Kristian Magnani dos Santos

Um Arcabouço Para Otimizações em Máquinas de Estado Abstratas

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.

Belo Horizonte

23 de Março de 2006

Substituir pela folha de aprovação (Impressão frente-e-verso).

Substituir pela folha de aprovação (Impressão frente-e-verso).

Entrega o teu caminho ao Senhor, confia nEle e o mais Ele fará.
(Bíblia Sagrada)

Se de tudo fica um pouco,
mas por que não ficaria um pouco de mim?
(Carlos Drummond de Andrade)

Para ser grande, sê inteiro: nada
Teu exagera ou exclui.
Sê todo em cada coisa. Põe quanto és
No mínimo que fazes.
Assim em cada lago a lua toda
Brilha, porque alta vive.
(Ricardo Reis, heterônimo de Fernado Pessoa)

Valeu a pena? Tudo vale a pena
Se a alma não é pequena.
Quem quer passar além do Bojador
Tem que passar além da dor.
Deus ao mar o perigo e o abismo deu,
Mas nele é que espelhou o céu.
(Fernando Pessoa)

The Architet: *Hope. It is the quintessential human delusion, simultaneously the source of your greatest strength and your greatest weakness.*
Neo: *If I were you, I would hope that we don't meet again.*
(The Matrix Reloaded)

Agradecimentos

Nenhum trabalho é resultado exclusivo do esforço individual de uma pessoa. Ao contrário, no longo caminho deste mestrado, recebi a ajuda e o apoio de muitas pessoas, às quais externo aqui minha gratidão.

Em primeiro lugar, agradeço a Deus, que não deixou faltar a força necessária para ir adiante, ainda quando toda a minha razão tentava me convencer do contrário.

Agradeço à minha esposa, Ana Maria, uma verdadeira companheira. Suas palavras de incentivo e suas atitudes de amor foram imprescindíveis em cada momento. Seu cuidado para comigo me ajudou a chegar até aqui.

Agradeço a meus pais, Wagner e Virlene, que sempre estiveram incondicionalmente ao meu lado e me mostraram o caminho. Agradeço também à minha irmã, Simone, que me ensinou a compartilhar. Agradeço a toda a minha família, tanto do meu lado quanto por parte de Ana Maria.

Agradeço aos meus orientadores, Mariza e Roberto Bigonha, toda a orientação, compreensão e apoio. O aprendizado que recebi deles jamais será esquecido.

Agradeço ao meu amigo João Rafael, cujas consultorias em assuntos diversos foram importantíssimas para a conclusão deste trabalho. Agradeço, inclusive, o apoio no que diz respeito às questões não técnicas. Agradeço também ao Fábio Tirelo o apoio com os grafos e outras questões.

Finalmente, agradeço a todos aqueles que, de uma forma ou de outra, conscientemente ou sem se dar conta, contribuíram para a realização deste trabalho. A todos vocês, minha mais sincera gratidão.

Resumo

Máquinas de Estado Abstratas oferecem um mecanismo poderoso e de fácil utilização para a especificação formal da semântica de algoritmos. O arcabouço kℓar incrementa esta metodologia com a capacidade de otimização, permitindo que especificações ASM sejam traduzidas em programas *eficientes*, característica importante de programas a serem utilizados comercialmente. Mais ainda, as otimizações neste arcabouço são módulos independentes que podem ser adicionados com o arcabouço em pleno funcionamento, de modo que desenvolvedores independentes possam desenvolver suas otimizações sem se preocupar com detalhes internos do kℓar. Finalmente, o grande conjunto de construções da linguagem utilizada pelo kℓar permite o seu uso como alvo por compiladores de linguagens ASM.

Abstract

The Abstract State Machines methodology offers a powerful, easy-to-use mechanism to formally specify the semantics of algorithms. The kℓar framework adds to it optimization capability, allowing the transformation of ASM specifications into efficient programs, which is important in order to use the specifications as realistic programs. Moreover, the optimizations are modules to be plugged-in "on the fly", so that independent developers can build their own optimizations without concerning about the internal details of the kℓar framework. Finally, the wide set of constructions of the language understood by the framework allows its use as a target for compilers aiming the ASM methodology.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Métodos formais de definição semântica servem a diversos propósitos. Dentre eles, pode-se destacar a descrição precisa de conceitos de forma independente de máquina, a não-ambigüidade de uma especificação e o fato de oferecer um ambiente para prova de teoremas e dedução de propriedades [Gor79].

Segundo Nielson [NN99], semântica formal diz respeito à especificação rigorosa do significado ou comportamento de programas, sistemas, arquiteturas etc. A necessidade do rigor surge particularmente porque:

- ele pode revelar ambigüidades e complexidades sutis em uma definição aparentemente clara;
- este mesmo rigor pode servir como base para implementação, análise e verificação, especialmente a prova de correção de programas.

Diversos são os métodos formais, bem como diversas são as suas finalidades e seus pontos fortes e fracos. Destacam-se aqui três abordagens, a saber:

Semântica Operacional Em semântica operacional, o significado de uma construção é especificada pelo efeito de sua execução em uma *máquina virtual*. Em particular, o maior interesse é definir *como* o efeito de uma computação é produzido.

Semântica Denotacional Nesta abordagem, os significados são modelados por um conjunto de mapeamentos dos construtos da linguagem em elementos matemáticos abstratos, chamados de *denotações*. A semântica de um construto é uma função de seus constituintes. Assim, apenas o *efeito* da execução é de interesse, e não a maneira como este efeito é alcançado.

Semântica Axiomática Propriedades específicas, ou *axiomas*, do efeito da execução de construções são expressas por meio de *asserções*. Desta forma, aspectos gerais da execução são ignorados.

O uso de semântica formal na especificação de linguagens de programação não é apenas desejável, é também necessário, porque apenas uma definição formal pode ser livre de ambigüidades [NN99]. O estudo de propriedades de um programa deve partir de sua definição

semântica formalizada, que fornece o rigor e a precisão necessários para a prova de asserções a respeito do programa.

Embora mais custoso em tempo e esforço, as vantagens fornecidas por uma especificação formal justificam sua adoção. Em particular, todas as áreas cuja segurança de pessoas e outros valores dependam de programas são clientes potenciais de métodos formais.

Entre os diversos métodos formais de definição semântica, um método bastante interessante é denominado *Máquinas de Estado Abstratas* (ASM, do inglês *Abstract State Machines*), introduzidas por Yuri Gurevich [Gur95, Gur91]. Trata-se de um conceito poderoso e elegante para a modelagem matemática de sistemas dinâmicos discretos, baseado em semântica operacional. Basicamente, define-se uma máquina de estados que simula a execução passo a passo de um algoritmo. Conceitos simples e bem conhecidos são utilizados na definição desta máquina, o que simplifica a tarefa de especificação de um sistema por meio desta metodologia.

Informalmente, uma ASM é uma máquina abstrata onde os estados são *álgebras*, ou seja, um conjunto, chamado de *superuniverso*, juntamente com funções e relações sobre elementos deste conjunto [HW02]. Ao conjunto de nomes de funções e relações de um estado dá-se o nome de *vocabulário* de um estado. Existe também uma função de interpretação, que mapeia nomes do vocabulário em suas respectivas funções e relações. A mudança de estado de uma ASM acontece por meio da execução de sua *regra de transição*, que modifica a interpretação de alguns nomes do vocabulário do estado.

Uma regra de transição é semelhante a um programa escrito em uma linguagem imperativa, com a diferença que não existem comandos de iteração. Desta forma, o comportamento iterativo é obtido pela característica inerente a uma ASM, que é a sua execução cíclica. Mais exatamente, dado um estado inicial, a regra de transição é aplicada neste estado gerando um novo estado. Neste novo estado, a regra de transição é executada novamente, gerando um novo estado. Este processo se repete indefinidamente. Enquanto o vocabulário permanece imutável ao longo da execução, a interpretação dos nomes do vocabulário é modificada de estado para estado.

As regras básicas que formam uma regra de transição são a regra condicional, a regra de atualização e a regra bloco. A regra condicional é da forma

$$\textbf{if } g \textbf{ then } R_0 \textbf{ else } R_1$$

e seu efeito é o de executar as regras R_0 ou R_1 de acordo com o valor resultante da avaliação da guarda g.

Uma regra de atualização altera uma função dinâmica em um ponto específico. Uma função dinâmica é uma função cujos valores nos pontos de seu domínio podem ser alterados ao longo da execução. Regras de atualização têm a forma

$$f(\overline{x}) \ := \ y$$

e seu efeito é transformar f em uma função tal como antes, à exceção que o valor de sua avaliação no ponto $\overline{x}$ agora resulta em y. Outra peculiaridade do modelo ASM é que a atualização de uma função dinâmica só é perceptível no próximo passo da execução. Ou seja, se $f(\overline{x})$ for avaliada em outras sub-regras no mesmo estado, no mesmo passo de execução, então o valor resultante não será necessariamente y, mas sim o seu valor anterior à atualização.

Finalmente, uma regra bloco tem a forma

$$R_0, \ldots, R_n$$

e seu efeito é executar cada uma das regras $R_0, \ldots, R_n$ em paralelo.

A metodologia de máquinas de estado abstratas apresentam características importantes que permitem o seu uso na especificação formal de algoritmos. Dentre estas características, destacam-se:

Precisão Se a semântica da metodologia de especificação é ambígua, então a especificação em si também o será [Gor79]. Para a descrição de uma especificação ASM são utilizados estruturas e conceitos clássicos da matemática, o que garante a precisão e a clareza necessárias.

Demonstração de Correção da Especificação A prova de asserções a respeito de uma especificação ASM é facilitada por características presentes nesta metodologia. Particularmente, a transição entre os estados é de fácil entendimento, pois não existe a noção de fluxo de controle como nas linguagens de programação convencionais. Além disso, a execução de um conjunto de regras não produz efeitos colaterais, o que facilita na determinação do conjunto de atualizações a cada transição. Por efeitos colaterais entende-se a possibilidade da execução de uma sub-regra atualizar funções dinâmicas em pontos consultados por outras regras, o que não é o caso do modelo, pois mesmo que isto aconteça, estas atualizações só são perceptíveis no próximo estado.

Generalidade A metodologia ASM já foi usada com sucesso em uma ampla gama de domínios de aplicação, dentre eles:

- linguagens de programação [BR94, GH93, KP97, Wal97];
- sistemas seqüenciais [BM97, Str97];
- sistemas paralelos e distribuídos [BS97, BGR95];
- sistemas de tempo real [GK97, GM95].

A bibliografia comentada de ASM [BH98] apresenta muitos outros exemplos de aplicações de ASM em diversos tipos de sistemas dinâmicos discretos. Esta característica do modelo contrasta com o fato de que muitos modelos de especificação são úteis em apenas algum escopo limitado.

Facilidade de Aprendizado A sintaxe de uma especificação ASM é semelhante em muitos aspectos à sintaxe de uma linguagem de programação. Mais do que isso, seus conceitos são herdados de uma matemática simples e bem conhecida. Isto torna o seu uso adequado até mesmo para aqueles não iniciados na área de métodos formais.

Facilidade de Leitura e Escrita Um ponto a favor de uma especificação formal ASM é o fato desta ser de fácil leitura. Caso contrário, a sua utilização seria limitada. Especificações ASM possuem uma sintaxe bastante simples, o que torna a sua leitura fácil e direta. Qualquer programador médio é capaz de escrever uma especificação ASM após uma rápida introdução ao método.

Escalabilidade O uso de ASM permite que a complexidade inerente de um sistema seja abordada em diversos níveis de abstração. Isto é possível por meio da construção de hierarquias de níveis de sistemas, de forma que se pode, a cada momento, examinar certas características de um sistema, ignorando-se detalhes que não são de interesse naquele momento.

Possibilidade de Execução Uma maneira de se testar a correção de uma especificação é por meio de sua execução. Algumas vantagens desta abordagem são:

- possibilidade de experimentar a especificação a cada instante, reparando erros que não foram detectados inicialmente;
- parametrização de condições do ambiente externo, de modo a testar a especificação em diversos cenários, inclusive cenários que imitem situações de falhas;
- possibilidade de combinar teste com verificação formal em um mesmo momento.

Del Castillo divide as ferramentas ASM potencialmente úteis em duas categorias [Del99]:

1. Ferramentas que dão suporte ao usuário durante o processo de desenvolvimento de especificações ASM, como editores, analisadores estáticos, interpretadores e depuradores simbólicos. Essas ferramentas auxiliam o autor de uma especificação ASM a formalizar descrições do sistema e melhoram essas formalizações como o resultado de um processo iterativo.

2. Ferramentas que transformam especificações ASM traduzindo-as em outras linguagens, de forma a permitir o processamento dessas especificações por outras ferramentas, ou seja, geradores de código que transformam especificações ASM em programas em uma dada linguagem ou lógica. Ferramentas desse tipo tornam possíveis o uso de compiladores que produzem código eficiente ou ferramentas que permitem a verificação de propriedades da especificação.

Apesar do esforço de muitos pesquisadores e das melhoras obtidas ao longo dos últimos anos, o atual estado da arte de ferramentas de suporte para especificações ASM encontra-se ainda longe de ser completamente satisfatório [Del99]. De um lado, existem vários simuladores ASM, a maioria baseada em interpretação, que, apesar da possibilidade de executar a especificação ASM, não provêem nenhuma outra característica interessante. As possibilidades de interação são em geral poucas, o que não as torna muito convenientes como ferramentas de desenvolvimento. Mais ainda, muitas delas se baseiam na linguagem de implementação para definir parte dessa implementação, por exemplo, as funções estáticas. Por outro lado, quase não existem ferramentas de transformação.

Nesse contexto, foi desenvolvido o kℓar[1], o qual é apresentado neste trabalho. kℓar é um arcabouço (*framework*) para a geração de código C++ otimizado a partir de uma especificação ASM. É importante ressaltar que as otimizações referidas são otimizações específicas do modelo de máquinas de estados abstratas, as quais não se superpõem com as otimizações convencionais. Estas últimas são deixadas a cargo do compilador que compila o código C++ gerado. Algumas das possíveis otimizações são apresentadas na Seção 2.5, e a implementação de uma destas otimizações é relatada no Capítulo 7.

[1] Em alemão, o adjetivo *klar* possui o sentido denotativo de "claro", "evidente", "nítido", mas conotativamente pode ser interpretado também como "inteligente", "esperto", "perspicaz", "inconfundível". (Langenscheidts Großwörterbuch, 4. Auflage, 2000)

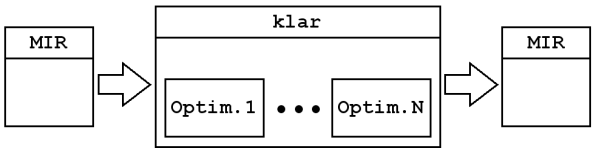

Figura 1.1: O processo de otimização de uma especificação ASM por meio do kℓar.

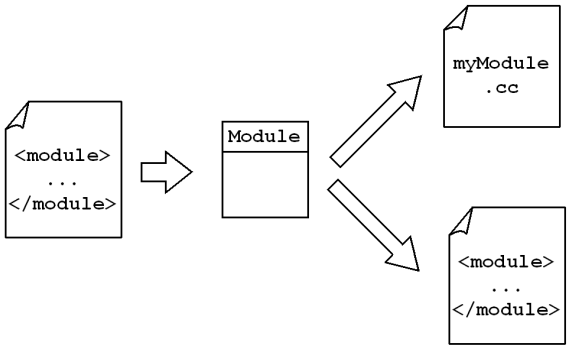

Figura 1.2: Geração de código C++ a partir de MIR e a conversão bidirecional em XML.

1.2 Trabalho Proposto

1.2.1 Descrição do Trabalho

kℓar é um *arcabouço otimizador*. Como entrada, esse arcabouço recebe uma especificação ASM em um formato conhecido como *Machĭna Intermediate Representation*, abreviadamente MIR [Tir00, Oli04]. A linguagem para descrição de especificações neste formato é apresentada no Capítulo 4. 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 em separado [Oli04], e então acopladas ao kℓar (vide Seção 1.2.2). Esse acoplamento é dinâmico e parametrizado, de modo que a tarefa de desenvolver novas otimizações ou modificar otimizações existentes não implica alterar o código fonte do arcabouço e recompilá-lo. O processo é ilustrado na Figura 1.1.

Neste trabalho, a linguagem para descrição de estruturas MIR é estendida, e uma coleção de classes é implementada de modo a dar suporte à utilização desta linguagem. 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, definido no Apêndice A. 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.

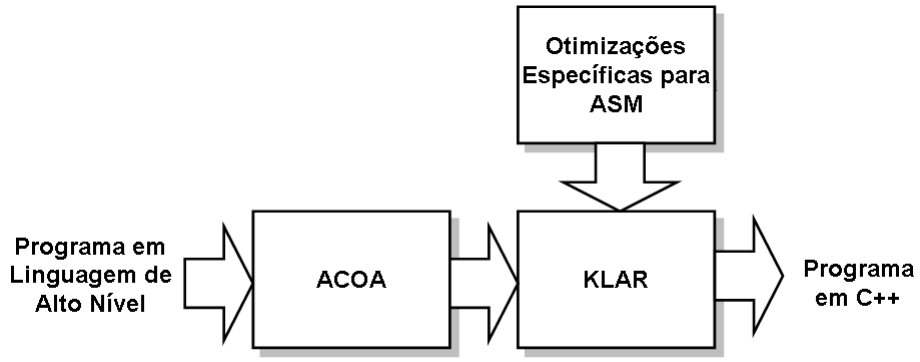

Figura 1.3: A integração dos projetos.

