C]
$C_1; C_2$	$[C_1, C_2]$

Tabela 6.3: Tradução de comandos de *Tiny* para a forma de lista

Expressões	Representação
0	[0]
1	[1]
$true$	[TRUE]
$false$	[FALSE]
$read$	[READ]
I	$[I]$
not E	[NOT, E]
$E_1 = E_2$	[EQUALS,E_1,E_2]
$E_1 + E_2$	[ADD,E_1,E_2]

Tabela 6.4: Tradução de expressões de *Tiny* para a forma de lista

6.2.1.4 O Vocabulário Υ

O vocabulário de $Machĭna_{tiny}$, o qual é definido na seção **algebra**, contém os seguintes nomes de função:

- *program* : função de aridade 0 que contém um programa em Tiny;;

- *memory* : função de aridade 1 que associa nomes de identificadores a valores;
- $infile$: função de aridade 0 que contém a seqüência de valores de entrada;
- *output* : função de aridade 0 que contém a seqüência de valores de saída do programa;
- *opstack* : função de aridade 0 que simula uma pilha auxiliar de execução.

6.2.1.5 O Estado Inicial S_0

No estado inicial da especificação $Machĭna_{tiny}$, seção **initial state**, definimos interpretações iniciais de alguns nomes de função pertencentes a Υ. Na $Machĭna_{tiny}$, definiremos as seguintes interpretações:

- o nome de função *program* é interpretado, inicialmente, como o programa que calcula a soma dos valores de entrada;
- *memory*, $Val_{S_0}(memory) = \lambda x.undef$, simbolizando que, inicialmente, nenhum identificador está associado a valores;
- $infile$ é interpretada como uma lista de valores; o objetivo é possibilitar uma simulação do programa carregado;
- *output* e *opstack* são interpretados inicialmente como a lista vazia.

6.2.1.6 A Regra da Especificação $\mathcal{P}_{tiny}$

O principal objetivo desta seção é destacar a modularização obtida com a utilização da nova versão da linguagem Machĭna. Deste modo, a maioria das regras foi omitida para focar a explicação em alguns pontos. No entanto, todas as regras da especificação $\mathcal{P}_{tiny}$ são mostradas no Apêndice B.

Inicialmente, estas regras são definidas em seis módulos, a saber:

1. Módulo principal (`MainProgram`) – contém a regra principal da especificação e as rotinas de inicialização;
2. Módulo de Expressões (`Expressions`) – contém as regras de avaliação de expressões;
3. Módulo de Comandos (`Commands`) – contém as regras de execução de comandos;
4. Módulo de Operações (`Operations`) – contém as regras de operação na pilha;

5. Módulo de Globais (`Globals`) – contém as declarações dos principais tipos e funções da especificação de *Tiny*;

6. Módulo de Tratamento de Erro (`TypeCheck`) – contém a verificação de tipos e alguns tratamentos de erros.

Basicamente, para ilustrar a modularização obtida na especificação, utilizando as novas construções, serão apresentados alguns trechos da especificação de *Tiny*, a saber parte do módulo de operações e expressões. Desta forma, a Listagem 6.2 mostra um trecho da especificação do módulo para o tratamento de operações. Esta listagem utiliza a versão antiga de Machĭna. Observe que o código responsável pelo tratamento de erros fica entrelaçado e espalhado na especificação, dificultando a sua compreensão e entendimento.

```
1  abstractions:
2    public action treatAdd is
3      let x = head(program);
4      let y = head(tail(program));
5      if (x is Int) and (y is Int) then
6        program := (Int(x) + Int(y))::tail(tail(program));
7        opstack := tail(opstack)
8      else
9        error := true
10     end
11   end treatAdd
12   public action treatEquals is
13     let x = head(program);
14     let y = head(tail(program));
15     if (x is Int) and (y is Int) then
16       program := (Int(x) = Int(y))::tail(tail(program));
17       opstack := tail(opstack)
18     else
19       error := true;
20     end
21   end treatEquals
```

Listagem 6.2: Entrelaçamento de regras em um trecho do módulo para o tratamento de operações

A Listagem 6.3 apresenta o mesmo código sem o tratamento de erros. Pode-se notar que a especificação ficou mais coesa, tratando apenas de um interesse, o que facilita o seu entendimento.

Agora, utilizando as novas construções da linguagem Machĭna, pode-se incluir o tratamento de erros sem causar o entrelaçamento do código. A Listagem 6.4 ilustra a inclusão do tratamento de erros na Listagem 6.3.

De forma semelhante, a Listagem 6.5 apresenta outro trecho de uma especificação utilizando a versão antiga da linguagem Machĭna. Esta listagem mostra uma parte do

```
1  abstractions:
2    public action treatAdd is
3      let x = head(program);
4      let y = head(tail(program));
5      program := (Int(x) + Int(y))::tail(tail(program));
6      opstack := tail(opstack)
7    end treatAdd
8    public action treatEquals is
9      let x = head(program);
10     let y = head(tail(program));
11     program := (Int(x) = Int(y))::tail(tail(program));
12     opstack := tail(opstack)
13   end treatEquals
```

Listagem 6.3: Trecho do módulo para o tratamento de operações sem entrelaçamento de regras

```
1  aspect:
2    pointcut executeAdd : execution(action Operations.treatAdd);
3    pointcut executeEquals : execution(action Operations.treatEquals);
4    around(op:Operations):(executeAdd or executeEquals) and target(op)do
5      let x = head(op.program);
6      let y = head(tail(op.program));
7      if (x is Int) and (y is Int) then
8        proceed
9      else
10       op.error := true
11     end
12   end
```

Listagem 6.4: Aspecto para incluir o tratamento de erros

módulo para o tratamento de expressões. Note que, como dito anteriormente, o código responsável pelo tratamento de erros fica entrelaçado e espalhado na especificação, diminuindo a coesão do módulo e, consequentemente, aumentando o seu acoplamento.

A Listagem 6.6 mostra o mesmo código sem o tratamento de erros. Novamente, deixando claro que quando o código trata de apenas um requisito, ou seja, está bem coeso, além de facilitar a sua compressão, aumenta a sua reusabilidade, uma vez que possui um baixo acoplamento.

Por fim, a Listagem 6.7 mostra a inclusão do tratamento de erros na Listagem 6.6, utilizando as novas construções da linguagem Machĭna.

6.2.2 Caldeira a Vapor

O problema da caldeira a vapor abordado nesta seção foi descrito por Jean-Raymond Abrial [Abr94], como sugestão de trabalho para os participantes do *Dagstul Meeting Methods for Semantics and Specification*, realizado em Junho de 1995. Assim, Abrial

```
1  abstractions:
2    public action readExpMemory is
3      let id = head(ex.program);
4      if (id is String) then
5        if memory(String(id)) is Globals.Undefined then
6          error := true
7        else
8          program := memory(id) :: tail(program)
9        end
10     else
11       error := true
12     end
13   end
14   public action treatExpRead is
15     if infile = nil then
16       error := true
17     else
18       program := head(infile)::tail(program);
19       infile := tail(infile)
20     end
21   end
```

Listagem 6.5: Entrelaçamento de regras em um trecho do módulo para o tratamento de expressões

```
1  abstractions:
2    public action readExpMemory is
3      let id = String (head(program));
4      program := memory(id) :: tail(program)
5    end
6    public action treatExpRead is
7      program := head(infile)::tail(program);
8      infile := tail(infile)
9    end
```

Listagem 6.6: Trecho do módulo para o tratamento de expressões sem entrelaçamento de regras

especificou um programa para controlar o funcionamento de uma cadeira a vapor. Os participantes do evento deveriam apresentar uma definição em alto nível e, caso aplicado, o refinamento necessário para obtenção do código executável.

Segundo Abrial, a caldeira deve trabalhar com o nível de água dentro de limites considerados normais de funcionamento e nunca ultrapassar os níveis de risco definidos. Assim, é importante que o sistema trabalhe corretamente porque a quantidade de água presente, quando a caldeira estiver funcionando, deve sempre estar entre esses limites; caso contrário, a caldeira ficará danificada.

Inicialmente, Abrial descreve o ambiente físico no qual o programa deve interagir. Cada unidade física é apresentada, assim como suas restrições e condições de fun-

```
1  aspect:
2    pointcut executeMemory:execution(action Expressions.readExpMemory);
3    pointcut executeRead:execution(action Expressions.treatExpRead);
4    around(ex:Expressions): executeMemory and target(ex) do
5      let id = head(ex.program);
6      if (id is String) then
7        if (ex.memory(String(id)) is Globals.Undefined) then
8          ex.error := true
9        else
10         proceed
11       end
12     else
13       ex.error := true
14     end
15   end
16   around(ex:Expressions): executeRead and target(ex) do
17     if ex.infile = nil then
18       ex.error := true
19     else
20       proceed
21     end
22   end
```

Listagem 6.7: Aspecto para incluir o tratamento de erros

cionamento. Além disto, são definidas algumas constantes e variáveis que devem ser verificadas e controladas ao longo do funcionamento do sistema.

Logo após, é apresentado como o programa deve se comunicar com as unidades físicas e quais são as fases de comunicação. Segundo Abrial, o programa comunica-se com os dispositivos físicos através de mensagens, sendo que o tempo para transmissão pode ser negligenciado. O programa segue um ciclo que, em princípio, não termina. Este ciclo acontece a cada cinco segundos e consiste nas seguintes ações: (a) receber as mensagens dos dispositivos físicos, (b) analisar a informação recebida e (c) transmitir uma mensagem aos dispositivos. Por fim, são detalhados os modos de operação que o programa pode assumir e, finalmente, é descrito a detecção de erros nos equipamentos.

Em [ABL95], foram apresentadas 23 soluções abordando diferentes métodos de especificação. Uma delas foi proposta por Börger[BBD+96]. Neste trabalho foi apresentado um modelo em alto nível, a saber Modelo Básico, fundamentado na Álgebra Evolutiva. Como linguagem para o Modelo Básico foi utilizado o modelo ASM. Por fim, apresentou-se o refinamento do modelo até obtenção do código executável em C++.

Esta seção apresenta a implementação da caldeira a vapor, utilizando a nova versão da linguagem Machĭna. A especificação de Abrial permite, em alguns momentos, divergências nas interpretações. Desta forma, a implementação realizada nesta seção foi baseada na solução proposta por Börger. Um fato importante na solução de Börger é a descrição clara de alguns interesses transversais que serão abordados no decorrer da

seção.

6.2.2.1 Unidades Físicas

O sistema de controle da caldeira a vapor deve se comunicar com as seguintes unidades físicas:

- uma válvula;
- quatro bombas de água para a caldeira;
- quatro controladores de bomba (um para cada bomba);
- unidade medidora de água;
- unidade medidora de vapor;

A caldeira a vapor é constituída de uma válvula para liberação de água, que é utilizada para esvaziar a caldeira apenas na fase inicial. A capacidade total de água C é indicada em litros e M_1 e M_2 são os limites inferiores e superiores, respectivamente, também em litros. Se o nível de água estiver abaixo de M_1 com liberação de vapor sem o devido suprimento de água, ou se o nível estiver acima de M_2 com as bombas fornecendo água e pouca liberação de vapor, a caldeira pode estar em perigo. São estipulados os níveis inferior N_1 e superior N_2 para o funcionamento normal, que devem ser mantidos durante o funcionamento. A Figura 6.1 ilustra a caldeira com suas unidades físicas.

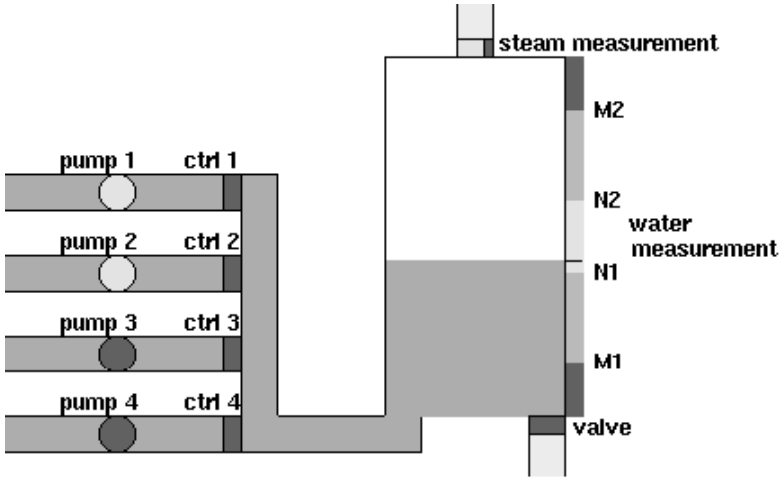

Figura 6.1: Caldeira a vapor com suas unidades físicas

Além disso, a quantidade máxima W de vapor saindo da caldeira é medida em litros/seg e o gradiente de aumento U_1 e diminuição U_2 de vapor é dado em litros/seg^2. A Tabela 6.5 resume as variáveis e constantes existentes no sistema.

	Unidade	Comentários
		Quantidade de água na caldeira
C	litros	Capacidade máxima
M_1	litros	Limite mínimo de risco
M_2	litros	Limite máxima de risco
N_1	litros	Limite mínimo normal
N_2	litros	Limite máxima normal
		Liberação de calor na caldeira
W	litros/seg	Quantidade máxima
U_1	litros/seg^2	Gradiente máximo de aumento
U_2	litros/seg^2	Gradiente máximo de diminuição
		Medidas correntes
$waterLevel$	litros/seg	Quantidade de água na caldeira
$steamLevel$	litros/seg	Quantida de de vapor dissipado

Tabela 6.5: Resumo de variáveis e constantes

A caldeira a vapor é formada por estas unidades físicas que possuem comunicação direta com o sistema de controle. Como estas unidades possuem algumas características em comum, essas foram agrupadas em um módulo denominado `PhysicalUnit`. Assim, toda unidade física deve incluir o módulo `PhysicalUnit`. A Listagem 6.8 apresenta a codificação deste módulo.

```
1  module PhysicalUnit
2    import SteamBoilerControl;
3    algebra:
4      dynamic failure: Bool;
5      dynamic steamBoiler: agent of SteamBoilerControl;
6      dynamic programReady: Bool;
7      dynamic ready: Bool := true;
8
9    abstractions:
10     public action setReady (in r:Bool) is
11       ready := r
12     end
13     public action setFailure (in f:Bool) is
14       failure := f
15     end
16     public action sendProgramReady is
17       programReady := true;
18     end
19 end PhysicalUnit
```

Listagem 6.8: Especificação do módulo com características comuns das unidades físicas

A unidade medidora de água (*water level measuring unit*) mede a quantidade $waterLevel$ em litros de água na caldeira em um determinado instante, assim como a

unidade medidora de vapor (*steam level measuring unit*) mede a quantidade *steamLevel* de vapor que sai da caldeira em litros/seg. Estas duas unidades podem ser modeladas de forma semelhante. Assim, ambas são representadas como agentes do módulo `MeasuringUnit`, diferenciando-se pela função `idMeasuring`. A codificação deste módulo é apresentada na Listagem 6.9.

```
1  module MeasuringUnit
2
3    include PhysicalUnit;
4
5    algebra:
6      dynamic idMeasuring: Int;
7      dynamic measure: Int;
8
9    abstractions:
10     public action setMeasure (in measureUnit:Int) is
11       measure := measureUnit
12     end
13     public action setIdMeasuring (in id:Int) is
14       idMeasuring := id
15     end
16
17   initial state:
18     steamBoiler := SteamBoilerControl.theAgent;
19
20   transition:
21     if programReady then
22       steamBoiler.recieveMeasuringInfo(self, ready, failure,
23                                        measure, idMeasuring);
24     end;
25
26 end MeasuringUnit
```

Listagem 6.9: Especificação do módulo para as unidades medidoras

A válvula da caldeira a vapor tem comportamento semelhante das unidades de medidas. Em vez de medir a quantidade de água ou vapor, a válvula apenas possui dois estados, fechado ou aperto, representado pela função `opened`. A Listagem 6.10 apresenta a codificação da válvula da caldeira.

As demais unidades físicas, a saber, bombas de água e controladores de bomba, são similares às apresentadas. Basicamente, uma unidade física é responsável por enviar uma informação. No caso da unidade medidora de água envia a quantidade de água atual da caldeira. Já as bombas de água informam se estão ligadas ou não e os controladores de bomba indicam se existe ou não circulação de água no sentido bomba-caldeira. Além disso, toda unidade física só começa a mandar suas informações após receber uma mensagem do sistema de controle informando que a inicialização está pronta.

```
1  module ValveUnit
2
3    include PhysicalUnit;
4
5    algebra:
6      opened: Bool := false;
7
8    abstractions:
9      public action setOpened (in openedValve:Bool) is
10       opened := openedValve
11     end
12
13   initial state:
14     steamBoiler := SteamBoilerControl.theAgent;
15
16   transition:
17     if programReady then
18       steamBoiler.recieveValveInfo(self, ready, failure, opened);
19     end;
20
21 end ValveUnit
```

Listagem 6.10: Especificação do módulo para a válvula

A Figura 6.2 ilustra a estrutura dos módulos para as unidades físicas. Nesta figura os módulos são representados por retângulos e as operações de *include* correspondem às setas.

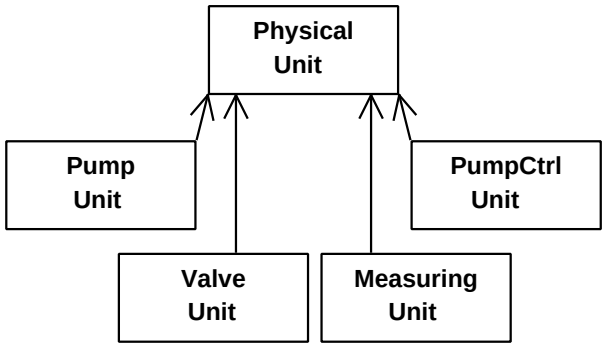

Figura 6.2: Estrutura dos módulos para as unidades físicas

6.2.2.2 Sistema de Controle

O sistema de controle da caldeira opera em cinco modos de operação: `inicialização`, `normal`, `degradado`, `recuperação` e `emergência`. Nesta especificação, optou-se por criar um módulo para cada modo de operação do sistema, com exceção do modo `emergência`, que apenas finaliza a execução do sistema. Assim, cada modo de operação

possui um agente de um tipo específico que fica responsável em executar as regras referentes a este modo. Como cada modo de operação possui características em comum, essas foram agrupadas em um módulo denominado `Common`. Além disso, o módulo `Common` inclui mais dois módulos, a saber `Constants`, que contém as constantes do sistema, e `SteamBoilerAgents`, que contém a declaração dos agentes.

No modo `inicialização`, o sistema de controle aguarda o sinal de que a caldeira a vapor está pronta, especificado pela função `steamBoilerWaiting`. Logo após, deve-se aguardar até que o nível de água esteja entre os valores permitidos. Feito isso, o programa envia constantemente uma mensagem para todas as unidades físicas indicando que está pronto, até que receba a confirmação das unidades. Se houver alguma falha nas unidades medidoras, o sistema entra no modo `emergência`; e caso a falha seja em outra unidade física, o sistema entra no modo `degradado`. Caso contrário, ele entra no modo `normal`. A Listagem 6.11 apresenta as regras do modo `inicialização`.

```
1  if initMode and steamBoilerWaiting and anyMeasuringFailure then
2    enterEmergencyStopMode;
3  end;
4  if initMode and steamBoilerWaiting and waterLevelAdjusted and
5                                  not physicalUnitsReady then
6    indicateProgramReady;
7  end;
8  if initMode and steamBoilerWaiting and physicalUnitsReady then
9    if waterLevelAdjusted and not anyMeasuringFailure then
10     if allPhysicalUnitsOK then enterNormalMode
11     else enterDegradedMode end
12   else enterEmergencyStopMode end
13 end;
```

Listagem 6.11: Regras para o modo inicialização do sistema de controle

O modo `normal` representa o funcionamento padrão da caldeira com todas as unidades físicas funcionando corretamente. Na presença de algum erro na unidade medidora de nível de água, o programa passa para o modo `recuperação`. Qualquer erro em outra unidade física leva o programa para o modo `degradado`. As regras do modo `normal` são mostradas na Listagem 6.12.

```
1  if normalMode and not allPhysicalUnitsOK then
2    if failure(waterMeasuring) then
3      enterRescueMode;
4    else
5      enterDegradedMode;
6    end
7  end;
```

Listagem 6.12: Regras para o modo normal do sistema de controle

No modo `degradado`, o programa tenta manter o nível normal de água mesmo na presença de falhas de algumas unidades físicas. Assume-se que a unidade medidora de nível de água esteja funcionando corretamente. Caso seja detectada uma falha nessa unidade, o programa passa para o modo `recuperação`. Uma vez que todas as unidades tenham sido recuperadas, o funcionamento volta a operar no modo `normal`. A Listagem 6.13 apresenta as regras do modo `degradado`.

```
1  if degradedMode then
2    if allPhysicalUnitsOK then
3      enterNormalMode;
4    elseif failure(waterMeasuring) then
5      enterRescueMode;
6    end
7  end
```

Listagem 6.13: Regras para o modo degradado do sistema de controle

O modo `recuperação` tenta manter o nível da água quando a unidade medidora de nível de água está danificada. Então, para manter o correto nível de água na caldeira, é necessário realizar um cálculo utilizando a medição do vapor e as informações da vazão de cada bomba. Desta forma, caso a unidade medidora de vapor ou alguma bomba esteja danifica, o sistema entra no modo `emergência`. Assim que a unidade medidora de nível de água for reparada, o sistema verifica se todas as unidades físicas estão funcionando. Caso estejam entra no modo `normal`, caso contrário entra no modo `degradado`. As regras do modo `recuperação` são mostradas na Listagem 6.14.

```
1   if rescueMode then
2     if anyPumpsCtrlFailure or failure(steamMeasuring) then
3       enterEmergencyStopMode;
4     elseif not failure(waterMeasuring) then
5       if allPhysicalUnitsOK then
6         enterNormalMode;
7       else
8         enterDegradedMode;
9       end
10    end
11  end
```

Listagem 6.14: Regras para o modo recuperação do sistema de controle

Por fim, o modo `emergência` é o modo que o sistema deve entrar quando alguma unidade vital estiver danificada ou quando o nível de água estiver perto de atingir um dos limites estipulados. Uma vez que o sistema entrou no modo `emergência`, este deve ser finalizado e aguardar até que o ambiente externo tome as providências necessárias.

O módulo principal do sistema, a saber `SteamBoilerControl`, não possui uma regra de transição, uma vez que as regras do sistema foram distribuídas em outros módulos.

Basicamente, este módulo fica responsável por criar os agentes declarados, no módulo `SteamBoilerAgents`, e disparar a execução dos mesmos. Além disso, representa a interface para as unidades físicas, enviando e recebendo informação das mesmas. A Figura 6.3 ilustra a estrutura dos módulos utilizados para o sistema de controle da caldeira a vapor.

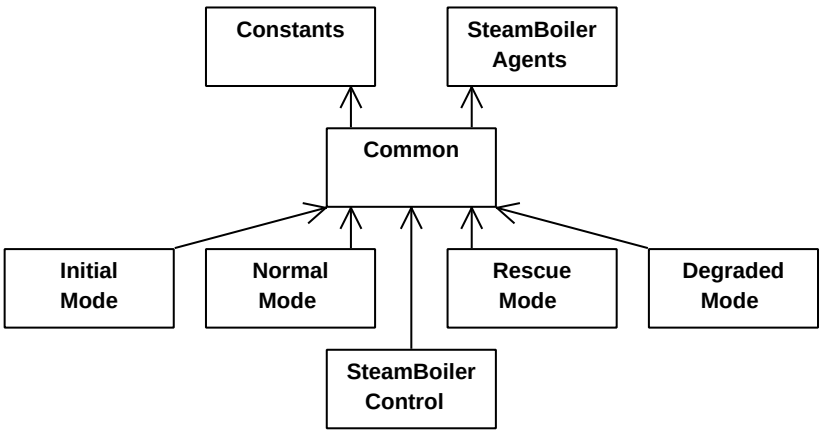

Figura 6.3: Estrutura de módulos para o sistema de controle

6.2.2.3 Interesses Transversais na Caldeira a Vapor

Em [BBD+96], Börger define alguns pré-requisitos globais, o que AOP nos dias atuais denomina de interesses transversais. Basicamente, estes pré-requisitos globais entrecortam a maioria das regras da especificação, causando um entrelaçamento e espalhamento das regras. Para evitar estes problemas na especificação, Börger deixa para o leitor a tarefa de compor o sistema mentalmente, apresentando a especificação de regras sem estes pré-requisitos globais, o que facilita a compreensão e o entendimento da especificação. Este fato fica claro em alguns trechos do artigo, como:

> [*We futher assume that the condition 'phase = executing' specifies a global precondition extending the guards of all the rules*]
>
> [*Although this is not explicitly stated in the informal description, one can reasonably argue that the operations of adjusting the water level to a default value or of maintaining its value within an admissible range are essentially the same for any* mode $\in \{normal, degraded, rescue\}$]
>
> [*Note that we assume the condition* ¬EmergencyStop *to be global precondition extending the guards of all rules - except for global rules*]

Assim, um interesse transversal descrito é a sincronização da comunicação das unidades físicas, denominado no artigo como *Timer Behavior*. Esta comunicação acontece por meio da troca de mensagens, sendo que cada ciclo é constituído de três fases, a saber leitura, execução e escrita. A fase de leitura recebe as mensagens das unidades físicas. Em seguida, na fase de execução, o sistema de controle processa os valores recebidos e, na fase de escrita, envia algumas informações para as unidades físicas. Logo, todas as regras do sistema devem ser executadas apenas da fase de execução. As Listagens 6.15 e 6.16 apresentam trechos para esta codificação. Sem os meios apropriados a precondição `phase=EXECUTING` ficaria espalhada por toda especificação.

```
1  around(agente:Agent): transition(*Unit) and target(agente) do
2    with agente
3     as p:PumpUnit=> if p.phase = p.EXECUTING then proceed end;
4     as pc:PumpCtrlUnit=> if pc.phase = pc.EXECUTING then proceed end;
5     as v:ValveUnit=> if v.phase = v.EXECUTING then proceed end;
6     as m:MeasuringUnit=> if m.phase = m.EXECUTING then proceed end;
7    end;
8  end
```

Listagem 6.15: Sincronização de mensagens para as unidades físicas

```
1  around(agente:Agent): transition(*Mode) and target(agente) do
2   with agente
3     as i:InitialMode => if i.phase = i.EXECUTING then proceed end;
4     as n:NormalMode => if n.phase = n.EXECUTING then proceed end;
5     as r:RescueMode => if r.phase = r.EXECUTING then proceed end;
6     as d:DegradedMode => if d.phase = d.EXECUTING then proceed end;
7   end
8  end
```

Listagem 6.16: Sincronização de mensagens para o sistema de controle

Cabe ressaltar que, por questões didáticas optou-se por separar o tratamento das unidades físicas e do sistema de controle, existindo assim dois módulos para o tratamento da sincronização de mensagens, a saber `TimerUnit` e `TimerMode`.