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 apresentado por Stroustrup [Str00], 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.

Estas características são ilustradas na Figura 1.2. Detalhes sobre a arquitetura de MIR podem ser encontrados em [OBB04a].

1.2.2 Contexto

O projeto do kℓar não é um projeto isolado. Ao contrário, o kℓar está inserido em um projeto maior, desenvolvido no Laboratório de Linguagens de Programação do Departamento de Ciência da Computação da UFMG. A Figura 1.3 apresenta este projeto, bem como a contextualização do kℓar.

O *Arcabouço para Compilação Orientada por Aspectos*, ou ACOA, é desenvolvido por Lobato como um trabalho de mestrado [Lob05], e consiste em um arcabouço que permite a geração de compiladores utilizando-se técnicas de orientação por aspectos. O kℓar é utilizado como *back-end* para os compiladores utilizados por este arcabouço.

As pesquisas sobre otimizações específicas do modelo ASM estão sendo desenvolvidas como trabalho de doutorado de Oliveira [Oli04]. Estas otimizações serão aclopadas ao kℓar e farão uso da estrutura oferecida por este arcabouço para o desenvolvimanto de otimizações.

1.2.3 Metodologia

kℓar é implementado como uma biblioteca de vínculo dinâmico em C++, conforme detalhado no Capítulo 6, e seguindo a definição da linguagem apresentada em [Str00]. No projeto e implementação do kℓar são observados aspectos importantes para garantir sua reusabilidade, tais como:

Estruturação como Arcabouço Do ponto de vista das otimizações, kℓar é estruturado como um arcabouço. A principal característica de um arcabouço é a inversão de controle: o código desenvolvido pelo cliente do arcabouço é invocado pelo arcabouço, em

contraste com uma biblioteca, onde o código do cliente é quem invoca o código exposto pela biblioteca.

Padrões de Projeto Sucintamente, um *padrão de projeto* é uma solução para um problema recorrente em desenvolvimento de programas. A Seção 3.3 esclarece em detalhes este conceito e introduz os padrões empregados no kℓar, retirados do trabalho seminal de Gamma *et alii* [GHJV95].

Técnicas Eficientes de Codificação Meyers [Mey98, Mey96] apresenta técnicas e decisões para a escrita de código eficiente na linguagem C++. Sempre que aplicáveis, estas técnicas foram adotadas no desenvolvimento e implementação do kℓar.

1.2.4 Validação do Trabalho

Como validação do kℓar, é implementada uma otimização, descrita no Capítulo 7. Esta otimização não se superpõe às otimizações convencionais, aplicadas ao código C gerado a partir de uma estrutura MIR, conforme evidenciado no Capítulo 7, que reúne os resultados de diversos testes realizados com a ferramenta.

1.2.5 Principais Contribuições

As principais contribuições deste trabalho são apresentadas a seguir:

Extensão da linguagem intermediária MIR, definição de uma representação XML de programas MIR e implementação da estrutura subjacente A linguagem intermediária MIR é estendida neste trabalho, permitindo a descrição de especificações ASM no formato da arquitetura MIR. Um conjunto de classes foi implementado de modo a prover uma representação em memória de um programa escrito em tal linguagem, e também de fornecer funcionalidades como carregamento de uma especificação a partir de sua descrição na linguagem proposta, geração desta mesma descrição a partir de uma representação em memória e a geração de código C++. Com isso, esta implementação consiste em si mesma em um *back-end* para compiladores de linguagens baseadas em ASM.

Projeto e implementação do arcabouço kℓar O arcabouço kℓar é a principal contribuição deste trabalho, e consiste em um arcabouço para otimizações em programas escritos na linguagem para representação intermediária MIR, a qual segue a metodologia de ASM. As otimizações podem facilmente serem acrescentadas ou removidas do arcabouço, pois o projeto deste é totalmente modular, o que facilita o desenvolvimento de novas otimizações. Este arcabouço usa uma estrutura de classes associada à linguagem proposta, e juntamente com esta estrutura, forma tanto um *back-end otimizador* para compiladores de linguagens baseadas em ASM, assim como um arcabouço para pesquisa e desenvolvimento de otimizações para a execução de especificações ASM.

Validação do arcabouço kℓar por meio da implementação de uma das otimizações ASM Além de prover o arcabouço kℓar, este trabalho também realiza a sua validação por meio da implementação de uma das otimizações ASM propostas. Testes empíricos são realizados com o intuito de comprovar a eficiência da otimização implementada e a sua não-superposição com as otimizações convencionais de linguagens imperativas. Para assegurar que

as construções sintáticas ASM foram devidamente implementadas, são apresentados pequenos programas de teste que cobrem todas as suas construções.

1.3 Organização do Texto

O texto desta dissertação está dividido 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. Os objetivos da pesquisa são citados, e é proposta a solução para o problema identificado.

Máquinas de Estado Abstratas Este capítulo apresenta, de forma resumida, a literatura básica a respeito dos temas necessários à compreensão do trabalho proposto, bem como os fundamentos teóricos do contexto no qual o trabalho se insere.

Projeto e Uso de Arcabouços Neste capítulo são feitas considerações pertinentes a respeito do projeto de arcabouços de software, bem como são consideradas as implicações de seu uso. No final são apresentados os padrões de projeto presentes na implementação do kℓar.

Linguagem Intermediária para Representação de ASM Multi-Agentes A linguagem utilizada para descrever as especificações ASM a serem otimizadas é apresentada neste capítulo. A sintaxe e a semântica são apresentadas.

Implementação da Linguagem MIR São apresentadas as classes da implementação das funcionalidades pertinentes à linguagem MIR, assim como é detalhado o seu mapeamento em C++.

Arquitetura do kℓar A arquitetura global do kℓar é apresentada. Além de um registro explicativo, este capítulo serve também como documentação inicial para futuros aperfeiçoamentos que venham a ser propostos. O protocolo para criação e acoplamento de otimizações é apresentado, bem como a utilização do kℓar.

Validação do kℓar e Avaliação dos Resultados Como argumento na validação do kℓar, uma otimização foi implementada, e é descrita neste capítulo. O resultado da otimização proposta é apresentado e seus efeitos sobre o código final são discutidos. A superposição entre as otimizações ASM e as otimizações convencionais do compilador C é empiricamente mostrada como não existente.

Conclusões Conclusões gerais sobre o trabalho são feitas neste capítulo final, bem como são apontados caminhos futuros que podem ser trilhados a partir dos resultados alcançados.

O texto da dissertação inclui também os seguintes apêndices:

O Arquivo de Serialização MIR O padrão do arquivo de serialização MIR é definido neste apêndice. Trata-se de um padrão *XML-like*, e por isso esta definição é feita por meio de um XSD *(XML Schema Definition)*.

Exemplos Os exemplos utilizados para a validação do kℓar são apresentados neste apêndice. Estes exemplos foram construídos de modo a cobrir todas as marcas XML da linguagem.

Capítulo 2

Máquinas de Estado Abstratas

Máquinas de Estado Abstratas (ASM), originalmente chamadas de *Álgebras Evolutivas*, foram propostas por Yuri Gurevich com o objetivo de fornecer uma semântica operacional para algoritmos por meio da elaboração da tese implícita de Turing, que diz que todo algoritmo é simulado por uma máquina de Turing apropriada [Gur95, Gur91] [1]. Entretanto, cada passo de um algoritmo a ser simulado pode necessitar de um número exponencial de passos na máquina de Turing, tornando esta simulação difícil em termos práticos. Isto é particularmente verdadeiro quando faz-se necessária a simulação de interação com o ambiente. Neste contexto, as ASM surgiram com o objetivo de simular algoritmos de uma forma mais próxima do nível do problema, de tal forma que a máquina precise de um número limitado de passos para simular um passo do algoritmo. A tese proposta por Gurevich diz que *"todo algoritmo seqüencial pode ser simulado passo a passo por uma ASM seqüencial apropriada"* [Gur00].

A utilização de ASM para a especificação formal de semântica de linguagens de programação apresenta algumas vantagens importantes. Uma destas vantagens consiste no fato de que as ASM possuem recursos para simular interações com o ambiente de forma eficiente. Além disso, uma especificação ASM é de fácil leitura, sendo compreensível até mesmo por não-iniciados em métodos formais de especificação semântica.

Outra vantagem apresentada pela metodologia de ASM é a facilidade de se provar propriedades a respeito de uma especificação ASM construída para um dado propósito. Esta facilidade se deve principalmente à possibilidade de se ajustar diferentes partes da especificação ASM para diferentes níveis de abstração. Como existem ambientes de execução ASM, uma especificação ASM também pode ser executada. Como exemplos de tais ambientes, cabe destacar o ASM Workbench [Del00] e a linguagem Machĭna [BTM$^+$99, TMIB99b, Tir00], esta última abordada na Seção 2.4.

Informalmente, a execução de uma especificação ASM se inicia em um estado S_0 e continua através dos estados S_1, S_2, etc., de tal forma que apenas uma parte do trabalho é realizada em cada passo. Cada estado pode ser representado por uma estrutura de primeira ordem, que nada mais é do que um conjunto com funções e relações sobre os elementos deste conjunto. A execução de uma especificação pode ser vista como uma sucessão de estruturas de primeira ordem. Um estado S_{i+1} é atingido fazendo-se um número finito de modificações no estado S_i.

[1] Segundo Gurevich [Gur00], a tese "oficial" de Turing diz que toda função computável é computada por uma máquina de Turing apropriada. Entretanto, o argumento informal apresentado por Turing em favor de sua tese justifica uma tese maior, a saber, a de que todo algoritmo é simulado por uma máquina de Turing apropriada

Estas modificações são determinadas pela regra de transição da especificação.

A seguir, faz-se a definição formal de uma especificação ASM e sua execução. A Seção 2.1 introduz os conceitos básicos necessários para a definição formal de uma ASM, a qual é feita na Seção 2.2. A Seção 2.3 aborda a questão das ASM Multi-Agentes. Esta revisão é particularmente baseada em [Gur95, DDG96, TMIB99a, Tir00].

2.1 Definições Preliminares

Para que seja possível definir uma álgebra evolutiva, fazem-se necessárias algumas definições preliminares, as quais são feitas nesta seção. Estas definições são utilizadas na Seção 2.2 para a definição formal de uma ASM.

Definição 2.1. (Vocabulário) Um *vocabulário* (ou *assinatura*) Υ é uma coleção finita de nomes de funções, onde cada um destes nomes possui uma aridade fixa. Alguns nomes podem ser marcados *nomes de relação*, ou como *nomes estáticos*, ou ainda como ambos.

□

Com o objetivo de facilitar o uso deste conceito na definição de álgebras, assume-se que todo vocabulário Υ possui os seguintes nomes estáticos: o sinal de igualdade; os nomes de função $true$, $false$ e $undef$, com aridade nula; e também os nomes das operações booleanas usuais.

As funções em Υ podem ser de três tipos mutuamente exclusivos, a saber, *estáticas*, *derivadas* ou *dinâmicas*. Funções dinâmicas podem ser alvo de uma atualização, ao passo que funções estáticas permanecem inalteráveis através dos estados. Assume-se que toda função básica é dinâmica, a menos que seu nome seja marcado como estático. Uma relação é uma função cujo domínio seja $\{true, false\}$. Assume-se que uma função não é uma relação, a menos que esta seja marcada como tal. Uma função também pode ser *externa*, o que significa que sua implementação não é conhecida *a priori*. Estas funções servem para se modelar a interação com o ambiente eficientemente. Uma função derivada não pode sofrer atualizações, mas pode utilizar em sua definição funções dinâmicas ou externas, de modo que, ao longo da execução, funções derivadas podem sofrer alterações indiretas.

Definição 2.2. (Estado) Uma *álgebra estática* (ou simplesmente um *estado*) S de vocabulário Υ é um conjunto $X \neq \emptyset$, também chamado de *super-universo* de S, junto com as interpretações dos nomes de Υ em X. Uma *função básica* de S é um nome de função r-ária interpretado como uma função $f : X^r \rightarrow X$, enquanto uma *relação básica* de S é um nome de função r-ária interpretado como uma função $f : X^r \rightarrow \{true, false\}$. O vocabulário de S, Υ, é denotado por $Fun(S)$.

□

Toda função parcial f pode ser vista como sendo total, arbitrando-se que $f(x) = undef$ para os valores de x nos quais f seria indefinida.

Definição 2.3. (Universo) Uma relação básica f pode ser vista como o conjunto de todas as tuplas $\overline{x}$, tais que $f(\overline{x}) = true$. A notação $\overline{x} \in f$ é equivalente à notação $f(\overline{x})$. Se f é unária, a relação f é dita ser um *universo*.

□

Como regra geral, *undef* não é incluído em universos.

Definição 2.4. (Termos) Os *termos* podem ser definidos recursivamente da seguinte forma:

1. Uma variável é um termo.
2. Se f é um nome de função r-ária, $r \geq 0$, e se $t_1, \ldots, t_r$ são termos, então $f(t_1, \ldots, t_r)$ é um termo.

□

Uma função 0-ária $f()$ pode ser escrita simplesmente como f, omitindo-se os parênteses, sem alteração de significado.

Definição 2.5. (Estados Apropriados) Um estado S é dito apropriado para um objeto sintático s se $Fun(S)$ inclui a coleção de nomes que ocorrem em s. Esta coleção é denotada por $Fun(s)$.

□

Em um estado apropriado S, um termo sem variáveis $t = f(t_1, \ldots, t_r)$ é avaliado pela função Val_S de tal forma que $Val_S(t) = Val_S(f)(Val_S(t_1), \ldots, Val_S(t_r))$.

Definição 2.6. (Redução e Expansão) A *redução* de um Υ-estado S para um vocabulário menor Υ' é o Υ'-estado S' obtido a partir de S removendo-se as interpretações dos nomes de funções em $\Upsilon - \Upsilon'$. Por outro lado, S é dito ser a *expansão* de S' em Υ.

□

Definição 2.7. (Portador) Um *portador* é um estado cujo vocabulário contém apenas nomes *estáticos* de função. O portador $|S|$ de um estado S é a redução de S à parte estática de $Fun(S)$.

□

Definição 2.8. (Endereço) Um *endereço* sobre um portador C é um par $\ell = (f, \overline{x})$, onde $f \notin Fun(C)$, ou seja, f é o nome de uma função dinâmica, e $\overline{x} = x_1 \ldots x_r$ tal que a aridade de f é igual a r. Define-se $Loc_\Upsilon(C)$ como $Loc_\Upsilon(C) = \{\ell = (f, \overline{x}) \mid f \in \Upsilon\}$.

□

Se um estado S é apropriado para um termo básico $t_0 = f(\overline{t})$, então o endereço de t_0 em S é o endereço $(f, Val_S(\overline{t}))$.

Definição 2.9. (Atualização) Uma *atualização* de um estado S é um par $\alpha = (\ell, y)$ tal que ℓ é um endereço de S e $y \in |S|$. O endereço ℓ é o endereço $Loc(\alpha)$ de α, e y é o valor $\mathrm{Val}(\alpha)$ de α.

□

Definição 2.10. (Disparo de uma Atualização) Para disparar uma atualização $\alpha = (\ell, y)$ em um estado S, deve-se redefinir S de tal forma que o endereço ℓ mapeie para y. Assim, seja $S(\ell) = Val_S(f)(Val_S(\overline{x}))$. O resultado do disparo da atualização $\alpha = (\ell, y)$ em um estado S é um novo estado S' tal que $Fun(S') = Fun(S)$, $|S'| = |S|$, $S'(\ell) = y$ e $S'(\ell') = S(\ell'), \forall \ell' \neq \ell$.

□

Definição 2.11. (Conjunto de Atualizações) Um *Conjunto de Atualizações* β sobre um estado S é um conjunto composto pelas atualizações α de S. $Loc(\beta)$ do conjunto β é definida como $Loc(\beta) = \{Loc(\alpha) | \alpha \in \beta\}$.

□

Definição 2.12. (Consistência de um Conjunto de Atualizações) Um conjunto de atualizações β é dito *consistente* se e somente se

$$((f, \overline{x}), y_1) \in \beta \wedge ((f, \overline{x}), y_2) \in \beta \Longrightarrow y_1 = y_2.$$

ou seja, duas atualizações de um mesmo endereço atribuem a este endereço um mesmo valor.

□

Definição 2.13. (Disparo de um Conjunto de Atualizações) Seja β um conjunto *consistente* de atualizações em um estado S. Desta forma, o *disparo de β em S* é o disparo *simultâneo* de todas as atualizações $\alpha \in \beta$. O resultado é um novo estado S', cujo vocabulário e cujo portador são os mesmos daqueles em S, de tal forma que:

$$Val_{S'}(f(\overline{x})) = \begin{cases} y & \text{se } ((f, \overline{x}), y) \in \beta \\ Val_S(f(\overline{x})) & \text{caso contrário} \end{cases}$$

□

Caso o conjunto de atualizações β seja inconsistente, várias abordagens podem ser utilizadas. Um delas consiste em definir que o disparo de β em S resulta em um estado S' tal que $S = S'$. Outra abordagem consiste em escolher não-deterministicamente entre as atualizações conflitantes qual será a atualização que terá efeito. Neste texto, optou-se pela primeira abordagem, de tal forma que a máquina pode ser considerada *determinista*. Abordagens como a segunda, onde uma escolha é feita não-deterministicamente, estabelecem um modelo *não-determinista*.