Outro interesse transversal descrito é o controle do nível de água na caldeira a vapor. Basicamente, deve ser verificado se o nível de água está dentro dos limites normais. Caso esta condição seja desrespeitada, o programa tenta ajustar o nível, tomando as providências necessárias. Convém mencionar que as mensagens à válvula somente deve ser enviada no estado inicial e esta deve permanecer fechada nos demais modos de operação. Desta forma, antes de executar as regras dos modos normal, degradado e recuperação deve-se verificar e ajustar, se necessário, o nível de água. A Listagem 6.17 apresenta esta codificação.

```
before(agente:Agent): transition(*Mode) and target(agente) do
  with agente
    as n:NormalMode   =>
      if n.waterLevelAdjusted then n.retainWaterLevel
      else n.adjustWaterLevel end;
    as r:RescueMode   =>
      if r.waterLevelAdjusted then r.retainWaterLevel
      else r.adjustWaterLevel end;
    as d:DegradedMode =>
      if d.waterLevelAdjusted then d.retainWaterLevel
      else d.adjustWaterLevel end;
  end
end
```

Listagem 6.17: Ajuste do nível de água para o sistema de controle

O tratamento de erro na caldeira a vapor representa mais um interesse transversal. Quando se identifica um erro grave, através da função `emergencyStop`, o sistema deve ser imediatamente transferido para o modo `emergência`, e aguardar que o ambiente externo tome as providências necessárias. Desta maneira, antes de executar as regras, seja das unidades físicas ou dos modos de operação, deve ser verificado que não existe um erro grave. A Listagem 6.18 apresenta esta codificação para os dispositivos físicos. Convém destacar que a função `emergencyStop` é definida por outras 3 funções, a saber `externalStop`, `reachingLimitLevel` e `transmissionFailure`. Como objetivo é apenas demonstrar como este requisito pode ser adicionado às regras da especificação sem causar um entrelaçamento de código, optou-se por declarar essas funções como externas.

```
external PhysicalUnit.externalStop: Bool;
external PhysicalUnit.reachingLimitLevel: Bool;
external PhysicalUnit.transmissionFailure: Bool;
derived function PhysicalUnit.emergencyStop : Bool :=
       reachingLimitLevel or externalStop or transmissionFailure;

around(agente:Agent): transition(*Unit) and target(agente) do
  with agente
    as p:PumpUnit      => if not p.emergencyStop then proceed end;
    as pc:PumpCtrlUnit => if not pc.emergencyStop then proceed end;
    as v:ValveUnit     => if not v.emergencyStop then proceed end;
    as m:MeasuringUnit => if not m.emergencyStop then proceed end;
  end;
end
```

Listagem 6.18: Verificação de erros graves nas unidades físicas

Além dos aspectos relacionados com a sincronização das mensagens e controles globais, para a execução do sistema, especificou-se um módulo contendo aspectos para simularem falhas nos dispositivos e alguns valores. Um trecho da codificação deste

módulo, a saber `SteamBoilerSimulator`, é apresentado na Listagem 6.19. Convém relembrar que a especificação completa encontra-se no Apêndice C.

```
1  static function PhysicalUnit.trueOrFalse :
2                      set of Bool := {true, true, false, true};
3
4  before(p:PumpUnit): transition(PumpUnit) and target(p) do
5    choose e: p.trueOrFalse do p.failure := e end;
6    if p.failure then p.turnOn := false
7    else p.turnOn := true end
8  end
9
10 before(pc:PumpCtrlUnit): transition(PumpCtrlUnit) and target(pc) do
11   choose e: pc.trueOrFalse do pc.failure := e end;
12   if pc.failure then pc.waterCicurlate := false
13   else pc.waterCicurlate := true end
14 end
```

Listagem 6.19: Trecho do módulo com aspectos para simular falhas nos dispositivos

6.2.2.4 O Problemas dos Requisitos Transversais

O objetivo desta seção é apenas possibilitar uma pequena comparação. A Listagem 6.20 mostra as regras do modo `recuperação`. Esta listagem utiliza a versão antiga da linguagem Machĭna, ou seja, o código dos interesses transversais, no contexto do artigo denominado pré-requisitos globais, devem ser especificados junto com as regras, dificultando a sua compreensão e entendimento.

Comparando o código desta listagem com a Listagem 6.14, página 139, pode-se perceber algumas vantagens obtidas com a utilização das novas construções, dentre elas, destacam-se:

- a nova versão da linguagem Machĭna facilita a obtenção de um maior grau de modularização;
- facilita a compreensão e o entendimento. Como as regras ficam mais bem modularizadas, o engenheiro de software não precisa procurar regras espalhadas por todos os módulos da especificação, pois essas estão localizadas em módulos específicos;
- facilita a prova de determinadas propriedades, por exemplo, para provar que as regras das unidades físicas não serão executadas na presença de algum erro grave, representado pela função `EmergencyStop`, basta analisar as 14 linhas da Listagem 6.18, em vez de quase toda a especificação.

```
1  step 1:
2    if phase = EXECUTING then
3      if waterLevelAdjusted then
4        retainWaterLevel
5      else
6        adjustWaterLevel
7      end
8    end
9
10 step 2:
11   if phase = EXECUTING and not emergencyStop then
12     if rescueMode then
13       if anyPumpsCtrlFailure or failure(steamMeasuring) then
14         enterEmergencyStopMode;
15       elseif not failure(waterMeasuring) then
16         if allPhysicalUnitsOK then
17           enterNormalMode;
18         else
19           enterDegradedMode;
20         end
21       end
22     end
23   end;
24   next := 1;
```

Listagem 6.20: Entrelaçamento de código nas regras do modo recuperação

6.2.3 Outros exemplos

Para testar a implementação de mais algumas construções da nova versão da linguagem Machĭna, foram especificados mais alguns exemplos. Assim, nesta seção são descritos estes exemplos. Por questões de simplicidade, omitiu-se a declaração dos módulos de definição Machĭna e algumas interfaces.

Pesquisa Binária

Este exemplo implementa a pesquisa binária. Inicialmente, uma seqüência de números é lida de um arquivo, e logo após, a chave da pesquisa é procurada. Caso a chave seja encontrada é exibido a mensagem `Find !`, caso contrário é exibido a mensagem `Dont Find !`. A Listagem 6.21 apresenta esta codificação.

Um ponto importante deste exemplo, é o módulo `ArrayIndex`, Listagem 6.22. Tal módulo contém um aspecto para controlar o acesso ao vetor que armazena os números lidos, não permitindo assim uma violação na sua indexação.

Ordenação por Seleção

Este exemplo constitui-se da leitura de um arquivo de entrada contendo uma seqüência de números inteiros não ordenados. Em seguida, estes números são ordenados pelo

```
1  module PesquisaBinaria
2    include StringManipulation;
3    algebra:
4      static n : Int := 10;
5      external key : Int;
6      i, k, pos, inf, sup, s : Int;
7      found : Bool;
8      array : Int -> Int;
9      f: Input;
10
11   initial state:
12      open(f,"myvector.dat",s);
13      k := key; pos := (n+1)/2; inf := 1; sup := n;
14      found := false; i := 1;
15
16   transition:
17     step 1: readInt(f,array(i), s); i := i + 1;
18     step 2: if i <= n then next := 1; end
19     step 3: let middle = (inf+sup)/2;
20             if not found and inf < sup then
21               if array(middle) = k then found := true; pos := middle
22               elseif array(middle) < k then inf := middle;
23               else sup := middle;
24               end;
25               next := 3;
26             else
27               if found then printString(" Find !")
28               else printString(" Dont find !")
29               end;
30               destroy self;
31             end
32 end PesquisaBinaria
```

Listagem 6.21: Módulo referente a pesquisa binária

método de ordenação por seleção e são exibidos na tela. São utilizados dois módulos, a saber `SelSort`(6.23) e `StringManipulation`(B.10).

O Jantar dos Filósofos

Uma implementação do problema do jantar dos filósofos é realizada neste exemplo. Esta implementação faz uso dos mecanismos de troca de mensagens entre agentes de modo a evitar *deadlocks* e condições de corrida. São utilizados dois módulos, a saber `Host` 6.24 e `Philosopher` 6.25.

6.3 Considerações Finais

Com o exemplo da linguagem *Tiny*, foi possível mostrar que a nova versão da linguagem Machĭna facilita a obtenção de um maior grau de modularização. A versão

```
1  module ArrayIndex
2   include PesquisaBinaria;
3   aspect:
4     around(agente:PesquisaBinaria ,index:Int):
5        get(dynamic PesquisaBinaria.array(...)) and
6        args(index) and
7        target(agente) do
8            if (index > agente.n) then
9              agente.printString("Array index out of bound !")
10           else
11             proceed
12           end
13    end
14 end ArrayIndex
```

Listagem 6.22: Aspecto para controlar o acesso ao vetor

proposta fornece um novo mecanismo de composição que permite implementar cada interesse separadamente, com o mínimo de acoplamento. O resultado é uma especificação modularizada, mesmo na presença de interesses transversais. Assim, em alguns casos, foi possível reduzir a redundância das regras, uma vez que cada interesse encontra-se separado. Além disso, a especificação bem modularizada facilita a compreensão e o entendimento.

De forma análoga, utilizando as novas construções para especificar o problema da caldeira a vapor, descrito por Abrial, foi possível obter uma especificação mais coesa. As novas construções possibilitam diminuir a responsabilidade dos módulos, permitindo que um módulo seja responsável somente pelo seu interesse principal, não sendo mais entrelaçado por interesses de outros módulos, como os módulos dos modos de operação, que possuem apenas as regras específicas do seu modo.

Além destes exemplos, foram apresentados alguns outros que procuram validar algumas construções implementadas, como o designador `args`, no exemplo da pesquisa binária, que até o momento não tinha sido utilizado.

Por fim, com a implementação destes exemplos, foi possível perceber uma limitação da linguagem proposta. Similar a AspectJ e Eos, os recursos para definição de conjuntos de junção atendem à maioria das necessidades. Mas, como abordado por Breuel [Bre05], muitas vezes eles não permitem criar conjuntos de junção de boa qualidade, tanto no que se refere à resistência a mudanças quanto à clareza.

Breuel [Bre05] realiza uma análise dos trabalhos que têm sido apresentados com relação ao desenvolvimento de conjunto de junção abertos. Por conjunto de junção abertos, entende-se formas de definição de conjunto de junção que permitam uma flexibilidade maior em relação às linguagens existentes. Flexibilidade, dentro deste contexto, pode ser medida em duas dimensões: a granularidade e profundidade da

```
1  module SelSort
2    include StringManipulation;
3    algebra:
4      static n : Int := 4;
5      array : Int -> Int;
6      i, k, j, r, s: Int;
7      g: Input;
8
9    initial state:
10     open(g,"myvector.dat",s);
11     i := 1; k := 1; j:= 1; r:= 1;
12
13   transition:
14     step 1: readInt(g,array(r), s); r := r + 1;
15     step 2: if r <= n then next := 1; end
16     step 3: if i < n then
17               k := i; j := i+1
18             else
19               r := 1;
20               next := 6;
21             end
22     step 4: j := j+1;
23             if j <= n then
24                if array(j) <= array(k) then
25                  k := j; next := 4
26                end
27             end
28     step 5: i := i+1; next := 3;
29             if k != i then
30               array(k) := array(i);
31               array(i) := array(k);
32             end
33     step 6: printQuebraLinha;
34             printInteger(array(r));
35             r := r + 1;
36     step 7: if r <= n then next := 6;
37             else destroy self end;
38
39 end SelSort
```

Listagem 6.23: Módulo referente a ordenação por seleção

informação a que se tem acesso na definição de um conjunto de junção, e a capacidade de combinação dessa informação.