Definição 2.14. (Regras) *Regras* são definidas recursivamente da seguinte forma:

1. Seja f um nome de função (ou relação) dinâmica, e sejam t e t_0 termos. Então $R \equiv f(\overline{t}) := t_0$ é uma regra, também chamada de regra de atualização. Se f é um nome de relação, então $t_0 \in \{true, false\}$. Define-se o conjunto de atualizações de R em um estado S, $Updates(R, S)$, como o conjunto $\{(\ell, y)\}$, tal que $\ell = (f, Val_S(\overline{t}))$ e $y = Val_S(t_0)$.

2. Sejam $R_1, \ldots, R_k$ regras. Então o bloco de regras $R \equiv R_1, \ldots, R_k$ é uma regra, e seu conjunto de atualizações em um estado S é dado por:

$$Updates(R, S) = \bigcup_{i=1}^{k} Updates(R_i, S).$$

   Isto equivale a dizer que a ordem da regras $R_1, \ldots, R_k$ não é importante, pois todas são disparadas simultaneamente.

3. Seja k natural, $g_0, \ldots, g_k \in \{true, false\}$ e $R_0, \ldots, R_k$ regras. Então

$$R \equiv \begin{array}{l} \textbf{if } g_0 \textbf{ then } R_0 \\ \textbf{elsif } g_1 \textbf{ then } R_1 \\ \ldots \\ \textbf{elsif } g_k \textbf{ then } R_k \\ \textbf{endif} \end{array}$$

é uma regra, cujo conjunto de atualizações é dado por:

$$Updates(R, S) = \begin{cases} Updates(R_i, S) & \text{se } \exists i \in \{0..k\} | g_i \wedge \neg g_j, \forall j < i \\ \emptyset & \text{se } \neg g_i, \forall i \in \{0..k\} \end{cases}$$

Estas regras são conhecidas como *regras básicas*.

□

Definição 2.15. (Regras Estendidas) Além das regras básicas apresentadas acima, são definidas algumas regras que estendem o modelo.

1. Uma regra de importação **import** v R_0 **endimport** tem o efeito de executar a regra R_0 em um ambiente onde v é uma variável.

2. Uma regra de escolha **choose** v **in** U **satisfying** g R_0 **endchoose** escolhe um elemento de U que satisfaça a condição g e então associa o elemento escolhido a v, executando a regra R_0.

3. Uma regra **forall** v **in** U c **endforall** associa v a cada elemento de U e paralelamente executa R_0 com cada associação feita.

□

O comportamento não-convencional de se interpretar um bloco de regras como um *conjunto* de regras no lugar de uma *seqüência* de regras possui algumas vantagens que justificam esta escolha. A primeira delas é que esta escolha permite uma certa concorrência. Esta concorrência se mostra útil para se obter uma melhor simulação de algoritmos. Outra vantagem é que torna-se relativamente fácil a extensão do modelo para seu uso em computação assíncrona distribuída. Estas vantagens compensam o fato de que em uma abordagem de execução das regras em seqüência não existiria o problema de atualizações conflitantes. Além disso, o uso de guardas garante o seqüenciamento de ações na abordagem de conjunto.

Definição 2.16. (Disparo de uma Regra) O *disparo de uma regra* R em um estado S é o disparo do conjunto de atualizações $Updates(R, S)$ em S. O estado resultante deste disparo, S', é também definido como $Fire(R, S)$.

□

2.2 Definição Formal de Máquinas de Estado Abstratas

Segundo Del Castillo [DDG96], ASM são sistemas de transição que especificam uma computação cujos estados são álgebras. Os universos de álgebras que formam os estados da computação constituem o superuniverso da ASM.

Definição 2.17. (Especificação ASM) Uma *especificação ASM* é uma quádrupla $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$, tal que:

- $\Upsilon = \Upsilon_0 \cup \Upsilon_e$ é um vocabulário, composto pela união do vocabulário pré-definido Υ_0 com o vocabulário Υ_e definido na especificação. O vocabulário pré-definido Υ_0 contém os nomes de relação *Boolean*, *Integer*, *String*, *List* e *Set*, interpretados, respectivamente, como os universos dos valores lógicos, dos números inteiros, das cadeias de caracteres, das listas e dos conjuntos. Υ_0 também contém as operações usuais sobre estes conjuntos.
- $\mathcal{A}$ é um conjunto de Υ-álgebras ou estados S, cada um consistindo no vocabulário Υ e um conjunto X (o superuniverso de S) de funções, que são as interpretações em X dos nomes de funções de zero argumentos pertencentes a Υ. Um nome de função de aridade $r > 0$ é interpretado como uma função de $X^r \rightarrow X$. O conjunto X é comum a todos os estados pertencentes a $\mathcal{A}$.
- $S_0 \in \mathcal{A}$ representa o estado inicial, onde a interpretação de alguns nomes de funções são dados. Nomes de funções cujas interpretações não são dadas em S_0 são interpretados como $undef$.
- $\mathcal{P}$ é uma regra que descreve as modificações das interpretações de nomes de funções de uma álgebra (estado) para outra.

□

Definição 2.18. (Execução de uma Especificação ASM) Seja $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$ uma especificação ASM. Então define-se a *execução* desta especificação como a seqüência $\mathbb{S} = \langle S_n | n \geq 0 \rangle$, tal que cada S_n é um estado pertencente a $\mathcal{A}$, e $S_{n+1} = Fire(\mathcal{P}, S_n)$.

□

2.3 ASM Multi-Agentes

A extensão do modelo ASM apresentado para sua versão com múltiplos agentes é bastante natural, e é definida nesta seção, à maneira como feita por Tirelo [Tir00].

Definição 2.19. (ASM Multiagente) Uma *ASM Multiagente* $\mathcal{M}$, ou *ASM Distribuída*, é uma quádrupla $(\Upsilon_{\mathcal{M}}, \mathcal{A}, \mathcal{C}_0, \mathcal{P})$, tal que:

- $\Upsilon_{\mathcal{M}}$ é o vocabulário de $\mathcal{M}$. Pertencem a $\Upsilon_{\mathcal{M}}$ os nomes de função que aparecem nas regras de $\mathcal{P}$, à exceção da função *Self*. Também pertencem a $\Upsilon_{\mathcal{M}}$ os nomes de função de zero argumento que representam os nomes dos módulos, a saber, os elementos de $\mathcal{P}$, assim como uma função unária chamada *Mod*. O papel desta função *Mod* é representar a relação entre agentes e módulos da seguinte forma: um elemento a é um agente em um dado estado S se e somente se houver um nome de módulo μ tal que $S \models Mod(a) = \mu$. Desta forma, o programa de a é $Prog(a) = \pi_\mu$.

- $\mathcal{A}$ é um conjunto de estados.
- $\mathcal{C}_0$ é uma coleção de estados denominados *estados iniciais* de $\mathcal{M}$. Todos os elementos de $\mathcal{C}_0$ compartilham as mesmas interpretações dos nomes de função de $\Upsilon_{\mathcal{M}}$, à exceção de *Self*, que é interpretado como o agente associado ao estado.
- $\mathcal{P}$ é um conjunto finito indexado de programas π_μ (os módulos), identificados pelos nomes de módulos $\mu \in \Upsilon$. Cada Programa pertencente a $\mathcal{P}$ é uma regra.

□

Em uma ASM Multiagente, um número finito de agentes computacionais executam concorrentemente um número finito de programas, de modo que a cada agente está associado um programa. O nome de função *Self*, de zero argumento, é um nome de função especial que permite a auto-identificação de agentes, pois para cada agente a, *Self* é interpretada como o próprio a.

Definição 2.20. (View) Seja S um estado e a um agente. Então define-se o estado $View_a(S)$ como a expansão do estado S onde o nome de função de zero argumento *Self* é interpretado como a.

□

Definição 2.21. (Transição) Um agente a realiza uma *transição* (ou *move*) em um estado S se o conjunto de atualizações

$$Updates(a, S) = Updates(Prog(a), View_a(S))$$

é disparado em S, ou seja, se a regra que forma o programa do agente a for disparada no estado $View_a(S)$.

□

Dentre as diversas possibilidades de se definir uma execução em uma ASM Multiagente, faz-se a opção pela assim chamada *execução parcialmente ordenada*. Em tal tipo de execução, cada transição é executada por um agente de forma atômica, o que garante que não haverá inconsistências na execução de dois ou mais agentes. A seguir, esta definição é apresentada formalmente [BS03, Tir00].

Definição 2.22. (Execução Parcialmente Ordenada) Uma *execução parcialmente ordenada* ρ de uma ASM Multiagentes $\mathcal{M}$ é uma tripla (M, A, σ) que satisfaz as seguintes condições:

- M é um conjunto parcialmente ordenado, no qual todos os conjuntos de transição $\{y \mid y \preceq x\}$, $x \in M$, são finitos. Os elementos de M representam transições realizadas pelos vários agentes durante a execução. Duas transições x e y são tais que $x \prec y$ se a transição x se iniciou após o término de y.
- A é uma função sobre M tal que todo conjunto não-vazio $\{x \mid A(x) = a\}$ é linearmente ordenado. $A(x)$ é o agente que realiza a transição x. Supõe-se que as transições de um único agente sejam linearmente ordenadas.

- σ é uma função que associa um estado de $\mathcal{M}$ a cada segmento inicial [2] de M, tal que $\sigma(\emptyset)$ é um estado inicial. Para cada segmento inicial X de M, $\sigma(X)$ é o estado que resulta da realização de todos os movimentos em X.

- A *condição de coerência*: se x é um elemento maximal em um segmento inicial finito X de M e $Y = X - x$, então $A(x)$ é um agente em $\sigma(Y)$ e $\sigma(X)$ é obtido a partir de $\sigma(Y)$ disparando $A(x)$ em $\sigma(Y)$.

□

2.4 A Linguagem Machĭna

A linguagem Machĭna foi criada pelo grupo de pesquisa do Laboratório de Linguagens de Programação do DCC / UFMG em 1999 [TMIB99b], e atualmente se encontra em sua versão 2.0 [BTIB05]. Trata-se de uma linguagem baseada no modelo ASM. Apesar da existência de outros ambientes baseados em ASM à época de sua criação, a linguagem Machĭna se justificou pelos seguintes motivos [Tir00], 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:

- a presença de estruturas para modularização que contemplam mecanismos para controle de visibilidade e proteção;

- sistema de tipos forte, pré-existindo um rico conjunto de tipos *built-in*;

- possibilidade de definição de invariantes para a execução da regra de transição da máquina abstrata, o que é um facilitador na depuração de especificações;

- implementação das regras de transição básicas e de outras mais avançadas;

- possibilidade de especificação multi-agentes de maneira simples e direta;

- noção de submáquinas, via o conceito de ações e abstrações de execução de regras que podem ser executadas a partir de outras máquinas.

Uma especificação detalhada da linguagem Machĭna foge do escopo deste texto. Entretanto, tal especificação pode ser encontrada em [BTIB05, BTM$^+$99, Tir00]. Tirelo *et al* [TMIB99b] contextualiza o projeto como um todo à época de sua criação. A seguir, são apresentados alguns pequenos programas em Machĭna, de modo a ilustrar o seu uso.

[2] Um *segmento inicial* de um conjunto parcialmente ordenado P é uma sub-estrutura de X tal que $x \in X \wedge y < x \longrightarrow y \in X$

```
01:     machina Fibonacci
02:
03:       type
04:         Output is List of Int;
05:       external
06:         nmax : Int;
07:       dynamic
08:         i, iant, n : Int;
09:         output;
10:       init
11:         i := 1; iant := 0; n := 0;
12:         output := [];
13:     transition:
14:       if n < nmax and nmax > 0 then
15:         i := i + iant;
16:         iant := i;
17:         n := n + 1;
18:         output := concat(output, [i + iant]);
19:       end
20:     end
```

Figura 2.1: Programa em Machĭna para cálculo dos nmax primeiros números de Fibonacci.

2.4.1 Exemplos de Programas em Machĭna

Números de Fibonacci

Um primeiro exemplo de um programa em Machĭna é apresentado na Figura 2.1. O objetivo deste programa é calcular os nmax primeiros números da série de Fibonacci. A numeração à esquerda foi adicionada com o objetivo de facilitar a referência ao código neste texto, não fazendo parte do código original.

Na linha 1, define-se o nome da máquina, a saber, Fibonacci. Esta máquina contém dois constituintes básicos: um estado inicial (onde também é definido o vocabulário da máquina) e a regra de transição. O estado inicial é definido a partir da linha 2 até a linha 12, enquanto que a regra de transição é definida a partir da linha 13.

Na linha 4, um novo tipo é definido: o tipo Output. Este tipo nada mais é do que uma lista de inteiros, pois o programa apresentado imprime na saída os inteiros da série de Fibonacci. Em seguida, na linha 6, declara-se que existe um nmax de tipo inteiro, cujo valor é externo à definição da máquina. As linhas 8 e 9 declaram algumas funções dinâmicas úteis: i, iant e n são funções de aridade 0 e tipo Int que ajudam no cálculo dos números de Fibonacci, ao passo que output é a saída onde os números calculados são adicionados. O tipo de output é inferido automaticamente segundo as regras de inferência apresentadas em [TMIB99b, Tir00].

A regra de transição, apresentada nas linhas 14 a 19, é executada a cada passo da máquina. Se n é maior ou igual a nmax, ou se nmax for menor ou igual a zero, então nada é feito. Caso contrário, um novo número de Fibonacci é calculado e adicionado à saída, e o contador n é incrementado em uma unidade. Ao final da execução, output contém os nmax primeiros números de Fibonacci em ordem crescente.

Jantar dos Filósofos

A Figura 2.2 apresenta uma especificação em Machĭna com vários agentes, de forma que todos executam a mesma regra de transição, concorrentemente. Para explicitar a concorrência, a regra de transição é declarada intercalada (*interleaved*), pois a regra é uma seção crítica. Este exemplo foi retirado de [Tir00].

Nas linhas 1 a 25 é especificado o módulo que define o comportamento de um filósofo. As funções dinâmicas utilizadas para a modelagem do problema são definidas nas linhas 4 a 7. initialized, thinking, hungry e eating indicam, para cada agente, qual é o estado deste agente. leftFork e rightFork representam os garfos à esquerda e à direita de um agente, respectivamente, enquanto myId fornece o identificador de um dado agente.

A regra de transição, definida nas linhas 8 a 24, verifica a possibilidade de adquirir os dois garfos e entrar no estado eating. Caso não seja possível, o filósofo permanece em hungry. Após comer com sucesso, o filósofo poe-se a pensar, até voltar a ter fome e então tentar novamente comer. Cabe notar que o acesso às funções dinâmicas e toda a execução da transição é feita dentro de uma seção crítica, ou seja, apenas um filósofo está ativo em cada momento.

As linhas 27 a 29 especificam a criação de cinco filósofos que iniciam a execução do programa.

2.5 Otimização de Código no Contexto de ASM

No processo de compilação de uma especificação ASM para um programa em linguagem imperativa, tornam-se patentes algumas ineficiências do modelo ASM. Estas ineficiências constituem-se em oportunidades de otimização, justificando o desenvolvimento de técnicas para o tratamento das mesmas. Desta forma, faz-se a seguir uma revisão das principais técnicas de otimização propostas na literatura que visam abordar as ineficiências exclusivas do modelo ASM. Estas otimizações não se sobrepõem com as técnicas convencionais de otimização de código [Muc97, Att04].

2.5.1 Escalonamento de Regras

Seja $B = U_1, U_2$ um bloco tal que U_1 e U_2 são atualizações, onde U_1 é `y:=z` e U_2 é `x:=y`. De acordo com a semântica do modelo ASM, estas atualizações não podem ser convertidas diretamente em uma seqüência de atribuições em uma linguagem imperativa, como C++, por exemplo, porque `x` poderia receber um valor incorreto de acordo com o modelo.

Uma forma de se solucionar este problema é implementar um compilador tal que regras de atualização são traduzidas para comandos de inserção em listas de atualização. Os valores do lado direito das regras de atualização são avaliados no momento da inserção, ao passo que a função dinâmica no lado esquerdo só será modificada no final da iteração. Esta abordagem evita efeitos colaterais indesejáveis que violariam a lógica do modelo.

Esta abordagem é ineficiente, pois as operações envolvidas têm um custo computacional alto. Uma maneira de minimizar o problema advém do fato de que em certas situações a atualização pode ser feita diretamente, sem a inserção na lista de atualizações. Isto acontece quando a função dinâmica a ser atualizada não é utilizada mais à frente como um *right value*. A regra U_2, do bloco B definido no início desta seção, é um exemplo de atualização que pode ser efetuada diretamente. Desta forma, a ordem em que as regras de um bloco são traduzidas para um programa em linguagem imperativa pode resultar em um número maior ou menor de

```
01:     module DinPh
02:
03:       type PhId is 1..5;
04:       dynamic
05:         initialized, thinking, hungry, eating : agent of DinPh -> Bool;
06:         leftFork, rightFork : PhId -> Bool;
07:         myId : DinPh -> PhId;
08:     transition:
09:       if not inicialized(self) then
10:         thinking(self) := true;
11:         initialized(self) := true
12:       else
13:         if thinking(self) then
14:           thinking(self) := false, hungry(self) := true;
15:         elsif hungry(self) then
16:           if not leftFork(myId(self)) and not rightFork(myId(self)) then
17:             eating(self) := true; hungry(self) := false;
18:             leftFork(myId(self)) := true;
19:             rightFork(myId(self)) := true;
20:           end
21:         else
22:           eating(self) := false; hungry(self) := true;
23:         end
24:       end
25:     end
26:
27:     machina DinPh
28:       create DinPh(5)
29:     end
```

Figura 2.2: Programa em Machĭna para simular o problema do jantar dos filósofos.

inserções na lista de atualização. Por exemplo, se U_1 e U_2 forem comutados, nenhuma inserção na lista de atualização seria necessária, e ambas as atualizações poderiam ser levadas a termo imediatamente. A otimização de *escalonamento de regras* consiste em *escalonar* a tradução das regras que compõem um bloco de modo a minimizar as inserções na lista de atualização. Esta otimização foi primeiramente proposta em [TB00], e posteriormente um algoritmo mais aprimorado foi apresentado em [OBB04b].

Segundo Oliveira *et alii* [OBB04b], esta otimização pode ser modelada como um problema de grafos, conforme descrito a seguir. Seja R uma regra de transição de uma especificação ASM. Define-se $G_R = (V_R, E_R)$ como o *grafo de escalonamento ASM* associado à regra R. G_R é um grafo dirigido formado da seguinte maneira:

- V_R é o conjunto de vértices do grafo. Cada sub-regra constituinte da regra bloco R corresponde a um vértice $v_r \in V_R$.
- Existe uma aresta $(v_i, v_j) \in E_R$ se e somente a sub-regra v_i faz a avaliação de uma função dinâmica que é atualizada na sub-regra v_j.

As arestas de G_R são rotuladas com um mesmo peso, enquanto que cada vértice $v_r \in V_R$ é associado um peso que indica o potencial benefício obtido pela remoção deste vértice durante o escalonamento. Este peso é chamado de *relação de benefício*, e o seu cálculo é apresentado adiante.

O algoritmo consiste de três passos [OBB04b]:

Passo 1: Montagem da dependência entre as funções dinâmicas.

Passo 2: Construção do grafo de conflitos a partir das dependências detectadas.

Passo 3: Escalonamento das instruções.

O primeiro passo, a montagem da dependência entre as funções dinâmicas, consiste em construir uma tabela cujas linhas correspondem às sub-regras que constituem a regra de transição ora analisada. Para cada uma das sub-regras, são anotadas quais são as funções dinâmicas consultadas e quais são as funções dinâmicas atualizadas.

Em seguida, é construído o grafo de conflitos, que considera as funções dinâmicas consultadas e atualizadas em cada sub-regra.

Finalmente, procede-se com o escalonamento em si. Como é conhecido na literatura [CLR90], o problema do escalonamento é um problema NP-completo, o que significa que não existe uma solução conhecida que forneça a solução exata e ao mesmo tempo seja garantidamente eficiente para o pior caso. Desta feita, deve-se lançar mão de uma heurística para a solução deste problema.

Oliveira *et alii* propõem que o escalonamento consista na remoção sucessiva de vértices do grafo de confiltos, até esgotar-se todos. A remoção de um vértice implica que este deve preceder os vértices restantes no grafo no momento da geração de código para a regra que originou o grafo. Se a sub-regra que está associada ao vértice removido é uma atualização de função dinâmica, deve-se indicar se o código a ser gerado pode atualizar diretamente a função dinâmica ou se deve-se utilizar o recurso de empilhar a atualização para que esta seja processada no final da iteração corrente. A escolha de qual vértice deve ser removido do grafo obedece ao seguinte critério:

1. Se não existem vértices no grafo, o escalonamento está completo e o algoritmo termina.

2. Caso contrário, é feita uma pesquisa para a montagem do *grupo* atual do grafo $G_R = (V_R, E_R)$. Um grupo $\{v_i, \ldots, v_j\} \in V_R$ é o subconjunto de V_R que contém todos os vértices de grau zero de V_R.

   (a) Se o grupo não é vazio, os vértices pertencentes a este grupo são removidos do grafo. Se estes vértices são ou contém atualizações, estas podem ser efetuadas diretamente.

   (b) Caso contrário, um vértice é escolhido para remoção, e se este vértice é uma atualização, esta será realizada por meio de uma inserção na lista de atualizações. A escolha de tal vértice é feita por meio de uma pesquisa nos vértices que participam dos *ciclos mínimos* [3] do grafo. Dentre estes vértices, é escolhido aquele cuja relação de benefício é máxima. Esta relação indica o benefício potencial da retirada deste vértice e é explicada adiante.

3. Retoma-se o primeiro passo.

Conforme dito anteriormente, a *relação de benefício* é um índice associado a cada vértice que indica o potencial benefício de sua retirada para inserção na lista de atualizações. Quanto maior este índice para um certo vértice, maior será o benefício obtido por sua remoção do grafo de conflitos. Seja v_i um vértice do grafo de conflitos, e $\{v_j \mid 1 \leq j \leq K\}$ o conjunto de todos os vértices que possuem uma aresta partindo de v_i e chegando em v_j, ou seja, a aresta e_{ij} é uma aresta do grafo de conflitos. Desta forma, define-se o benefício potencial B_i associado ao vértice v_i como

$$B_i = \left(\sum_{j=1}^{K} \frac{u_j}{e_j} \right) - u_i \tag{2.1}$$

onde u_j é o custo da inserção da atualização associada ao vértice v_j e e_j é o grau de entrada do vértice v_j. Assim, quanto maior o custo u_i de inserção de um vértice v_i na lista de atualizações, menor a sua relação de benefício. Concomitantemente, são considerados os custos da inserção dos vértices beneficiados pela remoção de v_i. Quanto maior este custo, maior a relação de benefício. Além disso, quanto menor o grau de entrada dos vértices adjacentes a v_i, maior a chance de surgirem vértices de grau zero com a remoção de v_i, o que implica em atualizações diretas, e por isso quanto menor o grau de entrada dos vértices adjacentes a v_i, maior o benefício.

Segundo os autores [OBB04b], a ordem de complexidade deste algoritmo é $\mathcal{O}(n^3)$, onde n é o número de sub-regras.

2.5.2 Otimização de Desvios

No processo de compilação de uma especificação ASM para um programa em linguagem imperativa, uma regra de transição R deve ser traduzida para um laço infinito da seguinte forma:

```
L: R; goto L; .
```

[3] Um *ciclo mínimo* em um grafo $G = (V, E)$ é um ciclo C em G tal que toda aresta de G entre dois vértices de C é também uma aresta de C.

Na presença de uma regra condicional, da forma

$$\texttt{if } g \texttt{ then } R_1 \texttt{ else } R_2,$$

esta é compilada para

$$\texttt{L: if (}g\texttt{) } R_1\texttt{; else } R_2\texttt{; goto L;}.$$

Suponha que R_2 não atualize nenhuma função dinâmica utilizada na avaliação da guarda g, e que também nenhuma função externa é chamada na avaliação da mesma. Neste caso, uma vez que R_2 é executada, não é necessário retornar a `L`, pois certamente o valor da guarda g não foi alterado. Em tal situação, um código mais eficiente seria

$$\texttt{L: if (}g\texttt{) \{}R_1\texttt{\}; else \{L2: } R_2\texttt{; goto L2; \} goto L;}.$$

A otimização de *desvios* consiste em detectar possibilidades como esta, de modo a produzir código em linguagem imperativa com desvios mais eficientes. Esta otimização foi primeiramente proposta por Tirelo e Bigonha em [TB00]. Entretanto, não foi apresentado um algoritmo para a execução da otimização.

2.6 Conclusão

Máquinas de estado abstratas, conforme revisado neste capítulo, 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. A linguagem Machĭna, com seu rico conjunto de instruções, e a pesquisa de otimizações para o modelo de ASM formam o contexto para a elaboração do trabalho aqui apresentado.

Como ASM é um tipo de semântica operacional, uma especificação ASM pode ser executada. Este trabalho apresenta como a compilação de uma especificação ASM pode ser mapeada em C++. Além disso, linguagens ASM podem variar, oferecendo construções além daquelas apresentadas no modelo básico. Assim, a linguagem MIR [Tir00, Oli04] é utilizada como linguagem para a representação de programas ASM dentro do arcabouço kℓar. O propósito desta linguagem intermediária é servir de base para a construção de compiladores para diversas linguagens ASM.

Conforme visto neste capítulo, existem problemas de desempenho inerentes na compilação de especificações ASM. Por isso, otimizações específicas do modelo fazem-se necessárias. O kℓar oferece o ambiente adequado onde estas otimizações podem ser inseridas, fazendo uso de facilidades oferecidas para a implementação destas otimizações.

Finalmente, neste trabalho, são oferecidos recursos para a implementação de modelos de concorrência em ASM.

Capítulo 3

Arcabouços e Padrões de Projeto

Este capítulo tem por finalidade apresentar aspectos importantes a serem considerados no desenvolvimento de arcabouços. Adicionalmente, são explicados também os padrões de projeto utilizados no kℓar, visando facilitar o entendimento deste durante as explicações nos capítulos seguintes.

3.1 Projeto e Uso de Arcabouços

De acordo com Johnson [Joh97a, Joh97b], duas definições se aplicam ao conceito de *arcabouço* (do original *framework*), cada uma delas com um propósito diferente. A primeira definição diz que

> *"um arcabouço é um projeto reutilizável de um sistema como um todo ou de parte deste, representado por um conjunto de classes abstratas e a forma como estas classes abstratas interagem entre si".*

Esta definição serve ao propósito de descrever a *estrutura* de um arcabouço. Uma outra definição possível para arcabouço é

> *"um arcabouço é o esqueleto de uma aplicação que pode ser adaptada por um desenvolvedor para a sua aplicação específica".*

Esta definição diz respeito ao *propósito* de um arcabouço.

De forma geral, um arcabouço é mais uma técnica de reúso, dentre várias outras. Uma destas outras técnicas é a divisão de uma aplicação em *componentes* cuja especificação é de fácil entendimento e cujo conhecimento de sua implementação não é necessário ao cliente do componente. Estes componentes podem ser conectados entre si facilmente com o objetivo de se criar uma nova aplicação. Esta técnica enfatiza o *reaproveitamento de código*. Uma outra técnica de reúso consiste na utilização de *padrões de projeto*. Segundo Gamma *et alii* [GHJV95], padrões de projeto são *"descrições de objetos e classes comunicantes que são customizados para resolverem um problema geral de projeto em um contexto particular"*. A característica principal de um padrão é permitir o *reaproveitamento de projeto*. Arcabouços, por sua vez, situam-se em algum lugar entre estas duas abordagens, permitindo tanto o reaproveitamento de código

quanto o reaproveitamento de projeto. Com efeito, pode-se encontrar três formas distintas de reúso em um arcabouço: reúso de código, reúso de projeto e reúso de análise.

Reúso de código O reúso de código em um arcabouço pode ser feito de duas formas distintas. A primeira consiste em usar os componentes existentes no arcabouço para construir uma nova aplicação. Estes componentes são facilmente utilizáveis porque se baseiam nas interfaces do arcabouço. Na segunda abordagem, novos componentes são criados a partir dos componentes já existentes, por meio da herança. O reúso se manifesta na possibilidade de aproveitar boa parte da implementação da superclasse, evitando que comportamentos idênticos tenham que ser reescritos no novo componente.

Reúso de projeto Um arcabouço fornece algoritmos abstratos reusáveis e também um projeto de alto nível que decompõe um grande sistema em pequenos componentes cuja interface é bem definida e bem documentada. A padronização das interfaces dos componentes torna possível a combinação dos componentes de diversas maneiras, de modo que muitas aplicações podem ser construídas a partir de alguns mesmos componentes. Além disso, novos componentes a serem criados devem obedecer estas interfaces, o que também evidencia o aproveitamento do projeto.

Reúso de análise No momento em que um arcabouço é projetado, os objetos importantes para o modelo a ser utilizado devem ser identificados e descritos. Um perito em um arcabouço específico vê o mundo em termos dos objetos deste arcabouço, dividindo-o nos mesmos componentes. Um arcabouço também fornece um vocabulário comum para o problema que este aborda, facilitando a comunicação entre as pessoas envolvidas.

De acordo com a primeira definição de arcabouço apresentada, as classes abstratas desempenham um papel importante em sua elaboração. Como classes abstratas não podem ser instanciadas, estas servem como um modelo para sub-classes a serem criadas, e não para objetos, definindo um *contrato* que deve ser cumprido por estas sub-classes. Este contrato se manifesta de duas formas: por um lado, a classe abstrata estabelece uma interface mínima para suas sub-classes; por outro lado, uma implementação de um subconjunto de seus métodos pode ser fornecida pela classe abstrata às suas sub-classes, estabelecendo um comportamento padrão. Enquanto a primeira forma se traduz em reaproveitamento de projeto, a segunda contribui para o reaproveitamento de código.

Um arcabouço descreve uma arquitetura de um sistema orientado por objetos estabelecendo os tipos dos objetos que o constituem e a forma como estes objetos se relacionam e interagem. Uma linguagem de programação orientada por objetos deve possuir três características que a habilitam a receber tal classificação, a saber: *abstração de dados*, *polimorfismo* e *herança*. Estas características são aproveitadas por arcabouços, o que torna a sua implementação apropriada para uma linguagem orientada por objetos. Semelhantemente a um tipo abstrato de dados, uma classe abstrata representa uma interface cuja implementação pode variar em contextos diferentes. O polimorfismo permite que componentes e seus usos sejam intercambiáveis em tempo de execução via referência a uma classe mais genérica. Por sua vez, a herança atua como uma facilitadora no processo de criação de novos componentes. Uma aplicação desenvolvida utilizando um arcabouço possui três partes: o arcabouço propriamente dito, as classes concretas que são derivadas das classes abstratas do arcabouço, e todo o resto.

Uma característica importante de um arcabouço é a *inversão de controle*, e talvez seja esta característica que mais o diferencie de uma biblioteca de classes. Um usuário de uma

biblioteca de classes escreve um programa principal que usa os componentes desta biblioteca sempre que necessário, tornando-se o responsável por administrar o fluxo de controle de todo o programa. Em um arcabouço, é o programa principal que é reutilizado, e o papel do usuário do arcabouço é conectar componentes a este programa principal segundo o objetivo a ser alcançado. O código do usuário é que é invocado pelo código do arcabouço, e é este que determina o fluxo de controle do programa.

Segundo Mattsson [Mat96, Ton04], o primeiro arcabouço usado amplamente foi provavelmente o arcabouço de interfaces de Smalltalk, a saber, o MVC (*Model-View-Controller*). Como indica o próprio nome, o MVC divide um programa em três partes: os modelos, as visões e os controles. Um modelo é um objeto de aplicação que é independente da interface de usuário; uma visão gerencia alguma área da representação visual; finalmente, o controle converte ações do usuário em operações nos modelos e nas visões.

3.2 Arcabouços Frente a Outras Abordagens de Reúso

3.2.1 Arcabouços e Bibliotecas de Classe

Apesar de um arcabouço geralmente fornecer uma boa biblioteca de classes, esta biblioteca não é a sua principal justificativa. O que realmente importa é o modelo de interação e fluxo de controle entre seus objetos. Além disso, o fluxo de controle em um arcabouço é invertido quando comparado ao fluxo de controle em um aplicativo que faça uso de uma biblioteca de classes, conforme discutido anteriormente.

Aprender um arcabouço é sempre mais difícil do que aprender uma biblioteca de classes, pois não se pode considerar uma classe do arcabouço de cada vez, nem usá-la separada de seu contexto. Um arcabouço é sempre utilizado no seu todo, não em partes. Mais ainda, em um arcabouço as classes principais são abstratas, o que torna necessário a implementação de certos comportamentos.

3.2.2 Arcabouços e Componentes

De certa forma, arcabouços podem ser vistos como componentes, pois uma aplicação pode usar diferentes arcabouços, obtidos junto a diferentes desenvolvedores. A principal diferença, no entanto, reside no fato de que arcabouços são muito mais customizáveis do que componentes, servindo para uma gama maior de situações. Nesta questão, faz-se um compromisso entre *aplicabilidade* e *facilidade de uso*. Por um lado, arcabouços são muito mais poderosos do que componentes. Por outro lado, o uso de um arcabouço exige um aprendizado maior do que o exigido para se usar um componente.

Uma abordagem melhor consiste em considerar arcabouços e componentes como tecnologias distintas. Apesar de distintas, estas tecnologias cooperam entre si. Arcabouços fornecem um contexto reutilizável para componentes, de modo que a forma como estes componentes gerenciam erros, trocam dados e invocam operações uns sobre os outros se torna padronizada.