Desta forma, Bruel destaca algumas características que são necessárias e/ou desejáveis na definição de um conjunto de junção:

- Completude: um conjunto de junção deve capturar todos os pontos aos quais o aspecto deve ser aplicado;
- Resistência a mudanças: mudanças no programa devem afetar o mínimo possível a definição dos conjuntos de junção. Em particular, quando novos elementos são

adicionados ao programa, o aspecto deve ser automaticamente aplicado onde for necessário;

- Clareza de intenção: a definição de um conjunto de junção deve transmitir, tanto quanto possível, a intenção de quem o definiu. Ou seja, ao examinar tal definição, um programador sem conhecimento prévio deve ser capaz de entender os critérios que definem o conjunto de junção.

O conjunto de junção representa a interface entre os interesses modulares (ou principais) e os interesses transversais. Desta forma, o conjunto de junção pode ser visto como o ponto principal para analisar a qualidade de uma linguagem orientada por aspectos. Em relação às características destacadas por Bruel, os conjuntos de junção da linguagem proposta apresentam algumas limitações em relação à resistência a mudanças. No decorrer do desenvolvimento dos exemplos, praticamente, toda vez que se optou por modificar um interesse principal (modular), os aspectos aplicados tiverem que ser atualizados.

Apesar desta limitação, consideram-se satisfatórios os resultados obtidos, como nos exemplos apresentados, principalmente no que diz respeito à modularização, o que conseqüentemente melhora a legibilidade e o entendimento da especificação. Além disso, o estudo realizado sobre algumas linguagens que possuem uma maior resistência a mudanças, a saber Gamma e Alpha, revelou que as mesmas não possuem uma boa clareza de intenção, diminuindo a sua usabilidade.

```
1  module Host
2    import Philosopher(setRightFork,setLeftFork);
3    algebra:
4      public type ForkId = Int;
5      fork: ForkId -> Bool;
6      forkId: ForkId;
7      p, t, first, philosophers : agent of Philosopher;
8      agentCount : Int;
9      external numberOfGuests: Int;
10
11   abstractions:
12     public action getForks(l:ForkId, r:ForkId, granted:Bool) is
13       if fork(r) or fork(l) then
14         granted := false;
15       else granted := true;
16         fork(r) := true; fork(l) := true;
17       end
18     end
19     public action freeForks(leftFork:ForkId, rightFork:ForkId) is
20       fork(rightFork) := false; fork(leftFork) := false;
21     end
22
23   initial state:
24     create philosophers(numberOfGuests);
25
26   transition:
27     step 1: p := Philosopher.theAgent(1);
28             first := Philosopher.theAgent(1);
29             agentCount := 2;
30             forkId := 2;
31     step 2: if agentCount <= Philosopher.numberOfAgents then
32             t := Philosopher.theAgent(agentCount);
33             agentCount := agentCount + 1;
34             else next := 4;
35             end
36     step 3: fork(forkId) := false;
37             p.setRightFork(forkId);
38             t.setLeftFork(forkId);
39             p := t;
40             forkId := forkId + 1;
41             next := 2;
42     step 4: fork(1) := false;
43             p.setRightFork(1);
44             first.setLeftFork(1);
45     step 5: forall x:Philosopher.agents do dispatch x end
46
47 end Host
```

Listagem 6.24: Módulo para *Host*

```
1  module Philosopher
2    import Host(ForkId,getForks,freeForks);
3    algebra:
4      leftFork, rightFork: ForkId;
5      thinking, hungry, eating : agent of Philosopher -> Bool;
6      host: agent of Host;
7      granted: Bool;
8
9    abstractions:
10     public action setRightFork(f: ForkId) is
11       rightFork := f;
12     end
13     public action setLeftFork(f: ForkId) is
14       leftFork := f;
15     end
16
17   initial state:
18     host := Host.theAgent;
19     thinking(self) := true;
20
21   transition:
22     step 1: if thinking(self) then
23               thinking(self) := false;
24               hungry(self) := true;
25               next := 1;
26             elseif hungry(self) then
27               host.getForks(leftFork,rightFork,granted);
28             else
29               eating(self) := false;
30               thinking(self) := true;
31               host.freeForks(leftFork,rightFork); next := 1;
32             end;
33     step 2: if granted then
34               eating(self) := true;
35               hungry(self) := false;
36             end;
37             next :=1;
38 end Philosopher
```

Listagem 6.25: Módulo para *Philosopher*

Capítulo 7

Conclusões e Trabalhos Futuros

Neste texto, apresenta-se uma nova versão para a linguagem de especificação formal Machĭna. Na sua versão anterior, haviam problemas de espalhamento e entrelaçamento de regras causado pela especificação de interesses transversais. Assim, além de apresentar a linguagem Machĭna, esta dissertação apresenta alguns desses problemas causados pela especificação dos interesses transversais sem os meios de expressão apropriados, dentre eles destaca-se o baixo grau de reusabilidade e manutenibilidade do sistema.

As tecnologias de programação pós-objeto propõem mecanismos e abstrações inovadores para melhorar a separação de interesses oferecida pelos paradigmas atuais. A POA se concentra em mecanismos para a simplificação da implementação desses interesses transversais . Enquanto a tendência na POO, por exemplo, é encontrar pontos comuns entre as classes e empurrá-las para cima na árvore de herança (generalização), a POA tenta encontrar interesses espalhados, agrupá-los em abstrações de primeira classe, chamados de aspectos, e lançá-los horizontalmente para fora da estrutura dos objetos. Além disso, a POA também oferece novos mecanismos que permitem a composição transparente e flexível destes aspectos.

Laddad [Lad03] destaca alguns benefícios da POA, dentre esses, têm-se:

- Diminuição das responsabilidades dos módulos;
- Modularização;
- Facilidade na evolução do sistema;
- Protelar a implementação de interesses;
- Reusabilidade de código;
- Redução do custo de implementação.

Dentro deste contexto, este trabalho incorporou à linguagem Machĭna novas construções para especificar de forma modular os interesses transversais, unificando as vantagens existentes em uma especificação formal e as vantagens da programação orientada por aspecto. Com isso, obteve-se uma nova versão da linguagem Machĭna, a saber, Aspect$\mathcal{M}$. Assim, neste texto, além de descrever todas esss novas construções, foram apresentados pequenos exemplos que destacam o uso e a sintaxe utilizada.

Como o uso do modelo ASM, no qual a linguagem Machĭna é baseada, possibilita que um algoritmo especificado possa ser executado, foi implementado um compilador. Assim, a linguagem Machĭna e suas novas construções são compiladas para código C++ e AspectC++ respectivamente. Desta forma, no texto foi descrito todo o processo de tradução, podendo-se destacar a utilização dos arcabouços kℓar e ACOA, que agilizaram o desenvolvimento.

Por fim, a validação da nova versão da linguagem Machĭna com seu respectivo compilador foi realizada por meio de exemplos. Os exemplos da linguagem *Tiny* e o problema da caldeira a vapor procuram demonstrar a modularização da especificação obtida. Basicamente, apresentaram-se alguns trechos de código utilizando a versão anterior e a versão proposta para facilitar o entendimento. Além disso, convém destacar que a própria especificação realizada por Börger[BBD$^+$96] deixa clara a existência de interesses transversais no problema da caldeira, e, como no artigo, a separação destes interesses facilita o entendimento e compreensão.

O trabalho apresenta algumas contribuições para a pesquisa na área de máquinas de estado abstratas. Além disso, este trabalho também abre caminho para que novas pesquisas sejam realizadas. Assim, o decorrer do capítulo apresenta estas contribuições e desdobramentos futuros.

7.1 Principais Contribuições

As principais contribuições alcançadas com esta dissertação foram as seguintes:

- Projeto de uma linguagem de programação orientada por aspectos em nível de especificação de requisitos.
- Desenvolvimento de recursos lingüísticos para especificação de requisitos transversais em ambiente de especificação ASM, estendendo assim os conceitos relacionados AOP para ASM.
- Aumento do poder de abstração de uma linguagem de especificação formal, Machĭna, possibilitando a implementação de interesses transversais de forma modular.

- Implementação de um compilador para a nova versão da linguagem Machĭna, possibilitando que as especificações implementadas sejam executadas.

7.2 Trabalhos Futuros

A nova versão da linguagem Machĭna aqui apresentada abre caminho para novos trabalhos a serem desenvolvidos. A seguir, são enumeradas algumas destas possibilidades.

Ferramentas Visuais Os ambientes de programação modernos oferecem diversas facilidades para o desenvolvedor. Cabe citar, a título de exemplo, o *auto-complete* de comandos e expressões, a marcação de pontos de parada para depuração, editores para componentes visuais, destaque de sintaxe, verificação de erros de sintaxe em tempo real etc. O desenvolvimento de ferramentas visuais para a linguagem Aspect$\mathcal{M}$.0 seriam de grande valia para a utilização da linguagem de forma mais produtiva.

Integração e Adequação do Método de Refinamento Machĭna Segundo Stefani [Ste07], o Controle Global do método de refinamento Machĭna modela comportamentos transversais do sistema, porém somente é permitido um diagrama reduzido. Para representações de vários comportamentos, pode ocorrer um entrelaçamento de informações, prejudicando a legibilidade. Um possível trabalho futuro seria permitir, sem causar inconsistências no modelo, adição isolada de novos comportamentos, ou seja, cada novo comportamento teria um diagrama reduzido específico e estes diagramas seriam refinados para as novas construções da linguagem Machĭna.

Verificação de Propriedades adicionadas pelos Aspectos Para verificar um sistema especificado na linguagem Machĭna, Stefani [Ste07] propõe a conversão de Machĭna em NuSMV [NuS07]. Assim, um trabalho futuro seria o estudo da conversão das novas construções de Machĭna em NuSMV ou em outro método de verificação, com o objetivo de verificar se as propriedades adicionadas pelos aspectos estão respeitando o modelo, ou seja, é preciso garantir que a implementação esteja de acordo com a especificação do problema.

Mecanismo de Tratamento de exceções Situações inesperadas podem acontecer na execução de programas, seja por uma entrada incorreta por parte de um usuário, seja pela limitação de um recurso como memória, ou por outro motivo qualquer. As linguagens modernas oferecem mecanismos de alto nível para o tratamento de tais situações, conhecidas como exceções. Um trabalho futuro

consiste no estudo e implementação de um mecanismo de tratamento de exceções adequado à linguagem Aspect$\mathcal{M}$.

Validação Exaustiva Validar exaustivamente o compilador implementado e o uso de Aspect$\mathcal{M}$ na especificação de sistemas de grande porte e complexidade.

Apêndice A

Arquivo para o ACOA

Como o arquivo de entrada para o arcabouço ACOA é muito grande, optou-se por apresentar apenas algunas partes referentes as novas construções propostas.

A.1 Analisador léxico e nodos da AST para os *tokes*

```
%%Lexer

letter          [a-zA-Z_]
number          [0-9]
intnum          ({number})+
fracpart        ([\.])({number})+
exppart         [eE]([-+]?)({number})+
realnum         {number}+({fracpart})?({exppart})?
id              {letter}({letter}|{number})*

%Conditions
%inclusive comentarios

%Pattern
"//"[^\n]*                  ;
"/*"                        -> BEGIN(comentarios);
<comentarios>"*/"           -> END();
<comentarios>"\n"           -> endline;
<comentarios>[^*\n]         ;
<comentarios>"*"+[^*/\n]*   ;
```

```
"aspect"              -> aspect;
"pointcut"            -> pointcut;
"function"            -> function;
"..."                 -> any;
"before"              -> before;
"after"               -> after;
"addition"            -> addition;
"around"              -> around;
"execution"           -> execution;
"get"                 -> get;
"within"              -> within;
"withincode"          -> withincode;
"initialization"      -> initialization;
"target"              -> target;
"proceed"             -> proceed;
"args"                -> ARGS;
```

A.2 Analisador sintático e nodos da AST para as regras gramaticais

```
%%Parser

%right Attr of arrow
%left vertical_bar range
%right coloncolon
%left is in
%left Or Xor
%left And
%left equal dif lt gt ltet gtet
%left add sub
%left mult divi mod
%left Not UNARYSUB
%left old

%start specification_unit
%Grammar

specification_unit ::=
  program_module  -> UnitModule(program_module:prog_module) |
```

```
  agent_interface -> UnitInterface(agent_interface:ag_inter) |
  machina_definition -> UnitDefinition(machina_definition:mc_def);

program_module ::=
  module name module_body end module_name_opt ->
    ProgramModule(name:mod_name, module_body:mod_body,
                  module_name_opt:mod_name_opt);

module_body ::=
  declaration_part initial_state_part_opt transition_part_opt
  aspect_part_opt invariant_part_opt ->
    ModuleBody(declaration_part: declaration, initial_state_part_opt:initialOpt,
               transition_part_opt:trasitionOpt, aspect_part_opt:aspectOpt,
               invariant_part_opt:invariantOpt);

aspect_part_opt ::=
  aspect_part -> ExistAspectPart(aspect_part: asp_part) | ;

aspect_part ::=
  aspect colon aspect_section_listaspect_section ->
    AspectPart(aspect_section_listaspect_section: aspect_sectionList);

aspect_section_listaspect_section ::=
  aspect_section_listaspect_section aspect_section ->
    AspectSectionList(aspect_section_listaspect_section:
                      aspect_sectionList, aspect_section: asp_section) | ;

aspect_section ::=
  inter_type_part -> IntertypePart(inter_type_part: inter_part) |
  pointcut_part -> PointcutPart(pointcut_part: point_part) |
  advice_part -> AdvicePart(advice_part: adv_part);

inter_type_part ::=
  type_section_intertype ->
    IntertypeSection_Type(type_section_intertype: inter_tp_section) |
  function_section_intertype ->
    IntertypeSection_Function(function_section_intertype: inter_func_section) |
  external_section_intertype ->
    IntertypeSection_External(external_section_intertype: inter_ext_section) |
  abstraction_section_intertype ->
```

```
    IntertypeSection_Abstraction(abstraction_section_intertype: inter_abs_section);


pointcut_part ::=
  public_opt pointcut name colon pointcut_designators semicolon ->
    PointcutDeclaration(public_opt: publicOpt, name: namePointcut,
                        pointcut_designators: designators) |
  public_opt pointcut name left_parenthesis function_parameters
  right_parenthesis colon pointcut_designators semicolon ->
    PointcutDeclarationParameter(public_opt: publicOpt, name: namePointcut,
                                 function_parameters: parameter,
                                 pointcut_designators: designators);

advice_part ::=
  advice_type colon pointcut_designators Do single_transition end ->
    AdviceDeclaration(advice_type:adviceType,
                      pointcut_designators:pointcutDesignator,
                      single_transition:body) |
  advice_type left_parenthesis function_parameters right_parenthesis
  colon pointcut_designators Do single_transition end ->
    AdviceDeclarationParameter(advice_type:adviceType,
                               function_parameters:parameter,
                               pointcut_designators:pointcutDesignator,
                               single_transition:body);
```

A.3 Declarações dos passos de compilação

```
%%Passes

ImportPass IncludePass IntertypePass AlgebraPass
TypeCheckPass HeaderFilePass CodeFilePass
```

Apêndice B

Especificação Formal de *Tiny*

As regras de especificação são definidas em seis módulos, a saber:

1. Módulo de Globais (`Globals`) – contém as declarações dos principais tipos e funções da especificação de *Tiny*;
2. Módulo Principal (`MainProgram`) – contém a regra principal da especificação e as rotinas de inicialização;
3. Módulo de Expressões (`Expressions`) – contém as regras de avaliação de expressões;
4. Módulo de Comandos (`Commands`) – contém as regras de execução de comandos;
5. Módulo de Operações (`Operations`) – contém as regras de operação na pilha;
6. Módulo de Tratamento de Erro (`TypeCheck`) – contém a verificação de tipos e alguns tratamentos de erros.

Os módulos `Counting` e `StringManipulation` foram utilizados para possibilitar a execução desta especificação, sendo que o módulo `Counting` define um programa para somar os valores de entrada enquanto que o módulo `StringManipulation` permite a exibição de valores na saída padrão.

B.1 Módulo de Globais - `Globals`

```
1  module Globals
2
3    algebra:
4      public type Undefined = public enum {UNDEFINED};
5      public type Id = String;
6      public type Program = List;
7      public type InputFile = List;
8      public type OutputFile = list of Int;
9      public type Value = Bool | Int;
10     public type Memory = Id -> (Value | Undefined) default UNDEFINED;
11     public type KeyWord = public enum { TADD, TASSIGN, TCOND,
12                     TEXCHANGE, TEQUALS, TNOT, TOUTPUT, TREAD, TWHILE};
13     public type Opstack = List;
14     public shared program : Program;
15     public shared infile : InputFile;
16     public shared outfile : OutputFile;
17     public shared opstack : Opstack;
18     public shared memory : Memory ;
19
20   initial state:
21     infile := nil;
22     program := nil;
23     outfile := nil;
24     opstack := nil;
25
26 end Globals
```

Listagem B.1: Especificação do módulo para globais

B.2 Módulo Principal - MainProgram

```
1  module MainProgram
2
3    include Globals(program,outfile), Commands, Expressions,
4            Operations, Counting, StringManipulation;
5
6    abstractions:
7      public action flattenProgram is
8        let x = head(program);
9        program := concat(List(x),tail(program))
10     end
11
12   initial state:
13     loadProgram(program);
14     infile := 1::2::1::2::1::2::2::4::7::0::nil;
15
16   transition:
17     if program != nil then
18       with head(program)
19         as t:List            => flattenProgram;
20         as s:String          => readExpMemory;
21         as x:Globals.Value   => treatValue;
22         as p:Globals.KeyWord =>
23           case p
24             of Globals.TNOT    => treatExpNot;
25             of Globals.TEQUALS => treatExpEquals;
26             of Globals.TADD    => treatExpAdd;
27             of Globals.TREAD   => treatExpRead;
28             of Globals.TASSIGN => treatComAssign;
29             of Globals.TOUTPUT => treatComOutput;
30             of Globals.TCOND   => treatComConditional;
31             of Globals.TWHILE  => treatComWhile;
32           end;
33       end;
34     else
35       printQuebraLinha;
36       printInteger(Int(head(outfile)));
37       printQuebraLinha;
38       destroy self
39     end;
40
41 end MainProgram
```

Listagem B.2: Especificação do módulo principal

B.3 Módulo de Expressões - Expressions

```
1  module Expressions
2
3    include Globals(program,infile,opstack,memory);
4
5    abstractions:
6
7      public action readExpMemory is
8        let id = String (head(program));
9        program := memory(id) :: tail(program)
10     end
11
12     public action treatExpNot is
13       program := tail(program);
14       opstack := Globals.TNOT::opstack
15     end
16
17     public action treatExpEquals is
18       program := tail(program);
19       opstack := Globals.TEQUALS::opstack
20     end
21
22     public action treatExpAdd is
23       program := tail(program);
24       opstack := Globals.TEXCHANGE::Globals.TADD::opstack
25     end;
26
27     public action treatExpRead is
28       program := head(infile)::tail(program);
29       infile := tail(infile)
30     end
31
32 end Expressions
```

Listagem B.3: Especificação do módulo de expressões

B.4 Módulo de Comandos - Commands

```
1  module Commands
2
3    include Globals(program,opstack);
4
5    abstractions:
6
7      public action treatComAssign is
8        program := tail(tail(program));
9        opstack := Globals.TASSIGN::head(tail(program))::opstack
10     end
11
12     public action treatComOutput is
13       program := tail(program);
14       opstack := Globals.TOUTPUT::opstack
15     end
16
17     public action treatComConditional is
18       program := tail(program);
19       opstack := Globals.TCOND::opstack;
20     end
21
22     public action treatComWhile is
23       let second = head(tail(program));
24       let third = head(tail(tail(program)));
25       let whileExp = second;
26       let whileCom = Globals.TWHILE::second::third::nil;
27       let whileTrue = third::whileCom::nil;
28       let whileCont = tail(tail(tail(program)));
29       program := whileExp::whileTrue::nil::whileCont;
30       opstack := Globals.TCOND::opstack
31     end
32
33 end Commands
```

Listagem B.4: Especificação do módulo de comandos

B.5 Módulo de Operações - `Operations`

```
1  module Operations
2
3    include Globals(program,outfile,opstack,memory);
4
5    abstractions:
6
7      public action treatOutput is
8          let x = head(program);
9          outfile := outfile::Int(x);
10         opstack := tail(opstack);
11         program := tail(program);
12     end
13
14     public action treatAssign is
15         opstack := tail(tail(opstack));
16         let id = String(head(tail(opstack)));
17         memory(id) := Globals.Value(head(program));
18         program := tail(program);
19     end
20
21     public action treatExchange is
22         let x = head(program);
23         let y = head(tail(program));
24         opstack := tail(opstack);
25         program := y::x::tail(tail(program))
26     end
27
28     public action treatAdd is
29         let x = head(program);
30         let y = head(tail(program));
31         program := (Int(x) + Int(y))::tail(tail(program));
32         opstack := tail(opstack)
33     end
34         .
35         .
36         .
```

Listagem B.5: Especificação do módulo de operações 1/2

```
1
2       public action treatNot is
3           let b = head(program);
4           program := (not (Bool(b)))::tail(program);
5           opstack := tail(opstack)
6       end
7
8       public action treatCond is
9         if Bool(head(program)) then
10           program := head(tail(program))::tail(tail(tail(program)))
11        else program := tail(tail(program))
12        end;
13        opstack := tail(opstack);
14      end
15
16      public action treatEquals is
17          let x = head(program);
18          let y = head(tail(program));
19          program := (Int(x) = Int(y))::tail(tail(program));
20          opstack := tail(opstack)
21      end
22
23      public action treatValue is
24        case Globals.KeyWord (head(opstack))
25           of Globals.TOUTPUT => treatOutput;
26           of Globals.TASSIGN => treatAssign;
27           of Globals.TEXCHANGE => treatExchange;
28           of Globals.TADD => treatAdd;
29           of Globals.TNOT => treatNot;
30           of Globals.TCOND => treatCond;
31           of Globals.TEQUALS => treatEquals;
32        end
33      end
34
35  end Operations
```

Listagem B.6: Especificação do módulo de operações 2/2

B.6 Módulo de Tratamento de Erro - TypeCheck

```
1  module TypeCheck
2
3    include Operations, Expressions, MainProgram;
4
5    aspect:
6      public shared Globals.error : Bool;
7      pointcut executeAdd : execution(action Operations.treatAdd);
8      pointcut executeOutput : execution(action Operations.treatOutput);
9      pointcut executeNot : execution(action Operations.treatNot);
10     pointcut executeCond : execution(action Operations.treatCond);
11     pointcut executeEquals : execution(action Operations.treatEquals);
12     pointcut executeValue : execution(action Operations.treatValue);
13
14     around(op:Operations):
15       (executeAdd or executeEquals) and target(op) do
16         let x = head(op.program);
17         let y = head(tail(op.program));
18         if (x is Int) and (y is Int) then proceed
19         else op.error := true
20         end;
21     end
22
23     around(op:Operations):
24       (executeNot or executeCond) and target(op) do
25         let x = head(op.program);
26         if (x is Bool) then proceed
27         else op.error := true
28         end;
29     end
30
31     around(op:Operations): executeOutput and target(op) do
32         let x = head(op.program);
33         if (x is Int) then proceed
34         else op.error := true
35         end;
36     end
37
38     around(op:Operations): executeValue and target(op) do
39         let x = head(op.opstack);
40         if (x is Globals.KeyWord) then proceed
41         else op.error := true
42         end;
43     end
44     .
45     .
46     .
```

Listagem B.7: Especificação do módulo para o tratamento de erro (1/2)

```
1
2       around(op:Operations): executeValue and target(op) do
3           let x = head(op.opstack);
4           if (x is Globals.KeyWord) then proceed
5           else op.error := true
6           end;
7       end
8
9       pointcut executeMemory:execution(action Expressions.readExpMemory);
10      pointcut executeRead:execution(action Expressions.treatExpRead);
11
12      around(ex:Expressions): executeMemory and target(ex) do
13        let id = head(ex.program);
14        if (id is String) then
15          if (ex.memory(String(id)) is Globals.Undefined) then
16            ex.error := true
17          else
18            proceed
19          end
20        else ex.error := true
21        end;
22      end
23
24      around(ex:Expressions): executeRead and target(ex) do
25        if ex.infile = nil then
26          ex.error := true
27        end
28      end
29
30      around(agente:MainProgram):
31        transition(MainProgram) and target(agente) do
32          if agente.error then
33            agente.printString("Erro durante a execução!");
34            destroy agente;
35          else
36            proceed
37          end
38      end
39
40  end TypeCheck
```

Listagem B.8: Especificação do módulo para o tratamento de erro (2/2)