Uma outra forma de cooperação entre arcabouços e componentes pode ser visto no fato de arcabouços faciltarem o desenvolvimento de novos componentes. Arcabouços permitem que estes novos componentes sejam construídos a partir de componentes menores, já existentes, além de fornecer especificações e modelos de implementação para os mesmos.

3.2.3 Arcabouços e Padrões de Projeto

Um padrão de projeto descreve um problema a ser resolvido, uma solução e o contexto onde esta solução é implementada. De forma geral, padrões de projeto têm se tornado populares entre os desenvolvedores de aplicações orientadas por objetos. Um padrão nomeia uma técnica testada e consagrada pelo seu uso, e descreve os custos e benefícios em se adotar esta técnica em particular. Os devenvolvedores que usam os mesmos padrões também compartilham um vocabulário comum para conversarem sobre seus projetos e explicitarem decisões sobre os mesmos.

Segundo Johnson, como arcabouços foram implementados várias vezes, estes de certa forma se constituem um tipo de padrão [Joh97a, Joh97b]. A diferença é que um arcabouço também possui código, ao passo que um padrão é apenas uma idéia.

Outro ponto de relacionamento entre arcabouços e padrões é que muitos padrões surgiram do exame cuidadoso de arcabouços com o objetivo de encontrar elementos representativos de sua reusabilidade. Um arcabouço contém geralmente muitos padrões diferentes, de forma que um arcabouço é algo maior do que um padrão. Um padrão é mais abstrato que um arcabouço, ou seja, ambos estão em níveis diferentes de abstração.

Gamma *et alii* [GHJV95] apresenta três diferenças cruciais entre padrões de projeto e arcabouços:

- *Padrões de projeto são mais abstratos que arcabouços* Arcabouços podem ser representados por meio de código, mas apenas *exemplos* de padrões podem ser representados por código. Uma das vantagens de um arcabouço é que ele pode ser escrito em uma linguagem, de forma que sua especifiação não é apenas passível de ser estudada mas também de ser *executada e reusada diretamente*. Em contraste, padrões de projeto devem ser *implementados* cada vez que são utilizados.

- *Padrões de projeto são elementos arquiteturais menores do que arcabouços* Um arcabouço típico contém vários padrões, embora o contrário nunca seja verdadeiro.

- *Padrões de projeto são menos especializados que arcabouços* Um arcabouço sempre tem um domínio de aplicação bem específico. Em contraste, padrões de projeto podem ser usados em praticamente qualquer tipo de aplicação.

3.2.4 Arcabouços e Outras Formas de Reúso

Segundo Johnson [Joh97a], arcabouços são semelhantes a outras formas de reúso de código, tais como *templates* e *esquemas*. A principal diferença é que arcabouços são expressos em uma linguagem de programação, ao passo que estas outras formas de reúso de alto nível usualmente usam uma notação especial de projeto e freqüentemente se apóiam em ferramentas especiais de software. O fato de um arcabouço ser um programa facilita o seu uso. Ao mesmo tempo, limita uma implementação de um arcabouço específico àquela linguagem em que este foi escrito. Exceções são possíveis, embora limitadas em quantidade. Por exemplo, um programa Java pode invocar funções e métodos em C e C++ via JNI - *Java Native Interface* [Lia99].

Arcabouços também são semelhantes a geradores de aplicações. Um gerador de aplicações se baseia em uma linguagem de alto nível de domínio específico que é compilada para uma arquitetura padrão. Um arcabouço também é uma arquitetura padrão. Desta forma, à exceção da sintaxe e do fato que o tradutor de um gerador de aplicações pode realizar otimizações, as

duas técnicas são similares. É usual que estas duas técnicas sejam combinadas, de forma que programas na linguagem do gerador são traduzidos para um conjunto de objetos do arcabouço.

Existe uma semelhança entre arcabouços e arquiteturas. Entretanto, a principal diferença está no fato que um arcabouço é necessariamente um projeto orientado por objetos, ao passo que uma arquitetura de um domínio específico não precisa ser.

3.2.5 Considerações Finais sobre Arcabouços

Dentre as diversas técnicas de reúso de código, pode-se situar o arcabouço no meio de um compromisso entre facilidade de uso e generalidade. Como dito anteriormente, um arcabouço é mais abstrato e flexível do que componentes, e ao mesmo tempo mais concreto e mais fácil de usar do que um projeto iniciado da estaca zero. Esta última proposta é, no entanto, mais flexível e aplicável. Ao se comparar um arcabouço com técnicas que reusam tanto código quanto projeto, tais como geradores de aplicações e gabaritos (*templates*), sua princpal vantagem é que um arcabouço pode ser implementado sob qualquer linguagem orientada por objetos, sem necessidade de ferramentas especiais.

Apesar das várias vantagens de se utilizar arcabouços, a reusabilidade não é gratuita. Arcabouços são difíceis de se desenvolver e de se aprender. Usar um arcabouço requer que um mapeamento da estrutura do problema a ser resolvido na estrutura do arcabouço, o que dificulta a tarefa de projetar um arcabouço, pois este deve ser o mais genérico possível, mas sempre observando o compromisso assumido entre generalidade e facilidade de uso. A dificuldade também aparece do lado do usuário do arcabouço, que deve fazer este mapeamento entre o problema existente e o arcabouço disponível. Esta tarefa é facilitada pela presença de uma boa documentação do arcabouço, cuja tarefa consiste em explicar o propósito de arcabouço, como usá-lo e como este realiza a sua tarefa.

Arcabouços são eficientes para se expressar projetos reutilizáveis. Suas vantagens, citadas anteriormente, justificam a maior dificuldade em implementá-los. Cabe ao cliente do arcabouço avaliar se um arcabouço específico atende às suas necessidades e se este realiza os compromissos adequados entre poder e simplicidade.

3.3 Padrões de Projeto

Um dos objetivos deste trabalho é desenvolver um ambiente de otimizações para a representação intermediária MIR e sua conversão para código C++. Este ambiente deve ser facilmente *estensível* e *reutilizável*, e na busca por estes objetivos é que os padrões de projeto desempenham um papel importante.

Padrões de projeto foram propostos por Gamma *et alii* em [GHJV95], e logo se tornaram padrões *ipso facto* no projeto de sistemas orientados por objetos. Seus autores definem um padrão de projeto como *"descrições de objetos e classes comunicantes que são customizados para resolverem um problema geral de projeto em um contexto particular"*, e apresentam um conjunto de vinte e três padrões fundamentais, classificados segundo dois critérios: *escopo* e *propósito*.

O primeiro critério de classificação, o escopo, é definido para um padrão dependendo de qual tipo de elemento este se aplica primariamente.

De Classe: Um padrão pode ser *de classe*, quando lida com classes, subclasses e suas relações. Uma relação entre classes e subclasses é sempre estática, pois é estabelecida por meio da

herança, o que limita a sua definição ao momento em que estas classes são projetadas e implementadas.

De Objeto: Um padrão pode ser *de objeto*, quando lida com relacionamentos entre objetos. Diferentemente de uma relação entre classes, uma relação entre objetos é dinâmica, podendo ser alterada durante a execução de um programa.

O segundo critério de classificação, o propósito, reflete o que o padrão faz. Desta forma, quanto ao seu propósito, um padrão pode ser:

De Criação: quando o padrão atua no processo de criação de objetos. Neste caso, o sistema tende a se tornar independente de como os objetos são criados, compostos e representados. De forma geral, estes padrões encapsulam a informação de quais classes concretas um sistema faz uso e escondem a implementação do processo de instanciação de objetos destas classes concretas.

Estrutural: quando o padrão estabelece a composição de classes e objetos de maneira a se construir estruturas maiores. Enquanto padrões estruturais de classe se baseam no conceito de herança entre classes, padrões estruturais de objeto descrevem formas de se compor objetos existentes de modo a criar uma nova funcionalidade.

Comportamental: quando o padrão determina a maneira na qual classes ou objetos se interagem e distribuem responsabilidades. Nestes padrões, não apenas as classes, objetos e suas relações são estabelecidos, mas principalmente suas formas de interação e comunicação. Fluxos de controle de natureza mais complexa podem ser definidos apropriadamente por meio dos padrões comportamentais.

Esta seção tem por finalidade a revisão dos padrões de projeto empregados no kℓar. Cada padrão é primeiramente apresentado tal como definido por Gamma *et alii* [GHJV95]. Na Seção 6.3 é feita a sua contextualização no projeto, sendo apresentadas as classes envolvidas e seus papéis no padrão. A notação UML, e particularmente diagramas de classe, é utilizada para ilustrar os padrões e seu usos no kℓar. Detalhes sobre esta notação podem ser encontrados em [Fow04, Pil03].

3.3.1 Padrão *Estado*

O padrão *Estado* é um padrão comportamental de objeto e pode ser usado sempre que objetos de alguma classe devam mudar seu comportamento de acordo com alterações nas condições internas destes objetos. As condições internas caracterizam o seu estado atual. Nestes casos, há uma tendência à existência de comandos condicionais de estrutura similar em cada um dos métodos de tal classe. Esta estrutura avalia as condições internas, determinando o estado atual do objeto. O uso do padrão atua separando as partes destas estruturas similares em classes distintas apropriadamente, de modo que cada possível estado pode ser visto como um objeto por si mesmo.

Este padrão possui três participantes, a saber:

Contexto: sua responsabilidade é definir a interface de interesse dos clientes e manter uma instância de um estado concreto que defina o estado atual.

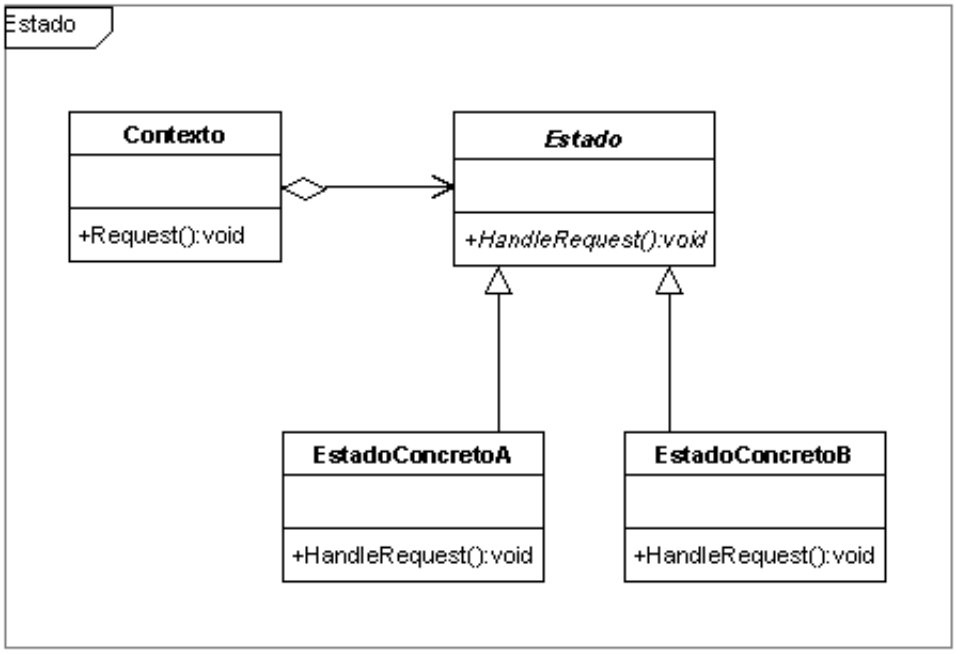

Figura 3.1: Diagrama de classes do padrão *Estado*.

Estado: define uma interface que encapsula o comportamento associado com um estado em particular do Contexto.

EstadoConcreto: consistem em subclasses de Estado que implementam um comportamento associado a um estado em particular do Contexto.

Estes participantes se relacionam segundo o diagrama de classes apresentado na Figura 3.1. O Contexto é a interface primária de clientes, e quando requisições lhe são feitas, estas são repassadas ao estado atual. O estado atual é um objeto de alguma subclasse concreta de Estado, o qual sabe exatamente o que deve ser feito. Os clientes não precisam lidar diretamente com os estados, e tanto o Contexto quanto as subclasses do tipo EstadoConcreto podem decidir qual é o próximo estado a cada instante.

3.3.2 Padrão *Fábrica Abstrata*

O padrão *Fábrica Abstrata* é um padrão de criação de objeto que pode ser utilizado quando um sistema deve ser independente de como seus produtos são criados, compostos e otimizados. Este padrão especifica uma interface para a criação de objetos de uma mesma família sem especificar suas classes concretas. Os participantes deste padrão são:

FábricaAbstrata: Especifica uma interface para as classes que criarão os produtos.

FábricaConcreta: Cada classe FábricaConcreta implementa a interface FábricaAbstrata apropriadamente, de modo que seus produtos concretos sejam corretamente criados.

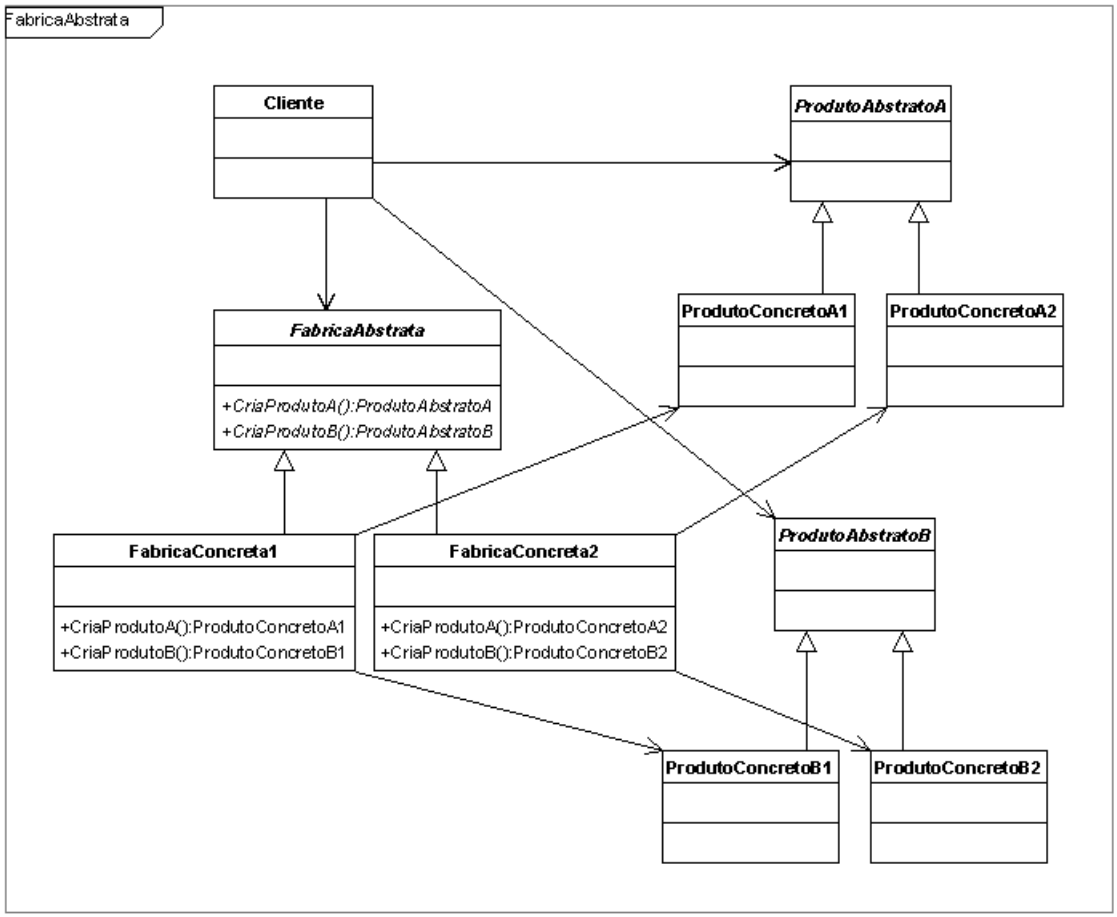

Figura 3.2: Diagrama de classes do padrão *Fábrica Abstrata*.

ProdutoAbstrato: Especifica uma interface para os produtos a serem criados por meio de suas classes concretas.

ProdutoConcreto: Cada classe ProdutoConcreto implementa a interface ProdutoAbstrato apropriadamente, de modo a constituir um produto a ser criado por sua respectiva FábricaConcreta.

Cliente: Faz uso das interfaces FábricaAbstrata e ProdutoAbstrato. O acesso às classes concretas é feito apenas por meio destas interfaces.

Estes participantes se relacionam segundo o diagrama de classes apresentado na Figura 3.2. Em geral, uma única instância de cada fábrica concreta é criada, e para se obter produtos diferentes, deve-se usar fábricas diferentes a cada momento.

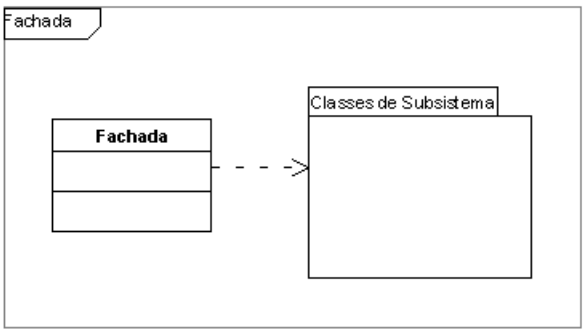

Figura 3.3: Diagrama de classes do padrão *Fachada*.