B.7 Exemplo de um programa - `Counting`

```
1  module Counting
2
3    include Globals(Program);
4
5    algebra:
6      initialCounter : Program;
7      readInputToX : Program;
8      test : Program;
9      cmd : Program;
10     cmd2 : Program;
11     printCounter : Program;
12     numberCounter : Program;
13
14   initial state:
15     initialCounter := Globals.TASSIGN::"sum"::0::nil;
16     readInputToX := Globals.TASSIGN::"x"::Globals.TREAD::nil;
17     test := Globals.TNOT::Globals.TEQUALS::"x"::0::nil;
18     cmd := Globals.TASSIGN::"sum"::Globals.TADD::"sum"::"x"::nil;
19     cmd2 := Globals.TASSIGN::"x"::Globals.TREAD::nil;
20     printCounter := Globals.TOUTPUT::"sum"::nil;
21
22   abstractions:
23     public action loadProgram(inout prog:List) is
24     loop:
25       step 1:
26         cmd := cmd::cmd2::nil;
27       step 2:
28         numberCounter := Globals.TWHILE::test::cmd::nil;
29       step 3:
30         prog := initialCounter::
31                 readInputToX::
32                   numberCounter::
33                 printCounter::nil;
34         return;
35     end
36
37 end Counting
```

Listagem B.9: Especificação do módulo que carrega um exemplo

B.8 Manipulação de Strings - StringManipulation

```
1  module StringManipulation
2
3    algebra:
4      external IntegerToString(i:Int):Int;
5      external RealToString(i:Real):Int;
6      external WriteExtFunc(s:String):Int;
7      dynamic retorno:Int;
8
9    abstractions:
10     public action printInteger(in a:Int) is
11       retorno := WriteExtFunc(IntegerToString(a));
12     end
13
14     public action printReal(in b:Real) is
15       retorno := WriteExtFunc(RealToString(b));
16     end
17
18     public action printString(in c:String) is
19       retorno := WriteExtFunc(c);
20     end
21
22     public action printQuebraLinha is
23       retorno := WriteExtFunc("\n");
24     end
25
26 end StringManipulation
```

Listagem B.10: Especificação do módulo para manipulação de strings para a saída padrão

Apêndice C

Especificação Formal da Caldeira a Vapor

A especificação da caldeira foi divida em XX módulos, sendo que esses podem ser agrupados em dois grupos, a saber: sistema controlador da caldeira e unidades físicas. O primeiro grupo contém os seguintes módulos:

1. Módulo de Constantes (`Constants`) – contém as declarações das constantes utilizadas no sistema especificação;
2. Módulo de Agentes (`SteamBoilerAgents`) – contém a declaração dos agentes utilizados;
3. Módulo de Funções Comuns aos Agentes (`Common`) – contém as principais funções e regras que são comuns aos agentes relacionados aos modos de operação;
4. Módulo Principal (`SteamBoilerControl`) – além de criar e disparar os agentes, recebe e processa as informações enviadas pelos dispositivos;
5. Módulos para cada modo de operação(`InitialMode, NormalMode, DegradedMode e RescueMode`) – cada módulo contém as regras referentes ao seu modo de operação.
6. Módulo de Sincronização (`TimerMode`) - sincroniza o envio das mensagens para os modos de operação.
7. Módulo de Controle Global (`GlobalCtrlMode`) - contém as regras que entrecortam os modos de operação.

C.1 Sistema Controlador da Caldeira

C.1.1 Módulo de Constantes (Constants)

```
1  module Constants
2
3    algebra:
4      public static WaterId : Int := 1;
5      public static SteamId : Int := 2;
6      public static C : Int := 250;
7      public static M1 : Int := 10;
8      public static M2 : Int := 240;
9      public static N1 : Int := 30;
10     public static N2 : Int := 230;
11     public static W : Int := 5;
12     public static U1 : Int := 1;
13     public static U2 : Int := 4;
14     public static numberOfPumps : Int := 2;
15
16 end Constants
```

Listagem C.1: Constantes utilizadas no sistema

C.1.2 Módulo de Agentes (SteamBoilerAgents)

```
1  module SteamBoilerAgents
2
3    import InitialMode, NormalMode, RescueMode, DegradedMode,
4           MeasuringUnit, PumpUnit, PumpCtrlUnit, ValveUnit;
5
6    algebra:
7      public shared waterMeasuring: agent of MeasuringUnit;
8      public shared steamMeasuring: agent of MeasuringUnit;
9      public shared pumpsCtrl: agent of PumpCtrlUnit;
10     public shared valve: agent of ValveUnit;
11     public shared pumps: agent of PumpUnit;
12
13     public shared initialRule: agent of InitialMode;
14     public shared normalRule: agent of NormalMode;
15     public shared degradedRule: agent of DegradedMode;
16     public shared rescueRule: agent of RescueMode;
17
18 end SteamBoilerAgents
```

Listagem C.2: Declaração dos agentes utilizados na especificação

C.1.3 Módulo de Funções Comuns aos Agentes (Common)

```
1  module Common
2
3   include Constants, SteamBoilerAgents;
4
5   algebra:
6
7    public type Mode = public enum { INITIAL, NORMAL, DEGRADED,
8                                     RESCUE, EMERGENCY} default INITIAL;
9    external steamBoilerWaiting : Bool;
10   public shared ready: Agent -> Bool;
11   public shared failure: Agent -> Bool;
12   public shared turnOn: Agent -> Bool;
13   public shared waterThroughput: Agent -> Bool;
14   public shared waterLevel : Int;
15   public shared steamLevel : Int;
16   public shared valveOpened : Bool;
17   public shared steamBoilerMode : Mode;
18
19   derived waterLevelBelowMim:Bool := waterLevel < N1;
20   derived waterLevelAdjusted:Bool := waterLevel >= N1 and
21                                      waterLevel <= N2 ;
22   derived allPumpsReady:Bool := all pump:PumpUnit.agents
23                                 satisfying ready(pump);
24   derived allPumpsOK:Bool := all pump:PumpUnit.agents
25                              satisfying not failure(pump);
26   derived allPumpsCtrlReady:Bool := all ctrl:PumpCtrlUnit.agents
27                                    satisfying ready(ctrl);
28   derived allPumpsCtrlOK:Bool := all ctrl:PumpCtrlUnit.agents
29                                 satisfying not failure(ctrl);
30   derived anyPumpsCtrlFailure:Bool := exist ctrl:PumpCtrlUnit.agents
31                                       satisfying failure(ctrl);
32   derived allMeasuringReady:Bool := all unit:MeasuringUnit.agents
33                                    satisfying ready(unit);
34   derived allMeasuringOK:Bool:= all unit:MeasuringUnit.agents
35                                 satisfying not failure(unit);
36   derived anyMeasuringFailure:Bool := exist unit:MeasuringUnit.agents
37                                       satisfying failure(unit);
38   derived allPhysicalUnitsOK:Bool := allPumpsOK and allPumpsCtrlOK and
39                                      allMeasuringOK and
40                                      not failure(valve);
41     .
42     .
43     .
```

Listagem C.3: Funções e regras que são comuns aos agentes (1/3)

```
1    derived physicalUnitsReady:Bool := allPumpsReady and
2                                       allMeasuringReady and
3                                       allPumpsCtrlReady and
4                                       ready(valve);
5
6   abstractions:
7
8     public action enterNormalMode is
9       steamBoilerMode := NORMAL;
10    end
11    public action enterDegradedMode is
12      steamBoilerMode := DEGRADED;
13    end
14    public action enterRescueMode is
15      steamBoilerMode := RESCUE;
16    end
17    public action enterEmergencyStopMode is
18      steamBoilerMode := EMERGENCY;
19    end
20    public action stopPumps is
21      forall pump:PumpUnit.agents do pump.setTurnOn(false) end;
22    end
23    public action stopSomePumps is
24      choose pump:PumpUnit.agents satisfying turnOn(pump)
25        do  pump.setTurnOn(false)
26      end;
27    end
28    public action activatePumps is
29      choose pump:PumpUnit.agents satisfying not turnOn(pump)
30        do pump.setTurnOn(true)
31      end;
32    end
33    public action openValve is
34      valve.setOpened(true);
35    end
36    public action closeValve is
37      valve.setOpened(false);
38    end
39    .
40    .
41    .
```

Listagem C.4: Funções e regras que são comuns aos agentes (2/3)

```
1
2       public action retainWaterLevel  is
3         stopPumps;
4         closeValve;
5       end
6       public action adjustWaterLevel is
7         if steamBoilerWaiting then
8           if waterLevelBelowMim then
9             raiseWaterLevel;
10          else
11            reduceWaterLevel;
12          end
13        end
14      end
15      public action raiseWaterLevel is
16        activatePumps;
17        if steamBoilerMode = INITIAL then
18          closeValve;
19        end;
20        waterLevel := waterLevel+10;
21        waterMeasuring.setMeasure(waterLevel+10);
22      end
23      public action reduceWaterLevel is
24        if steamBoilerMode = INITIAL then
25          stopPumps;
26          openValve;
27        else
28          stopSomePumps;
29        end;
30        waterLevel := waterLevel-10;
31        waterMeasuring.setMeasure(waterLevel-10);
32      end
33
34  end Common
```

Listagem C.5: Funções e regras que são comuns aos agentes (3/3)

C.1.4 Módulo Principal (SteamBoilerControl)

```
1  module SteamBoilerControl
2
3    include Common;
4
5    abstractions:
6      public action recievePumpInfo(in a:Agent, in r:Bool,
7                                    in f:Bool, in on:Bool)  is
8        ready(a) := r;
9        failure(a) := f;
10       turnOn(a):= on;
11     end
12     public action recieveMeasuringInfo(in a:Agent, in r:Bool,
13                                        in f:Bool, in m:Int,
14                                        in id:Int) is
15       ready(a) := r;
16       failure(a) := f;
17       if id = WaterId then waterLevel := m;
18       else steamLevel := m;
19       end;
20     end
21     public action recieveValveInfo(in a:Agent, in r:Bool,
22                                    in f:Bool, in o:Bool)  is
23       ready(a) := r;
24       failure(a) := f;
25       valveOpened := o;
26     end
27     public action recievePumpCtrlInfo(in a:Agent, in r:Bool,
28                                       in f:Bool, water:Bool)  is
29       ready(a) := r;
30       failure(a) := f;
31       waterThroughput(a):= water;
32     end
33     .
34     .
35     .
```

Listagem C.6: Módulo Principal (1/2)

```
1     public action startSteamBoiler is
2     loop:
3       step 1:
4         create pumpsCtrl(numberOfPumps);
5         create pumps(numberOfPumps);
6         create waterMeasuring;
7         create steamMeasuring;
8         create valve;
9         create initialRule;
10        create normalRule;
11        create degradedRule;
12        create rescueRule;
13      step 2:
14        waterMeasuring.setIdMeasuring(WaterId);
15        steamMeasuring.setIdMeasuring(SteamId);
16      step 3:
17        forall pump:PumpUnit.agents do dispatch pump end;
18        forall ctrl:PumpCtrlUnit.agents do dispatch ctrl end;
19        forall unit:MeasuringUnit.agents do dispatch unit end;
20        dispatch valve;
21        dispatch initialRule;
22        dispatch normalRule;
23        dispatch degradedRule;
24        dispatch rescueRule;
25        return
26    end
27
28  initial state:
29    startSteamBoiler
30
31 end SteamBoilerControl
```

Listagem C.7: Módulo Principal (2/2)

```
1 interface SteamBoilerControl
2   action recievePumpInfo(in a:Agent, in r:Bool, in f:Bool, on:Bool);
3   action recieveValveInfo(in a:Agent, in r:Bool, in f:Bool, o:Bool);
4   action recievePumpCtrlInfo(in a:Agent, in r:Bool, in f:Bool,
5                              in water:Bool);
6   action recieveMeasuringInfo(in a:Agent, in r:Bool, in f:Bool,
7                               in m:Int, in id:Int);
8 end SteamBoilerControl
```

Listagem C.8: Interface para o modulo `SteamBoilerControl`

C.1.5 Módulo para o Modo de Inicialização(InitialMode)

```
1  module InitialMode
2
3    include Common;
4
5    algebra:
6      derived initMode : Bool := steamBoilerMode = INITIAL;
7
8    abstractions:
9      public action indicateProgramReady is
10       forall pump:PumpUnit.agents do pump.sendProgramReady end;
11       forall ctrl:PumpCtrlUnit.agents do ctrl.sendProgramReady end;
12       forall unit:MeasuringUnit.agents do unit.sendProgramReady end;
13       forall valve:ValveUnit.agents do valve.sendProgramReady end;
14     end
15
16   transition:
17       if initMode and steamBoilerWaiting and anyMeasuringFailure then
18         enterEmergencyStopMode;
19       end;
20       if initMode and steamBoilerWaiting and waterLevelAdjusted and
21          not physicalUnitsReady then
22         indicateProgramReady;
23       end;
24       if initMode and steamBoilerWaiting and physicalUnitsReady then
25         if waterLevelAdjusted and not anyMeasuringFailure then
26           if allPhysicalUnitsOK then
27             enterNormalMode
28           else
29             enterDegradedMode
30           end
31         else
32           enterEmergencyStopMode
33         end
34       end;
35
36 end InitialMode
```