3.3.3 Padrão *Fachada*

O padrão *Fachada* é um padrão estrutural de objeto que deve ser usado para fornecer uma interface simples para um subsistema complexo. As várias interfaces presentes neste subsistema são sumarizadas em uma interface única, que representa uma visão simplificada do subsistema como um todo. Esta interface de mais alto nível é utilizada por clientes do subsistema, introduzindo um desacoplamento entre o subsistema e seus clientes. Este desacoplamento aumenta a independência e a portabilidade do subsistema. Além disso, cada subsistema pode ter a sua própria fachada, simplificando as dependências entre os subsistemas e permitindo a estruturação do sistema em camadas. Os participantes deste padrão são:

Fachada: Esta classe é a responsável pelo acesso às demais classes do subsistema. A Fachada sabe exatamente como tratar cada requisição recebida, repassando-a aos objetos da classe apropriada. Algum trabalho de interpretação e adaptação da requisição pode ser feito na Fachada antes do repasse aos objetos.

Classes do subsistema: As demais classes implementam as funcionalidades do subsistema, e por isso recebem as requisições da Fachada. Estas classes não têm consciência da existência da Fachada, sendo apenas conhecidas por esta.

Estes participantes se relacionam segundo o diagrama de classes apresentado na Figura 3.3. Os clientes que utilizam a Fachada não tem acesso às demais classes do subsistema.

3.3.4 Padrão *Singleton*

O padrão *Singleton* tem por finalidade garantir que uma classe possua uma e somente uma instância. Trata-se de um padrão de criação de objeto, e pode ser usado sempre que deva existir apenas uma única instância de uma classe e que esta seja acessível a clientes por meio de um ponto de acesso bem definido. Neste caso, os clientes acessam as funcionalidades do Singleton exclusivamente via este ponto de acesso.

Algumas vantagens deste padrão são: o controle de como os clientes acessam esta instância única; redução do espaço global de nomes, diminuindo a necessidade de variáveis globais; possibilidade de extensão da classe resultante por meio de herança, o que seria insatisfatório se

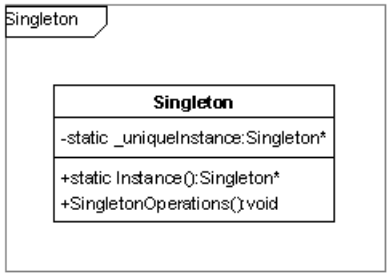

Figura 3.4: Diagrama de classes do padrão *Singleton*.

fossem utilizados métodos de classe como alternativa (por exemplo, métodos estáticos em C++ nunca são virtuais); e, finalmente, possibilidade de configuração rápida do comportamento almejado por meio do uso da instância da classe desejada que seja uma subclasse de alguma classe básica.

O único participante deste padrão é a classe Singleton, e sua estrutura é apresentada na Figura 3.4. Este participante possui um método de classe que retorna a instância única da classe.

3.3.5 Padrão *Estratégia*

O padrão *Estratégia* é um padrão comportamental de objeto e define uma família de algoritmos, encapsulando-os e fazendo estes algoritmos intercambiáveis. Este padrão pode ser usado em situações diversas, entre as quais cabe destacar a situação em que muitas classes diferem apenas em seu comportamento, mas não em sua constituição. Outra situação pertinente é quando se faz necessária a existência de várias versões de um mesmo algoritmo, ou de vários algoritmos com um mesmo objetivo. Enfim, o padrão Estratégia permite que um algoritmo varie independentemente das classes que fazem uso dele.

Os participantes deste padrão são:

Estratégia Define a interface comum a todos os algoritmos do padrão.

EstratégiaCompleta Implementa um algoritmo compatível com a interface Estratégia.

Contexto Faz uso das classes EstratégiaCompleta por meio de uma referência a uma Estratégia.

A estrutura geral deste padrão é apresentada na Figura 3.5. Cabe ressaltar que o Contexto e a Estratégia se interagem da seguinte forma: é o Contexto que passa para a Estratégia (ou, mais especificamente, para um objeto de uma de suas classes concretas derivadas) as estruturas de dados necessárias para a execução do algoritmo.

3.3.6 Padrão *Composite*

O padrão *Composite* é uma padrão do tipo estrutural e permite que estruturas sejam criadas de tal forma que é possível tratar objetos individuais e composições de objetos uniformemente. Assim, sua aplicabilidade é manifesta sempre que tal tratamento uniforme é desejado.

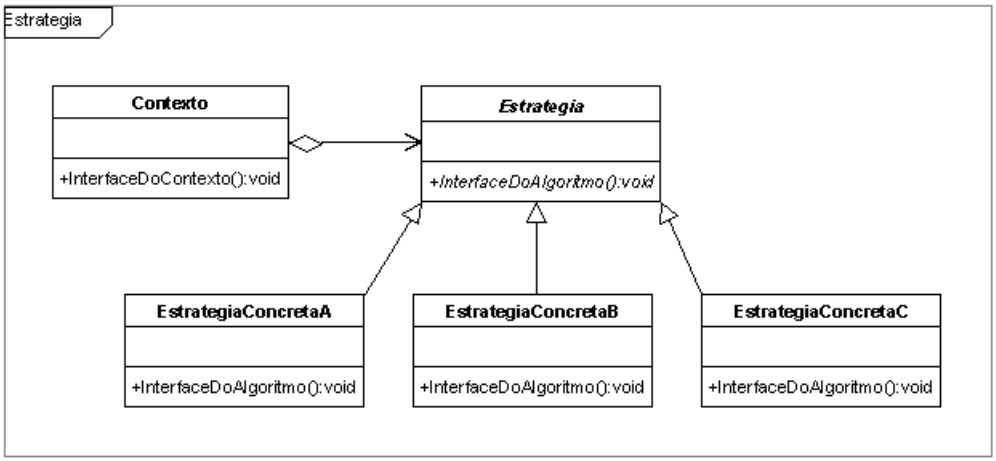

Figura 3.5: Diagrama de classes do padrão *Estratégia*.

Os participantes deste padrão são:

Componente Este participante determina a interface comum a todas as classes. Além disso, certos comportamentos padrões de elementos da hierarquia podem ser adicionados neste participante, haja vista que todas as classes da estrutura dele derivam.

Folha Uma folha representa um objeto que não possui nodos-filho. Trata-se dos objetos primitivos da composição.

Composição Este participante representa um nodo que possui filhos. Estes nós podem ser folhas ou outros nodos de composição. Este é o elemento que permite a recursão na estrutura.

Cliente Manipula a estrutura por meio da interface fornecida por Componente.

A estrutura geral deste padrão é apresentada na Figura 3.6.

3.3.7 Padrão *Visitor*

O padrão *Visitor* representa uma operação a ser realizada sobre os elementos de uma estrutura de objetos. Por meio deste padrão, é possível que novas operações sejam definidas sem que as classes dos elementos sobre os quais a operação é aplicada sejam alteradas.

A utilização de visitors é particularmente útil quando várias operações distintas e não-relacionadas necessitam de ser realizadas em objetos de uma dada estrutura, mas não se deseja poluir as classes de tais objetos com as operações. Um visitor permite que as operações relacionadas sejam agrupadas em uma única classe, ao invés de serem espalhadas pelas diversas classes da estrutura. Os principais participantes deste padrão são:

Visitor Este participante declara um método Visit para cada classe do tipo ElementoConcreto. O nome do método e sua assinatura identifica a classe que envia a requisição

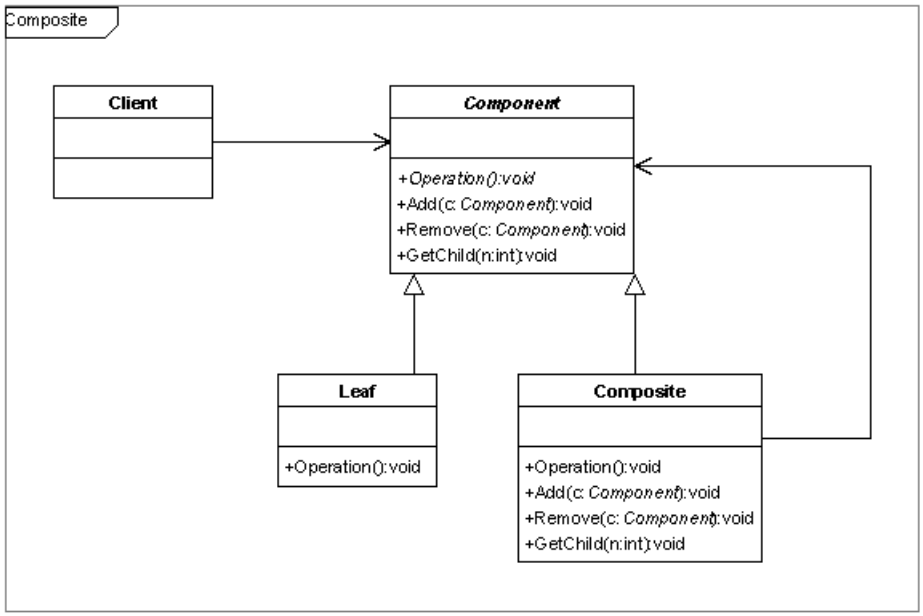

Figura 3.6: Diagrama de classes do padrão *Composite*.

Visit ao Visitor, o que permite que este efetue as ações corretas sobre o elemento que é visitado. Desta forma, o visitor pode acessar o elemento por sua interface própria, e não apenas por meio de uma interface comum.

VisitorConcreto Implementa cada método declarado pelo Visitor. Cada método consiste em uma parte do algoritmo definido para a classe correspondente na estrutura de objetos. O VisitorConcreto fornece o contexto para o algoritmo e armazena o seu estado local, que é modificado durante a travessia pela estrutura.

Elemento Define um método Accept que toma um Visitor como argumento.

ElementoConcreto Implementa a operação Accept.

Estrutura A estrutura a ser percorrida pelos visitors.

Um ponto a ser destacado é que este padrão é intimamente relacionado com o padrão Composite, haja vista que este último define de forma prática uma estrutura sobre a qual visitor podem caminhar realizando operações.

3.3.8 Considerações Finais sobre Padrões de Projeto

Padrões de projeto afetam o projeto de software orientado por objetos de várias formas [GHJV95]. Primeiramente, o uso de padrões estabelece uma linguagem comum acerca das estruturas modeladas pelos padrões. Esta linguagem comum favorece uma comunicação mais eficiente tanto para a equipe de desenvolvimento quanto para a documentação do projeto como um todo.

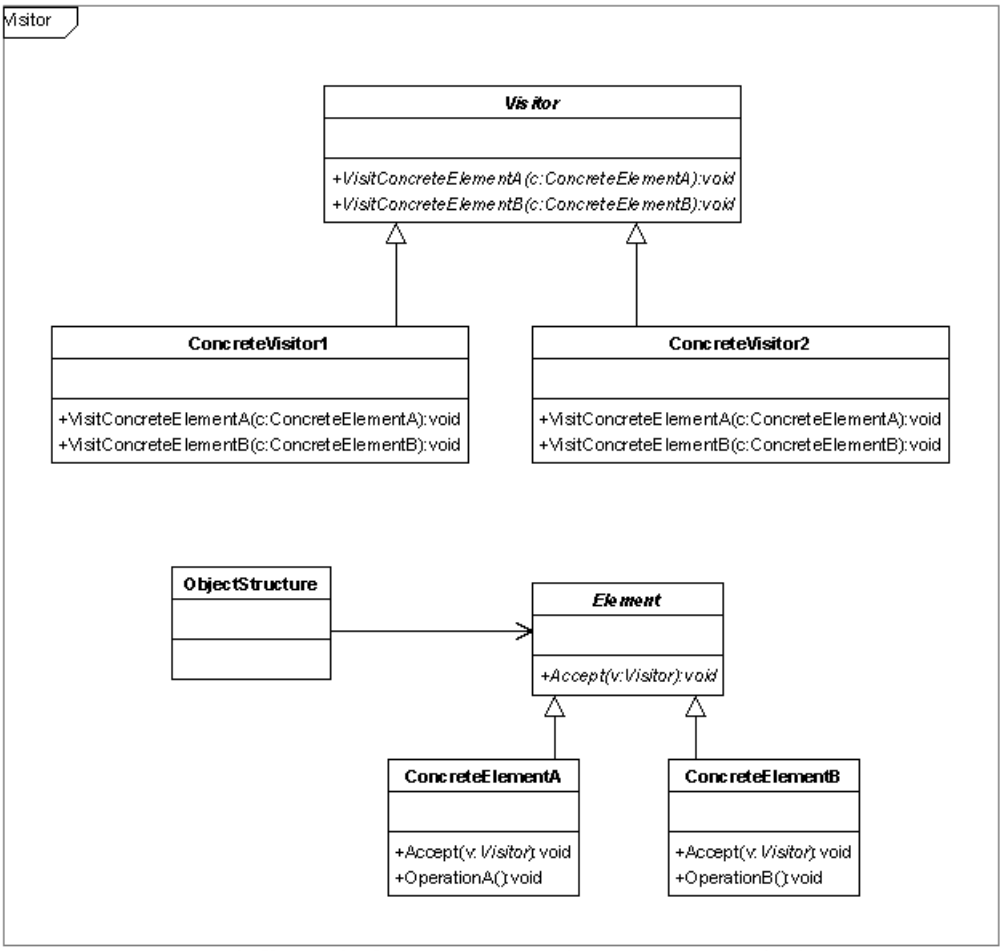

Figura 3.7: Diagrama de classes do padrão *Visitor*.

Além disso, a utilização de padrões facilita o entendimento de um sistema já desenvolvido. Descrever o sistema em termos dos padrões existentes permite que as estratégias e os conceitos envolvidos sejam rapidamente esclarecidos por meio da alusão a um padrão conhecido.

O uso de padrões de projeto também se integra a metodologias existentes de desenvolvimento de software orientado por objetos. Particularmente, os padrões são úteis na transição do modelo obtido durante a fase de análise para o modelo de implementação. Uma implementação flexível e reutilizável contém classes e associações destas que não são previstas durante a fase de análise.

Finalmente, padrões de projeto auxiliam durante a refatoração. *Refatoração*, segundo Fowler [Fow99], é o processo de alterar um sistema de software de tal forma que seu comportamento exterior não é alterado, mas que melhora a estruturação interna de tal sistema. Neste contexto, padrões de projeto auxiliam na reestruturação de um projeto durante a refatoração, e podem diminuir a necessidade de novas refatorações no futuro.

Os padrões de projeto aqui apresentados são os padrões empregados no projeto do kℓar e constituem apenas um pequeno subconjunto dos padrões existentes. Outros muitos exemplos são apresentados por Gamma *et alii* [GHJV95] e por Fowler [Fow03], entre outros.

3.4 Conclusão

Devido às características dos arcabouços, pode-se concluir que esta forma de estruturação se mostra a mais adequada para o projeto do kℓar. Em particular, a inversão de controle se manifesta no kℓar quando da chamada das otimizações acopladas ao mesmo: o desenvolvedor de otimizações escreve o código que será invocado pelo arcabouço em momento oportuno, a saber, o momento em que as otimizações se fazem necessárias. As estruturas de suporte às otimizações, a definição de interfaces a serem seguidas e a identificação e implementação no kℓar de classes que representam conceitos importantes de especificações ASM, como expressões, regras e tipos, caracterizam o reúso de código, projeto e análise.

Várias partes do kℓar foram estruturadas com base em padrões de projeto. Como estas são soluções consagradas para problemas recorrentes, a utilização de padrões de projeto facilita a implementação de muitos requisitos do kℓar, assim como permite que leitores familiarizados com estes padrões compreendam instantaneamente as estratégias utilizadas. Além disso, o uso de soluções já testadas aumenta o grau de confiabilidade do projeto. O uso destes padrões no kℓar é apresentado nos Capítulos 5 e 6.

Capítulo 4

Linguagem Intermediária para Representação de ASM Multi-Agentes

O arcabouço proposto neste trabalho tem por objetivo permitir que otimizações específicas do modelo ASM sejam realizadas facilmente. Estas otimizações implicam em *transformações* em uma dada especificação, de modo que a especificação resultante seja mais eficiente do que aquela proposta inicialmente, e concomitantemente preserve a semântica original. Desta forma, faz-se necessária a definição de uma *linguagem* na qual estas especificações sejam escritas de modo a serem entendidas pelo arcabouço kℓar. Este capítulo tem como propósito apresentar esta linguagem [Tir00, Oli04, OBB04a].

Uma vez definida a linguagem, foram projetadas classes no kℓar com o objetivo de prover funcionalidades importantes. O conjunto das funcionalidades providas é constituído por:

- representação em memória de especificações válidas por meio de classes apropriadas;
- geração de código C++ equivalente a uma especificação válida;
- carregamento de representações em memória de especificações descritas na linguagem proposta;
- persistência de representações em memória em arquivos que podem ser carregados posteriormente.

No fim do capítulo, alguns exemplos são apresentados com o intuito de ilustrar adequadamente o uso da linguagem proposta para a definição de especificações ASM e são tecidas conclusões sobre a mesma.