Listagem C.9: Regras para o modo de inicialização

C.1.6 Módulo para o Modo Normal(NormalMode)

```
1  module NormalMode
2    include Common;
3    algebra:
4      derived normalMode : Bool := steamBoilerMode = NORMAL;
5
6    transition:
7      if normalMode and not allPhysicalUnitsOK then
8        if failure(waterMeasuring) then
9          enterRescueMode;
10       else
11         enterDegradedMode;
12       end
13     end;
14 end NormalMode
```

Listagem C.10: Regras para o modo normal

C.1.7 Módulo para o Modo Degradado(DegradedMode)

```
1  module DegradedMode
2    include Common;
3    algebra:
4      derived degradedMode : Bool := steamBoilerMode = DEGRADED;
5
6    transition:
7      if degradedMode then
8        if allPhysicalUnitsOK then
9          enterNormalMode;
10       elseif failure(waterMeasuring) then
11         enterRescueMode;
12       end
13     end
14 end DegradedMode
```

Listagem C.11: Regras para o modo degradado

C.1.8 Módulo para o Modo de Recuperação(RescueMode)

```
1  module RescueMode
2    include Common;
3    algebra:
4      derived rescueMode : Bool := steamBoilerMode = RESCUE;
5
6    transition:
7      if rescueMode then
8        if anyPumpsCtrlFailure or failure(steamMeasuring) then
9          enterEmergencyStopMode;
10       elseif not failure(waterMeasuring) then
11         if allPhysicalUnitsOK then
12           enterNormalMode;
13         else
14           enterDegradedMode;
15         end
16       end
17     end;
18 end RescueMode
```

Listagem C.12: Regras para o modo de recuperação

C.1.9 Módulo de Sincronização (TimerMode)

```
1  module TimerMode
2
3    include Common, SteamBoilerControl, InitialMode, NormalMode,
4            RescueMode, DegradedMode;
5
6    aspect:
7      public type Common.Phase =
8      public enum {READING, EXECUTING, WRITING} default READING;
9      public shared function Common.phase : Phase;
10     external Common.inNextCycle:Bool;
11
12     addition(sb:SteamBoilerControl):
13       transition(SteamBoilerControl) and target(sb) do
14         if sb.phase = sb.READING and sb.inNextCycle then
15           sb.phase := sb.EXECUTING;
16           end;
17         if sb.phase = sb.EXECUTING then
18           sb.phase := sb.WRITING;
19         end;
20         if sb.phase = sb.WRITING then
21           sb.phase := sb.READING;
22         end;
23     end
24
25     around(agente:Agent): transition(*Mode) and target(agente) do
26       with agente
27         as i:InitialMode =>
28           if i.phase = i.EXECUTING then proceed end;
29         as n:NormalMode =>
30           if n.phase = n.EXECUTING then proceed end;
31         as r:RescueMode =>
32            if r.phase = r.EXECUTING then proceed end;
33         as d:DegradedMode =>
34            if d.phase = d.EXECUTING then proceed end;
35         otherwise => proceed;
36       end;
37     end
38
39 end TimerMode
```

Listagem C.13: Sincronização para os modos de operação

C.1.10 Módulo de Controle Global (`GlobalCtrlMode`)

```
1  module GlobalCtrlMode
2
3    include Common, SteamBoilerControl, InitialMode, NormalMode,
4            RescueMode, DegradedMode;
5
6    aspect:
7      external Common.externalStop: Bool;
8      external Common.reachingLimitLevel: Bool;
9      external Common.transmissionFailure: Bool;
10     derived function Common.emergencyStop : Bool :=
11         reachingLimitLevel or externalStop or transmissionFailure;
12
13     before(agente:Agent): transition(*Mode) and target(agente) do
14       with agente
15         as i:InitialMode =>
16           if i.emergencyStop then i.enterEmergencyStopMode end;
17         as n:NormalMode =>
18           if n.emergencyStop then n.enterEmergencyStopMode end;
19           if n.waterLevelAdjusted then n.retainWaterLevel
20           else n.adjustWaterLevel end;
21         as r:RescueMode =>
22           if r.emergencyStop then r.enterEmergencyStopMode end;
23           if r.waterLevelAdjusted then r.retainWaterLevel
24           else r.adjustWaterLevel end;
25         as d:DegradedMode =>
26           if d.emergencyStop then d.enterEmergencyStopMode end;
27           if d.waterLevelAdjusted then d.retainWaterLevel
28           else d.adjustWaterLevel end;
29       end;
30     end
31
32     around(agente:Agent): transition(*Mode) and target(agente) do
33       with agente
34         as i:InitialMode  => if not i.emergencyStop then proceed end;
35         as n:NormalMode   => if not n.emergencyStop then proceed end;
36         as r:RescueMode   => if not r.emergencyStop then proceed end;
37         as d:DegradedMode => if not d.emergencyStop then proceed end;
38         otherwise         => proceed;
39       end;
40     end
41
42 end GlobalCtrlMode
```

Listagem C.14: Funções e regras que entrecortam os modos de operação

C.2 Unidades Físicas

O segundo grupo, unidades físicas, contém os módulos apresentados abaixo.

1. Módulo de Unidade Física (`PhysicalUnit`) – contém as principais funções e abstrações que são comuns as unidades físicas;

2. Módulo para a Bomba (`PumpUnit`) – – contém as funções e abstrações referentes as bombas;

3. Módulo para o Controlador da Bomba (`PumpCtrlUnit`) – contém as funções e abstrações referentes aos controladores de bomba;

4. Módulo para Válvula (`ValveUnit`) – contém as funções e abstrações referentes a válvula;

5. Módulo para Unidades de Medidas (`MeasuringUnit`) – contém as funções e abstrações referentes as unidades de medida.

6. Módulo de Sincronização (`TimerUnit`) - sincroniza o envio das mensagens para o sistema de controle.

7. Módulo de Controle Global (`GlobalCtrlUnit`) - contém as regras que entrecortam as unidades físicas.

C.2.1 Módulo de Unidade Física (PhysicalUnit)

```
1  module PhysicalUnit
2    import SteamBoilerControl;
3    algebra:
4      dynamic failure: Bool;
5      dynamic steamBoiler: agent of SteamBoilerControl;
6      dynamic programReady: Bool;
7      dynamic ready: Bool := true;
8
9    abstractions:
10     public action setReady (in r:Bool) is
11       ready := r
12     end
13     public action setFailure (in f:Bool) is
14       failure := f
15     end
16     public action sendProgramReady is
17       programReady := true;
18     end
19 end PhysicalUnit
```

Listagem C.15: Módulo comum para as unidades físicas

C.2.2 Módulo para a Bomba (PumpUnit)

```
1  module PumpUnit
2
3    include PhysicalUnit;
4
5    algebra:
6      dynamic turnOn: Bool := false;
7
8    abstractions:
9      public action setTurnOn (in turnOnPump:Bool) is
10       turnOn := turnOnPump
11     end
12
13   initial state:
14     steamBoiler := SteamBoilerControl.theAgent;
15
16   transition:
17     if programReady then
18       steamBoiler.recievePumpInfo(self, ready, failure, turnOn);
19     end;
20
21 end PumpUnit
```

Listagem C.16: Módulo para representar as bombas

```
1  interface PumpUnit
2    action setReady (in readyPump:Bool);
3    action setFailure (in failurePump:Bool);
4    action setTurnOn (in turnOnPump:Bool);
5    action sendProgramReady;
6  end PumpUnit
```

Listagem C.17: Interface para o modulo `PumpUnit`

C.2.3 Módulo para o Controlador da Bomba (`PumpCtrlUnit`)

```
1  module PumpCtrlUnit
2    include PhysicalUnit;
3    algebra:
4      dynamic waterThroughput: Bool;
5
6    abstractions:
7      public action setWaterThroughput (in water:Bool) is
8        waterThroughput := water
9      end
10
11   initial state:
12     steamBoiler := SteamBoilerControl.theAgent;
13
14   transition:
15     if programReady then
16       steamBoiler.recievePumpCtrlInfo(self, ready,
17                                       failure, waterThroughput);
18     end;
19 end PumpCtrlUnit
```

Listagem C.18: Módulo para representar os controladores de bomba

```
1  interface PumpCtrlUnit
2    action setReady (in readyPump:Bool);
3    action setFailure (in failurePump:Bool);
4    action setWaterCicurlate (in water:Bool);
5    action sendProgramReady;
6  end PumpCtrlUnit
```

Listagem C.19: Interface para o modulo `PumpCtrlUnit`

C.2.4 Módulo para Válvula (`ValveUnit`)

```
1  module ValveUnit
2
3    include PhysicalUnit;
4
5    algebra:
6      opened: Bool := false;
7
8    abstractions:
9      public action setOpened (in openedValve:Bool) is
10       opened := openedValve
11     end
12
13   initial state:
14     steamBoiler := SteamBoilerControl.theAgent;
15
16   transition:
17     if programReady then
18       steamBoiler.recieveValveInfo(self, ready, failure, opened);
19     end;
20
21 end ValveUnit
```

Listagem C.20: Módulo para representar a válvula

```
1 interface ValveUnit
2   action setReady (in readyValve:Bool);
3   action setFailure (in failureValve:Bool);
4   action setOpened (in openedValve:Bool);
5   action sendProgramReady;
6 end ValveUnit
```

Listagem C.21: Interface para o modulo `ValveUnit`

C.2.5 Módulo para Unidade de Medida (`MeasuringUnit`)

```
1  module MeasuringUnit
2
3    include PhysicalUnit;
4
5    algebra:
6      dynamic idMeasuring: Int;
7      dynamic measure: Int;
8
9    abstractions:
10     public action setMeasure (in measureUnit:Int) is
11       measure := measureUnit
12     end
13     public action setIdMeasuring (in id:Int) is
14       idMeasuring := id
15     end
16
17   initial state:
18     steamBoiler := SteamBoilerControl.theAgent;
19
20   transition:
21     if programReady then
22       steamBoiler.recieveMeasuringInfo(self, ready, failure,
23                                        measure, idMeasuring);
24     end;
25
26 end MeasuringUnit
```

Listagem C.22: Módulo para representar as unidades de medida

```
1  interface MeasuringUnit
2    type MeasuringType;
3    action setReady (in readyUnit:Bool);
4    action setFailure (in failureUnit:Bool);
5    action setMeasure (in measureUnit:Int);
6    action setIdMeasuring (in id:Int);
7    action sendProgramReady;
8  end MeasuringUnit
```

Listagem C.23: Interface para o modulo `MeasuringUnit`

C.2.6 Módulo de Sincronização (TimerUnit)

```
1  module TimerUnit
2
3   include PhysicalUnit, PumpUnit, PumpCtrlUnit,
4           ValveUnit, MeasuringUnit;
5
6   aspect:
7      public type PhysicalUnit.Phase =
8         public enum {READING, EXECUTING, WRITING} default READING;
9      public shared function PhysicalUnit.phase : Phase;
10    external PhysicalUnit.inNextCycle:Bool;
11
12     addition(p:PumpUnit): transition(PumpUnit) and target(p) do
13       if p.phase = p.READING  and inNextCycle then
14        p.phase := p.EXECUTING;
15       end;
16       if p.phase = p.EXECUTING then
17        p.phase := p.WRITING;
18       end;
19       if p.phase = p.WRITING then
20        p.phase := p.READING;
21       end;
22     end
23
24    around(agente:Agent): transition(*Unit) and target(agente) do
25      with agente
26       as p:PumpUnit=> if p.phase = p.EXECUTING then proceed end;
27       as pc:PumpCtrlUnit=> if pc.phase = pc.EXECUTING then proceed end;
28       as v:ValveUnit=> if v.phase = v.EXECUTING then proceed end;
29       as m:MeasuringUnit=> if m.phase = m.EXECUTING then proceed end;
30       otherwise => proceed;
31      end;
32    end
33
34 end TimerUnit
```

Listagem C.24: Sincronização para as unidades de físicas

C.2.7 Módulo de Controle Global (`GlobalCtrlUnit`)

```
1  module GlobalCtrlUnit
2
3    include PhysicalUnit, PumpUnit, PumpCtrlUnit,
4            ValveUnit, MeasuringUnit;
5
6    aspect:
7      external PhysicalUnit.externalStop: Bool;
8      external PhysicalUnit.reachingLimitLevel: Bool;
9      external PhysicalUnit.transmissionFailure: Bool;
10     derived function PhysicalUnit.emergencyStop : Bool :=
11         reachingLimitLevel or externalStop or transmissionFailure;
12
13     around(agente:Agent): transition(*Unit) and target(agente) do
14       with agente
15         as p:PumpUnit =>
16           if not p.emergencyStop then proceed end;
17         as pc:PumpCtrlUnit =>
18           if not pc.emergencyStop then proceed end;
19         as v:ValveUnit =>
20           if not v.emergencyStop then proceed end;
21         as m:MeasuringUnit =>
22           if not m.emergencyStop then proceed end;
23         otherwise => proceed;
24       end;
25     end
26
27 end GlobalCtrlUnit
```

Listagem C.25: Funções e regras que entrecortam as unidades de físicas

Referências Bibliográficas

[ABL95] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, *The stream boiler case study: Competition of formal program specification and development methods.*, Formal Methods for Industrial Applications, 1995, pp. 1–12.

[Abr94] Jean-Raymond Abrial, *A steam-boiler control specification problem*, August 1994.

[AC02] Fernando Asteasuain and Bernardo Ezequiel Contreras, *Programación orientada a aspectos: Análisis del paradigma*, Tese de licenciatura, Universidad Nacional del SUR - Bahía Blanca. Buenos Aires. Argentina, octuber 2002.

[Anl00] M. Anlauff, *XASM – An Extensible, Component-Based Abstract State Machines Language*, Abstract State Machines: Theory and Applications (Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, ed.), LNCS, vol. 1912, Springer-Verlag, 2000, pp. 69–90.

[Asp06a] AspectC++, http://www.aspectc.org, último acesso: 30 de abril de 2006., 2006.

[Asp06b] AspectWerkz, http://aspectwerkz.codehaus.org, último acesso: 30 de abril de 2006., 2006.

[BBD+96] C. Beierle, E. Börger, I. Durdanovic, U. Glässer, and E. Riccobene, *Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code*, Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control (J.-R. Abrial, E. Börger, and H. Langmaack, eds.), LNCS, no. 1165, Springer, 1996, pp. 62–78.

[Bör03] Egon Börger, *The asm ground model method as a foundation for requirements engineering.*, Verification: Theory and Practice, 2003, pp. 145–160.

[Bre05] Cristiano Malanga Breuel, *Pointcuts abertos*, Monografia para a disciplina mac5701, Instituto de Matemática e Estatística da Universidade de São Paulo, june 2005.

[BS03] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.

[BTI+07] Roberto Bigonha, Fabio Tirelo, Vladimir Iorio, Mariza Bigonha, and Mário Lobato, *A linguagem de especificação formal machĭna 2.0*, Tech. report, Departamento de Ciência da Computação - UFMG, February 2007.

[CGMP00] Gianpaolo Cugola, Carlo Ghezzi, Mattia Monga, and Gian Pietro Picco, *Malaj: A proposal to eliminate clashes between aspect-oriented and object-oriented programming*, Proceedings of International Conference on Software: Theory and Practice (Bejing, China), August 2000.

[Cha04] Christina Von Flach Chavez, *Um enfoque baseado em modelos para o design orientado a aspectos*, Tese de doutorado, PUC-Rio, Departamento de Informática, 2004.

[CL03] Christina Von Flach Chavez and Carlos J. P. Lucena, *A theory of aspects for aspect-oriented software development*, XVII Simpósio Brasileiro de Engenharia de Software, 2003.

[CPA05] Walter Cazzola, Sonia Pini, and Massimo Ancona, *Aop for software evolution: a design oriented approach*, SAC '05: Proceedings of the 2005 ACM symposium on Applied computing (New York, NY, USA), ACM Press, 2005, pp. 1346–1350.

[Dem06] Demeter, http://www.ccs.neu.edu/research/demeter, último acesso: 30 de abril de 2006., 2006.

[Dij76] Edsger Wybe Dijkstra, *A discipline of programming*.

[dM05] Sérgio Queiroz de Medeiros, *Utilizando aspectos no projeto de sistemas de hardware desenvolvidos com systemc*, Junho 2005.

[EAK+01] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, *Discussing aspects of aop*, Commun. ACM **44** (2001), no. 10, 33–38.

[EFB99] Tzilla Elrad, Robert E. Filman, and Atef Bader, *Aspect-oriented programming: Introduction*, Commun. ACM **44** (1999), no. 10, 29–32.

[EFB01] ———, *Aspect-oriented programming: Introduction*, Commun. ACM **44** (2001), no. 10, 29–32.

[EMO04] Michael Eichberg, Mira Mezini, and Klaus Ostermann, *Pointcuts as functional queries*, Programming Languages and Systems: Second Asian Symposium, APLAS 2004 (Taipei, Taiwan) (Wei-Ngan Chin, ed.), Lecture Notes in Computer Science, Springer-Verlag Heidelberg, November 2004, pp. 366–382.

[FF00] R. Filman and D. Friedman, *Aspect-oriented programming is quantification and obliviousness*, 2000.

[Gar04] Alessandro Fabricio Garcia, *Objetos e agentes: Uma abordagem orientada a aspectos*, Tese de doutorado, PUC-Rio, Departamento de Informática, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns*, Addison-Wesley Professional, Janeiro 1995.

[GLG03] Joseph D. Gradecki, Nicholas Lesiecki, and Joe Gradecki, *Mastering aspectj: Aspect-oriented programming in java*, John Wiley & Sons, Inc., New York, NY, USA, 2003.

[GRS05] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte, *Semantic essence of asml*, Theor. Comput. Sci. **343** (2005), no. 3, 370–412.

[Gur94] Yuri Gurevich, *Evolving algebras 1993: Lipari Guide*, Specification and Validation Methods (Egon Börger, ed.), Oxford University Press, 1994, pp. 9–37.

[HG05] Marcel Hugo and Marcio Carlos Grott, *Estudo de caso aplicando programação orientada a aspecto*, XIV Seminário de Computação - FURB, 2005.

[HH04] Erik Hilsdale and Jim Hugunin, *Advice weaving in aspectj*, AOSD 04: Proceedings of the 3rd international conference on Aspect-oriented software development (New York, NY, USA), ACM Press, 2004, pp. 26–35.

[HO93] William Harrison and Harold Ossher, *Subject-oriented programming: a critique of pure objects*, OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (New York, NY, USA), ACM Press, 1993, pp. 411–428.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, *An overview of AspectJ*, Proc. ECOOP 2001, LNCS 2072 (Berlin) (J. L. Knudsen, ed.), Springer-Verlag, June 2001, pp. 327–353.

[Kic02] Gregor Kiczales (ed.), *Proc. 1st int' conf. on aspect-oriented software development (AOSD-2002)*, ACM Press, April 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, *Aspect-Oriented Programming*, ECOOP'97 (1997).

[KO05] K. Klose and K. Ostermann, *Back to the future: Pointcuts as predicates over traces*, 2005.

[Lad03] Ramnivas Laddad, *AspectJ in action: Practical aspect-oriented programming*, Manning, 2003.

[Lem05] Otávio A. L. Lemos, *Teste de programas orientados a aspectos: uma abordagem estrutural para aspectj*, Dissertação de mestrado, USP - São Carlos, February 2005.

[Lie96] K. J. Lieberherr, *Adaptive object-oriented software: The Demeter Method with propagation patterns*, PWS Publishing Company, 1996.

[LK97] Cristina Videira Lopes and Gregor Kiczales, *D: A language framework for distributed programming*, Tech. Report SPL97-010, P9710047, Palo Alto, CA, USA, February 1997.

[Lob06] Mário Celso Candian Lobato, *Um arcabouço para compilação de linguagens de especificação asm*, Dissertação de mestrado, UFMG, Março 2006.

[Mey97] Bertrand Meyer, *Object-oriented software construction (2nd ed.)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[MK03] Hidehiko Masuhara and Gregor Kiczales, *Modeling crosscutting in aspect-oriented mechanisms.*, ECOOP, 2003, pp. 2–28.

[MKD03] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn, *A Compilation and Optimization Model for Aspect-Oriented Programs*, Lecture Notes in Computer Science (2003).

[MLWR01] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard, *Separating features in source code: an exploratory study*, ICSE '01: Proceedings of the 23rd International Conference on Software Engineering (Washington, DC, USA), IEEE Computer Society, 2001, pp. 275–284.

[NuS07] NuSMV, http://nusmv.irst.itc.it, último acesso: 24 de abril de 2007., 2007.

[OL01] Doug Orleans and Karl J. Lieberherr, *DemeterJ*, Tech. report, Northeastern University, 2001.

[OL03] ———, *DAJ: Demeter in AspectJ*, Tech. report, Northeastern University, January 2003.

[OMB05] Klaus Ostermann, Mira Mezini, and Christoph Bockisch, *Expressive pointcuts for increased modularity*, ECOOP 2005: European Conference on Object-Oriented Programming, Jule 2005.

[OT98] Harold Ossher and Peri L. Tarr, *Operation-level composition: A case in (join) point*, ECOOP '98: Workshop ion on Object-Oriented Technology (London, UK), Springer-Verlag, 1998, pp. 406–409.

[OT01a] Harold Ossher and Peri Tarr, *Hyper/J: Multi-dimensional separation of concerns for Java*, Proc. 23rd Int'l Conf. on Software Engineering, IEEE Computer Society, 2001, pp. 729–730.

[OT01b] ———, *Using multidimensional separation of concerns to (re)shape evolving software*, Commun. ACM **44** (2001), no. 10, 43–50.

[Par02] David L. Parnas, *On the criteria to be used in decomposing systems into modules*, 411–427.

[RBJ99] J. Rumbaugh, G. Booch, and I. Jacobson, *The uml reference guide*, Addison Wesley, 1999.

[RC05] Ricardo A. Ramos and Jaelson B. Castro, *Avaliação de uma metodologia de medição da qualidade em um documento de requisitos orientado a aspectos.*, WER, 2005, pp. 161–172.

[RS03] Hridesh Rajan and Kevin Sullivan, *Eos: instance-level aspects for integrated system design*, ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (New York, NY, USA), ACM Press, 2003, pp. 297–306.

[RS05a] ———, *On the expressive power of classpects*, Tech. Report CS-2005-14, Department of Computer Science, University of Virginia, 2005.

[RS05b] Hridesh Rajan and Kevin J. Sullivan, *Classpects: unifying aspect- and object-oriented language design*, ICSE '05: Proceedings of the 27th international conference on Software engineering (New York), ACM Press, 2005, pp. 59–68.

[RSMA02a] A. Rashid, P. Sawyer, A. Moreira, and J. Araújo, *Early aspects: A model for aspect-oriented requirements engineering*, Joint Int'l Conf. Requirements Engineering, IEEE, 2002, pp. 199–202.

[RSMA02b] Awais Rashid, Peter Sawyer, Ana M. D. Moreira, and João Araújo, *Early aspects: A model for aspect-oriented requirements engineerin*, RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering (Washington, DC, USA), IEEE Computer Society, 2002, pp. 199–202.

[San06] Kristian Santos, *Um arcabouço para otimizações em máquinas de estado abstratas*, Dissertação de mestrado, UFMG, Março 2006.

[Sch01] Joachim Schmid, *Introduction to asmgofeer*, Siemens Corporate Technology (2001).

[SGSP] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat, *Aspectc++: An aspect-oriented extension to the c++ programming language.*

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schréder-Preikschat, *Aspectc++: an aspect-oriented extension to the c++ programming language*, CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific (Darlinghurst, Australia, Australia), Australian Computer Society, Inc., 2002, pp. 53–60.

[SL05a] Lyrene F. Silva and Julio C. S. P. Leite, *Integração de características transversais durante a modelagem de requisitos*, SBES '05: Simpósio Brasileiro de Engenharia de Software, 2005.

[SL05b] ———, *Uma linguagem de modelagem de requisitos orientada a aspectos.*, WER, 2005, pp. 13–25.

[Som00] Ian Sommerville, *Software engineering (6th edition)*, Addison Wesley, August 2000.

[Ste07] Italo Giovani Abdanur Stefani, *Método de refinamento machina*, Dissertação de mestrado, UFMG, Março 2007.

[TBBV04] Fabio Tirelo, Roberto S. Bigonha, Mariza A. S. Bigonha, and Marco Túlio Oliveira Valente, *Desenvolvimento de Software Orientado por Aspectos*, XXIII Jornada de Atualização em Informática – JAI'04 (2004).

[TMDB99] Fabio Tirelo, Marcelo A. Maia, Vladimir O. Di Iorio, and Roberto S. Bigonha, *Máquinas de Estado Abstratas*, III Simpósio Brasileiro de Linguagens de Programação (1999).

[TO00] P. Tarr and H. Ossher, *Hyper/J user and installation manual*, Tech. report, IBM T. J. Watson Research Center, 2000.

[Wir77] Niklaus Wirth, *What can we do about the unnecessary diversity of notation for syntactic definitions?*, Commun. ACM **20** (1977), no. 11, 822–823.

[WKD04] M. Wand, G. Kiczales, and C. Dutchyn, *A semantics for advice and dynamic join points in aspect-oriented programming*, ACM Transactions on Programming Languages and Systems **26** (2004), no. 5, 890–910.