4.1 Visão Geral

A linguagem proposta é baseada nas construções da linguagem Machĭna [BTIB05, Tir00], que foi abordada na Seção 2.4, e procura implementar a semântica da arquitetura MIR (*Machĭna Intermediate Representation*) [Oli04]. Por isto ela é chamada de *Linguagem MIR*. Entretanto, cabe ressaltar que esta linguagem tem por objetivo ser uma linguagem intermediária de uso geral para compiladores de especificações ASM, e embora seja usada dentro do contexto do projeto Machĭna, conforme discutido na Seção 1.2.2, a linguagem e o arcabouço como um todo podem ser usados em outros contextos diversos daquele ali apresentado.

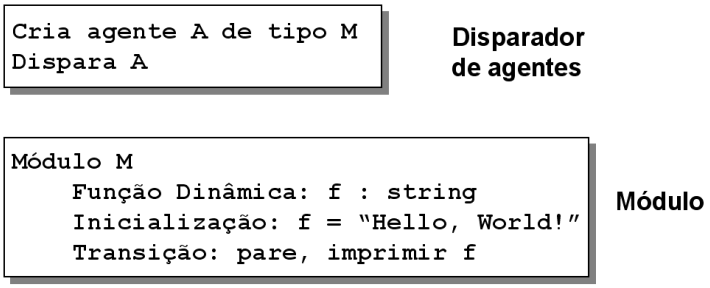

Figura 4.1: Um exemplo de programa em MIR.

A linguagem MIR foi definida por Tirelo *et alii* [Tir00], e depois modificada por Oliveira *et alii* [Oli04, OBB04a]. Nesta dissertação, propõe-se uma representação XML de MIR e introduz-se algumas modificações na mesma.

A nossa contribuição importante para a linguagem MIR é a proposta de um novo modelo de concorrência para ASM. A necessidade de tal modelo e a sua definição foram apresentadas em [MBB+05] e são descritas na próxima seção de modo a fornecer o contexto adequado para a definição de fato da linguagem MIR, feita na Seção 4.4. A idéia geral do modelo de concorrência é apresentada primeiramente de forma independente dos detalhes desta implementação em particular.

4.1.1 Fluxo de Controle: Um Primeiro Exemplo

Antes de detalhar as construções da linguagem, apresenta-se aqui um pequeno exemplo cujo objetivo é ilustrar como o fluxo de controle acontece para um programa em MIR. Ao invés de apresentar exatamente a sintaxe da linguagem utilizada, faz-se uso de uma representação mais intuitiva, de modo a concentrar as atenções no fluxo de controle em si. O programa de exemplo é apresentado na Figura 4.1. Nesta figura estão representadas as duas principais entidades de compilação de um programa MIR: um *disparador de agentes* e um *módulo*. Cada uma destas entidades consiste em um arquivo separado. De forma geral, um programa tem um disparador de agentes e um ou mais módulos.

É pelo disparador de agentes que a execução de um programa em MIR se inicia. Nele são criados um ou mais agentes, cada qual com o seu tipo. O tipo de um agente é um módulo. No exemplo da Figura 4.1, o agente criado tem nome A e é do tipo M, sendo M o módulo declarado no arquivo logo abaixo na figura. Isto significa que o agente A possui os elementos declarados no início de M, que neste caso é apenas uma função dinâmica f. Entretanto, outros tipos de elementos poderiam ser declarados, como funções de outros tipos, ações e semáforos. Depois de criados, os agentes são explicitamente disparados. Disparar um agente significa iniciar a execução de sua regra de transição em uma *thread* independente.

Quando A é criado, a sua regra de inicialização é executada uma e apenas uma vez. No caso de A, que possui tipo M, isto significa inicializar a sua função dinâmica 0-ária f com a string "Hello, World!".

O disparo de um agente implica na execução cíclica de sua regra de transição em uma *thread* separada. No caso de A, a regra de transição possui duas sub-regras, executadas em paralelo. Na primeira, existe um comando explícito para parar a execução da regra de transição. Na segunda, o conteúdo de f, a saber, "Hello, World", é impresso na saída padrão. Observe que como as sub-regras são executadas em paralelo, a regra que indica o fim da execução da

regra de transição não afeta a execução da regra que imprime a mensagem na saída padrão. Esta indicação de parada só é levada a termo no fim da execução da transição corrente. Um programa termina a execução quando todos os agentes disparados terminam a execução da regra de transição.

4.2 Modelo de Concorrência para Máquinas de Estado Abstratas

A definição original de ASM multiagentes, conforme apresentada por Gurevich [Gur95], é limitada do ponto de vista da modelagem de propriedades usualmente encontradas em sistemas concorrentes. De fato, esta definição original permite a especificação de sistemas concorrentes. Todavia, ela não oferece facilidades para a definição de sistemas multi-agentes da forma como apresentada neste trabalho. Por exemplo, dois ou mais agentes não podem executar simultaneamente regras de transição que fazem uso de um recurso comum, pois eventuais conflitos no uso de um recurso compartilhado são resolvidos por meio da ordenação parcial das execuções conflitantes. O modelo proposto pretende resolver questões como estas.

Em uma ASM *seqüencial*, conjuntos de atualizações são gerados pela execução de uma regra de transição apenas. Em uma ASM *concorrente*, múltiplos agentes podem executar regras de transição diferentes. O conceito de agente adotado neste trabalho é aquele apresentado por Zambonelli *et al* [ZJW03], no qual um *agente* é definido como uma entidade de software que exibe três propriedades: *autonomia*, *consciência de contexto* e *proatividade*. Estas propriedades se fazem presentes no modelo proposto da seguinte forma:

Autonomia: existe uma regra de transição associada a cada agente, o que lhe confere um comportamento autônomo e independente de outros agentes. Desta forma, um agente não está passivelmente sujeito a um fluxo de controle global e externo. Ao contrário, este possui sua própria *thread* de execução, tipicamente orientada de modo a atingir um objetivo específico, exibindo a capacidade de decidir o que fazer a cada momento.

Consciência de contexto: todo agente habita um contexto global comum a todos os agentes, de modo que um agente pode interagir com seus iguais, seja observando este contexto, seja alterando-o de modo a se fazer sentir pelos demais agentes.

Proatividade: de modo a cumprir seus objetivos em um ambiente dinâmico e imprevisível, agentes podem ser programados para exibir um comportamento antecipatório, tomando ações apropriadas que persigam um objetivo em particular.

Em um sistema multi-agentes como o de uma ASM concorrente, o comportamento global é derivado da interação entre os agentes constituintes. Agentes interagem entre si mesmos por meio de cooperação, coordenação ou negociação. Este tipo de interação é chamada de *sociabilidade*.

No modelo apresentado não é prevista nenhuma primitiva de sincronização entre agentes. O modelo é genérico o suficiente para permitir atualizações simultâneas inconsistentes produzidas por diferentes agentes. Não existe uma solução automática para a solução de um eventual uso conflitante de um recurso compartilhado. Toda a comunicação e sincronização deve ser programada explicitamente pelo projetista da especificação ASM concorrente. Para tal, faz-se uso de semáforos acompanhados das instruções tradicionais associadas a seu uso, a saber,

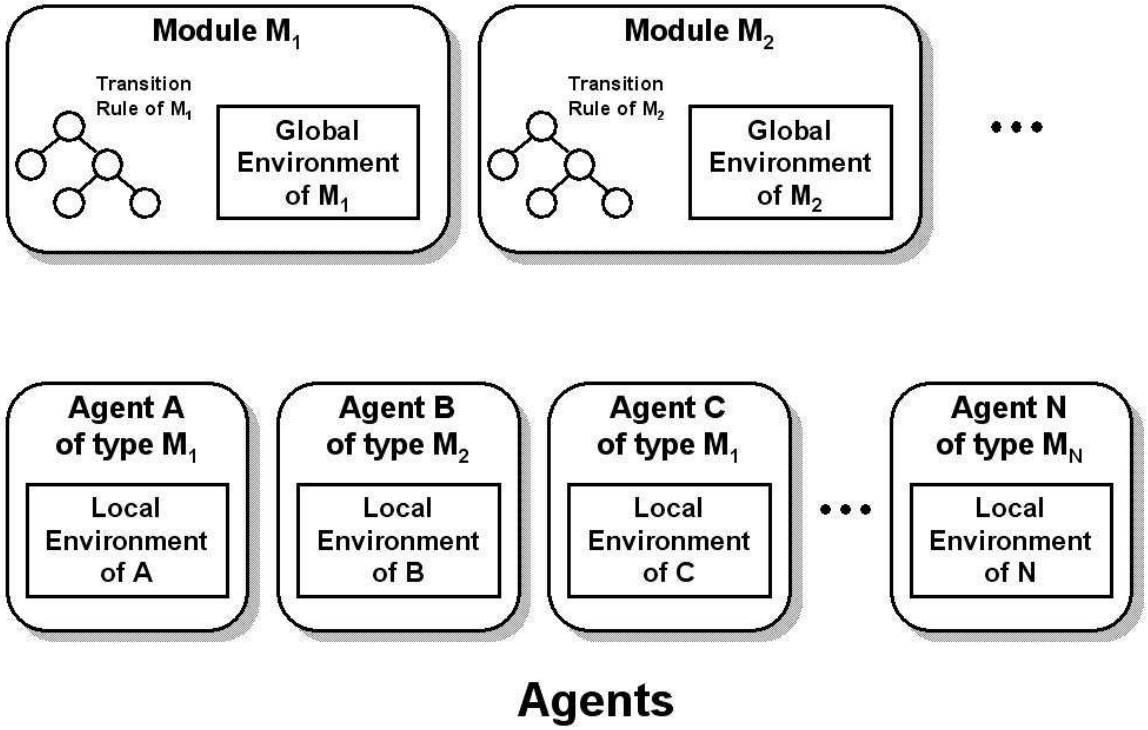

Figura 4.2: Arquitetura MIR

signal e *wait*, e também de memória compartilhada. A próxima seção apresenta como estas idéias são implementadas de fato.

4.3 Arquitetura

A arquitetura aqui apresentada é baseada naquela definida por Oliveira [Oli04]. As modificações introduzidas dizem repeito à introdução de semáforos como elementos da linguagem e também a introdução de regras para lidar com estes semáforos.

Uma especificação na linguagem MIR é composta basicamente de um conjunto de agentes, cada um de um dado tipo, que habitam em um contexto comum. O tipo de um agente é um módulo. Um módulo possui uma regra de transição, que é executada ciclicamente por cada agente cujo tipo é o módulo em questão. Adicionalmente, um módulo também possui dois ambientes, sendo um local e o outro global. O ambiente global é comum a todos os agentes de um mesmo módulo, ao passo que existe uma cópia individual do ambiente local para cada agente do módulo. Esta composição é ilustrada na Figura 4.2.

Um ambiente é constituído de tabelas para as funções estáticas, derivadas, dinâmicas e externas, agrupadas segundo os seus tipos, de tabelas para abstrações de regras, a saber, ações e submáquinas, e também de uma tabela de semáforos, como apresentado na Figura 4.3. Funções estáticas são aquelas cujos valores em pontos do seu domínio não podem sofrer alterações por regras de atualização. Estas funções são definidas por expressões parametrizadas, e permanecem imutáveis durante toda a execução. Dentro de sua definição, não são permitidas chamadas a funções que não outras funções estáticas. Uma função derivada é em

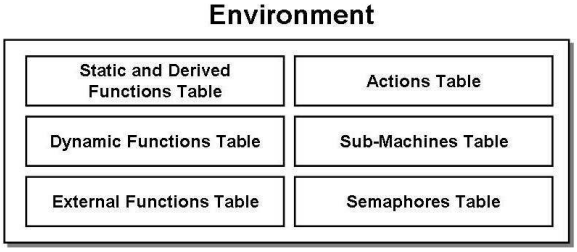

Figura 4.3: Os constituintes de um ambiente.

tudo semelhante a uma função estática, excetuando-se pelo fato de que chamadas a funções de qualquer natureza são permitidas em sua definição. Por outro lado, uma função dinâmica pode ter seus valores em pontos de seu domínio modificados ou definidos por regras de atualização. Funções externas são funções cujo corpo é definido fora da especificação em linguagem MIR, e apenas seu tipo de retorno e sua assinatura são conhecidas *a priori*. Este tipo de função é útil para simular interação com o ambiente. Funções estáticas e derivadas são agrupadas em uma mesma tabela por causa das semelhanças em suas definições.

As abstrações de regras são de dois tipos: as ações e as submáquinas. As ações são como procedimentos. Elas nomeiam e parametrizam uma dada regra, de modo que esta possa ser usada em vários lugares, permitindo aproveitamento de código. No ponto de chamada de uma ação, o corpo da ação chamada é executado uma vez (execução *single-step*), produzindo o efeito especificado pela regra que define este corpo. Este comportamento é diferente daquele observado pela chamada a uma submáquina. Uma chamada a uma submáquina tem como efeito a execução cíclica da regra associada a ela até que uma indicação explícita de retorno seja encontrada (execução *multi-step*).

Finalmente, um ambiente possui também uma tabela de semáforos, que são usados para a sincronização de agentes, sejam estes de um mesmo módulo ou não.

A regra de transição tem a forma de uma árvore cujos nós são as diversas regras de transição disponíveis na linguagem. Estas regras são detalhadas na Seção 4.4.1. O que deseja-se ressaltar neste ponto é a composição de regras mais complexas em função de outras regras mais simples, de modo a se obter a regra de transição.

A tabela de funções estáticas e derivadas é uma lista cujas entradas possuem três componentes: o nome da função, o tipo da função e a definição da função, como mostrado na Figura 4.4. O tipo da função de uma árvore cujos nós são os tipos definidos na linguagem, e cujo nó raiz é possivelmente o tipo funcional, exceção feita quando a função é 0-ária. Funções de alta ordem não são permitidas. De forma semelhante, a definição da função é uma árvore cujos nós são as expressões permitidas na linguagem. A definição de cada um dos tipos é feita na Seção 4.4.1, enquanto que a definição das expressões da linguagem é feita na Seção 4.4.1.

As entradas da tabela de funções dinâmicas possuem dois componentes. A diferença em relação às tabelas de funções estáticas e derivadas é que uma função dinâmica não possui uma expressão que a defina. Em seu lugar, esta possui um mapeamento que associa a pontos no domínio valores do conjunto imagem. A tabela de funções dinâmicas é ilustrada na Figura 4.4.

Como funções externas são definidas ulteriormente a uma especificação em linguagem MIR, as entradas da tabela de funções externas possuem apenas dois componentes: o nome da função e o seu tipo. Funções externas são escritas em C++, e o protocolo de comunicação, assim como a política de mapeamento dos tipos dos parâmetros são definidos na Seção 4.3.2.

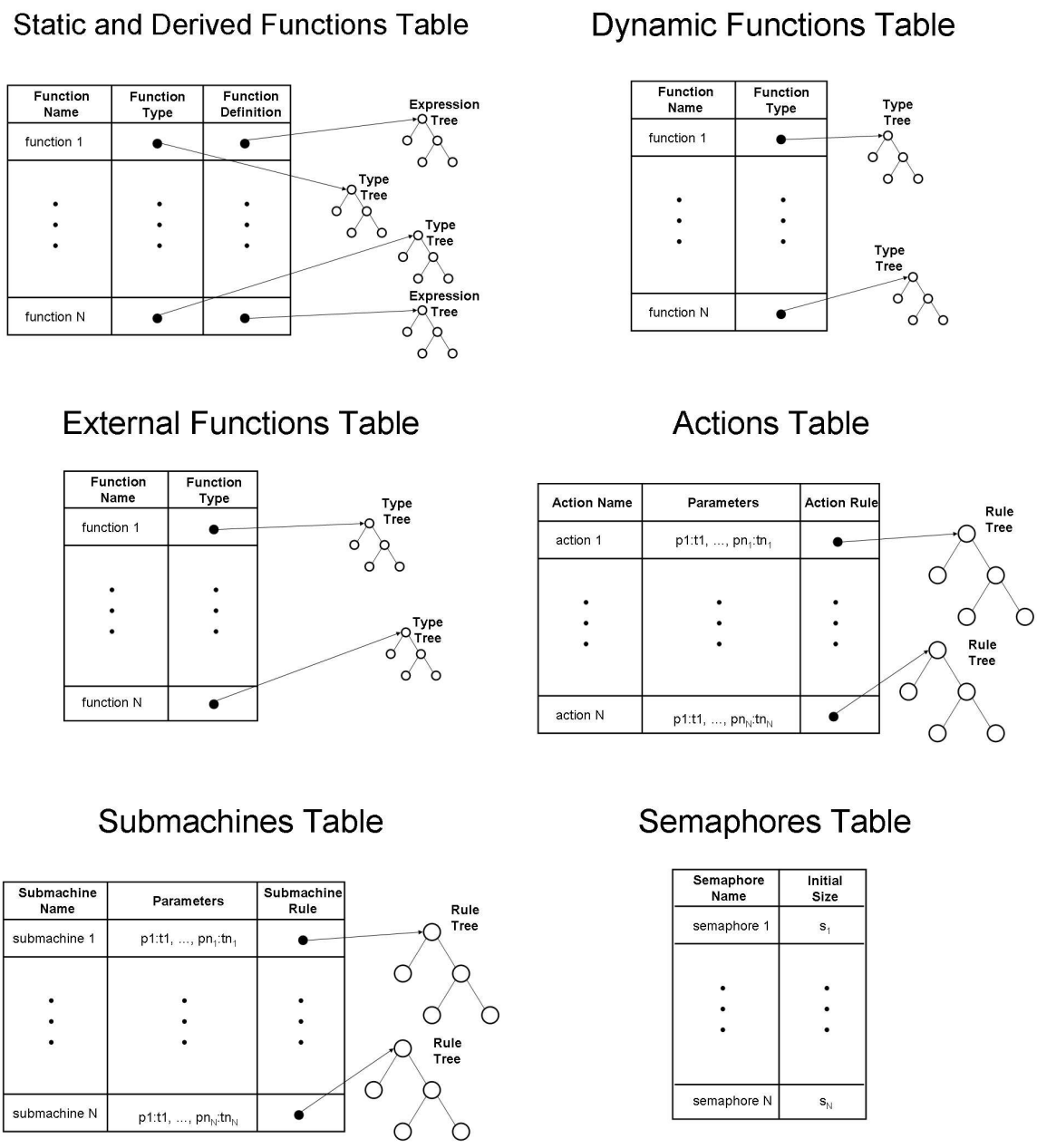

Figura 4.4: A estrutura das tabelas de um ambiente.

A tabela de funções externas é apresentada na Figura 4.4.

As tabelas de funções definem as funções utilizadas nas expressões da regras de inicialização e de transição do módulo. Esta utilização se dá por meio da chamada da função passando os seus argumentos, de modo a obter o seu valor naquele ponto. Adicionalmente, as funções dinâmicas também são alvo de atualizações.

As entradas da tabela de ações possuem três componentes: o nome da ação, a lista de parâmetros da ação, devidamente tipados, e a regra de transição que define ação. Esta também é a constituição da tabela de submáquinas. A diferença entre as duas esta presente na interpretação da regra de transição: enquanto na primeira a regra é executada uma vez no ponto de chamada, na segunda tem-se a execução cíclica desta regra.

As tabelas de ações e sub-máquinas definem as ações e sub-máquinas que são utilizadas nas regras de inicialização e transição do módulo. Seu uso se dá por meio da chamada destas abstrações passando-se os parâmetros adequados. Esta passagem de parâmetro pode ser por cópia ou por referência, e o modo exato de passagem é indicado para cada parâmetro quando da declaração da ação ou sub-máquina na tabela.

A tabela de semáforos armazena os semáforos daquele ambiente, os quais são especificados por um nome e um valor inicial. Estes semáforos definidos são usados na regra de transição do módulo. Para a utilização destes semáforos, existem as regras cuja semântica é a das operações fundamentais de um semáforo, como será mostrado adiante.

4.3.1 A Abordagem para a Implementação do Modelo de Concorrência

Na linguagem MIR, a especificação de um sistema concorrente se dá por meio do disparo simultâneo de agentes, cada qual com o seu tipo. O tipo de um agente é um módulo, e por isso um agente tem associado a si um conjunto de entidades locais, dito ambiente local, e um conjunto de entidades globais, compartilhadas por todos os agentes daquele módulo, dito ambiente global. Um ambiente é composto por funções estáticas, derivadas, dinâmicas e externas, e também por ações, submáquinas e semáforos. Além disso, o módulo de um agente fornece a este uma regra de transição a ser executada ciclicamente. Dizer que um agente foi disparado significa dizer que o agente foi criado e iniciou a execução de sua regra de transição, execução esta que acontece concorrentemente à execução dos demais agentes.

Agentes interagem entre si. Na forma da implementação realizada da linguagem, esta interação é feita por meio do acesso direto aos componentes dos ambientes globais e locais de cada agente. Em uma analogia com a programação orientada por objetos, um módulo pode ser visto como uma classe, agentes podem ser vistos como objetos destas classes associados a uma *thread*, o ambiente global são os componentes estáticos da classe, e o ambiente local são os componentes não-estáticos da classe. Um agente A qualquer pode avaliar a função $f : X \to Y$ presente no ambiente local do agente B, de tipo M, por meio de uma chamada semelhante a $B.f(x)$. Semelhantemente, uma função $g : W \to Z$ presente no ambiente global de B poderia ser chamada por meio de $B.g(z)$ ou ainda $M.g(z)$. O mesmo acesso se aplica aos outros componentes do ambiente, a saber, ações, submáquinas e semáforos.

Dentre as diversas modalidades de funções presentes no modelo, uma delas é de particular interesse para o modelo de concorrência: a modalidade das funções dinâmicas. Conforme visto anteriormente, funções dinâmicas podem sofrer atualizações, de modo que o valor de uma função pode ser alterado em pontos de seu domínio entre uma transição e outra. Estas funções estão para as máquinas de estado abstratas como as variáveis para a programação imperativa tradicional. Desta feita, pode-se dizer que estas funções são a *memória* dos agen-

tes em execução. Como as funções dinâmicas de um agente podem ser acessadas por outros agentes tanto para consulta como para atualização, estas funções caracterizam um modelo de *memória compartilhada*.

Todavia, o acesso aos componentes dos agentes se dá, em princípio, sem nenhum controle automático da disputa por um recurso comum. É possível, na linguagem proposta, escrever uma especificação cujo comportamento seja imprevisível, haja vista que não se sabe ao certo qual agente teria acesso primeiro a um recurso compartilhado. Por isso, um mecanismo para a sincronização deste acesso foi projetado como um recurso natural da linguagem. Este mecanismo é a especificação e o uso de semáforos para estabelecer uma política ordenada de acesso a recursos comuns. É responsabilidade do programador ASM especificar seus programas de modo a utilizar apropriadamente este mecanismo de sincronismo, assim como é responsabilidade de um programador de uma linguagem imperativa comum utilizar os mecanismos de sincronismo oferecidos pelas bibliotecas de concorrência disponíveis.

A título de exemplo deste mecanismo, vamos descrever, ainda em linguagem natural, como seria o acesso a um recurso comum. Suponha que existam dois agentes a e b, de tipos A e B, respectivamente, que acessem uma função dinâmica f de um módulo C. Uma forma de programar este acesso nos agentes a e b de modo a sistematizar o uso de f é definir um semáforo em C, digamos, s_f, com tamanho 1, e programar A e B de tal forma que todo acesso a f seria precedido por uma tentativa de capturar o semáforo s_f. Além disso, após o uso de f, s_f seria liberado. Este exemplo será retomado mais adiante quando tivermos definido a sintaxe da linguagem.

4.3.2 MIR Native Interface

A linguagem MIR permite a existência de *funções externas*, também conhecidas como *oráculos*, escritas como funções C++ (e não métodos). As funções externas estão presentes no modelo ASM desde o princípio e elas fornecem uma maneira de especificar a interação com o ambiente. Como argumentado por Gurevich [Gur95], uma função oráculo não precisa ser consistente em chamadas entre passos diferentes da execução de uma regra de transição, sendo adequadas para simular a entrada do usuário, por exemplo.

Para que funções escritas em C++ sejam chamadas adequadamente de dentro de um programa escrito na linguagem MIR, faz-se necessário que esta obedeça algumas convenções. Estas convenções são chamadas de *MIR Native Interface*, ou abreviadamente MNI, e determinam um protocolo comum ao qual tanto o programa em linguagem MIR quanto o oráculo em C++ devem obedecer. Este protocolo diz respeito a convenções sobre duas caracterísitas das funções externas: o tipo e o nome.

Nas funções externas escritas em C++ não é permitido `void` como tipo de retorno, pois tratam-se especificamente de *funções*, e não *procedimentos*, e por isso algum valor é esperado destas. Por outro lado, as funções podem ser 0-árias. O mapeamento dos tipos aceitáveis em linguagem MIR e seus correspondentes em C++ á apresentado na Tabela 4.1. Estes são os tipos permitidos tanto na assinatura de uma função externa quanto em seu retorno. Devido a restrições de implementação, o tipo do parâmetro deve ser um tipo básico ou uma lista de elementos dos tipos permitidos.

Quanto ao nome, se uma função externa de um módulo A é chamada de f, então o seu nome na implementação C++ deve ser `MNI__A__f`. Ou seja, a convenção do nome na implementação é `MNI__NomeDoMódulo__NomeDaFunção`. Isto evita conflito de nomes entre funções externas e também entre o código gerado a partir de uma especificação em linguagem MIR.

Tipo em linguagem MIR	Tipo C++ Correspondente
boolean	`bool`
character	`char`
integer	`int`
real	`double`
string	`std::string`
list of T	`std::vector<`T`>`

Tabela 4.1: Mapeamento de tipos previsto na *MIR Native Interface*. O prefixo `std::` significa que o elemento pertence à *Standard Template Library* de C++ [Str00].

4.4 A Linguagem para Representação de MIR

A unidade básica de compilação da linguagem MIR é o módulo. Cada módulo é representado por um arquivo de extensão `mod`, usando uma sintaxe baseada em XML, cuja estrutura é apresentada na Figura 4.6. A utilização de uma sintaxe *XML-like* se justifica pelos seguintes motivos:

- A linguagem proposta é uma representação *intermediária*, e por isso é necessário que seja de fácil interpretação. Analisadores sintáticos para documentos XML e outras ferramentas de suporte estão disponíveis em grande escala, o que facilita a manipulação de texto XML. Além disso, este formato tem sido utilizado[1] com sucesso em uma ampla gama de contextos como representação persistente de dados dos mais variados tipos.
- Ao mesmo tempo, é desejável que programas na linguagem MIR sejam compreensíveis ao ser humano, e esta é uma característica intrínseca de documentos XML, que carregam, junto com a informação representada, a meta-informação necessária ao entendimento.
- Um arquivo XML é, basicamente, um arquivo texto, de modo que pode ser lido e editado em qualquer plataforma sem dificuldades adicionais. Embora editores especializados existam para facilitar o trabalho com arquivos XML, em último caso editores de texto simples, como o *vi* ou o *notepad*, podem ser usados.
- A estrutura inerente de um documento XML é apropriada para o tipo de informação que se deseja representar neste contexto, a saber, listas e árvores que representam uma especificação ASM.

A marca[2] raiz de um arquivo de extensão `mod` é a marca `<module>`. O conteúdo desta marca é um módulo, que agrupa o seu nome, seu ambiente global comum a todos os agentes do módulo, as definições dos componentes locais a cada agente do módulo, e também a sua regra de transição. A Seção 4.4.1 define com exatidão a sintaxe de cada uma das marcas da linguagem.

Para a completa implementação do modelo, não basta que módulos sejam compilados, é necessária a instanciação de agentes que executam estes módulos. Um módulo é, *a priori*,

[1] Apenas para citar alguns exemplos, XML é usado em diversos arquivos de configuração em servidores J2EE, em arcabouços como o Struts e como representação universal em programas para desenho UML, como o Poseidon, o ArgoUML e o Rational Rose.

[2] marca, do inglês *tag*.

semelhante a uma classe em programação orientada por objetos, assumindo o papel de uma forma(ô) a partir da qual os agentes são criados. Agentes construídos a partir de um mesmo módulo possuem cópias individuais de seus componentes locais, assim como objetos de uma mesma classe possuem cópias individuais dos membros não-estáticos da classe. Os componentes globais de um módulo são compartilhados pelos agentes deste módulo, à semelhança dos membros estáticos de uma classe.

A instanciação inicial de agentes é definida por meio de um arquivo especial, de extensão `mas`, um acrônimo para *MIR Agents Starter*. Basicamente, este arquivo define agentes por meio de pares *(nome* × *módulo)*, obedecendo à sintaxe XML definida na Figura 4.35.

4.4.1 Marcas da linguagem MIR

Nesta seção, é apresentada a sintaxe da linguagem MIR. Para tanto, faz-se o uso de figuras que seguem uma notação simples, que porém merece ser esclarecida. Esta notação é ilustrada na Figura 4.5. A marca raiz de um trecho qualquer representado em uma figura fica à esquerda da figura e é assinalada de cinza, como a marca `<raiz>` na Figura 4.5. Cada marca é representada por um retângulo contendo o nome da marca, e sua composição pode ser uma seqüência ou uma escolha. Um exemplo de seqüência é a marca `<sequencia>`, cujos componentes obrigatórios são, nesta ordem, `<elemento_1>`, `<elemento_2>` e `<elemento_3>`. O indicador de que trata-se de uma seqüência é o elemento de ligação entre a marca `<sequencia>` e seus componentes, a saber, um nó com três quadrados em seqüência. Por outro lado, uma escolha é ilustrada pela marca `<escolha>`, que oferece as alternativas `<alternativa_1>`, `<alternativa_2>` e `<alternativa_3>`. Isto significa que, em um arquivo XML validado por este esquema, a marca `<escolha>` contém em seu interior uma e apenas uma dentre as três alternativas oferecidas. A característica de escolha é representada pelo nó de ligação entre a marca `<escolha>` e suas alternativas, a saber, um nó contendo três quadrados dispostos um sobre o outro. As marcas podem também conter indicação de cardinalidade. A marca `<um_ou_mais>` é um exemplo de uma marca que deve aparecer ao menos uma vez no lugar onde ela é definida. A linha dobrada indica que o limite superior é ilimitado, como em `<um_ou_mais>` e `<zero_ou_mais>`. Nesta última, como o limite inferior é zero, as linhas são tracejadas. A marca `<opcional>` ilustra um elemento que pode ou não aparecer no ponto de sua definição. As convenções de cardinalidade também podem ser aplicadas a seqüências e alternativas, como ilustrado em `<sequencia_com_cardinalidade>` e `<alternativa_com_cardinalidade>`.

Definição de um módulo

Todo documento XML deve possuir uma marca raiz, que marca o início da estrutura hierárquica representada. Para o arquivo de serialização de um módulo MIR, esta marca é a marca `<module>`. Um módulo é composto pelo seu nome; pela lista de referências, agrupadas sob a marca `<references>`; pela lista de importações, agrupadas sob a marca `<imports>`; pelas tabelas de funções, abstrações de regras e semáforos de escopo global, agrupadas sob a marca `<global>`; pela tabelas de funções, abstrações de regras e semáforos de escopo local, agrupadas sob a marca `<local>`; pela regra de inicialização, sob a marca `<init>`; e por uma regra de transição, sob a marca `<transition>`. Estes elementos devem ser declarados nesta ordem, conforme pode ser visto na Figura 4.6. A Figura 4.7 apresenta um pequeno exemplo da definição de um módulo.

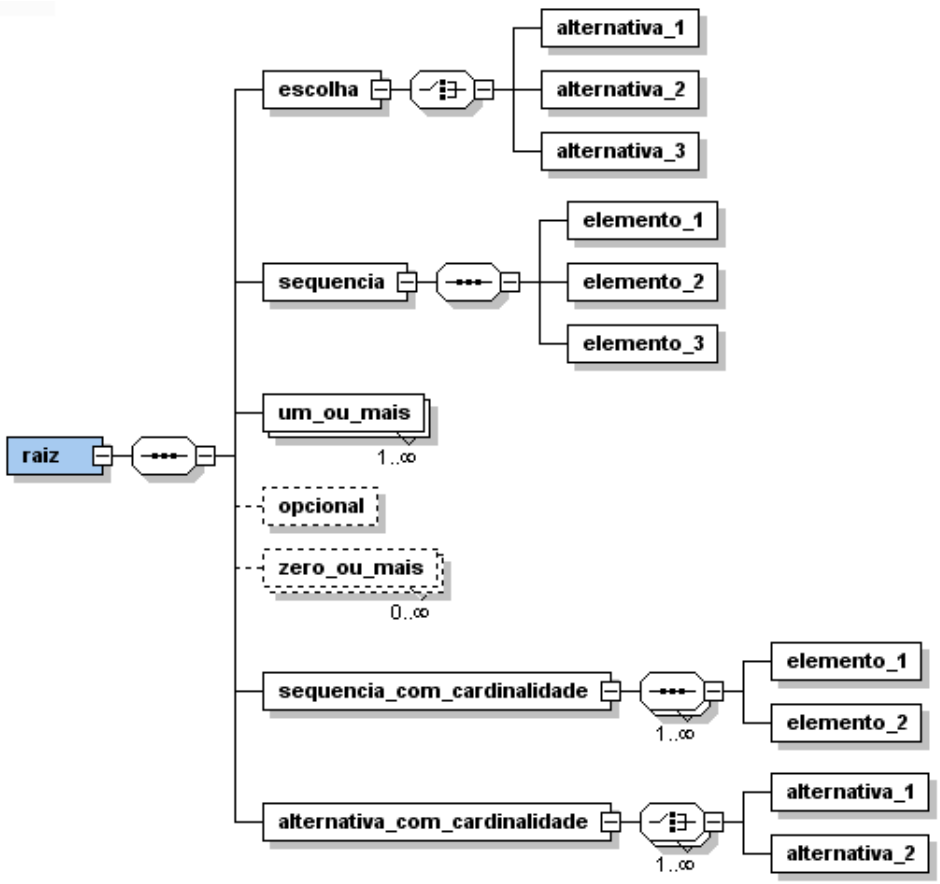

Figura 4.5: Exemplo do uso da notação visual para esquemas XML.

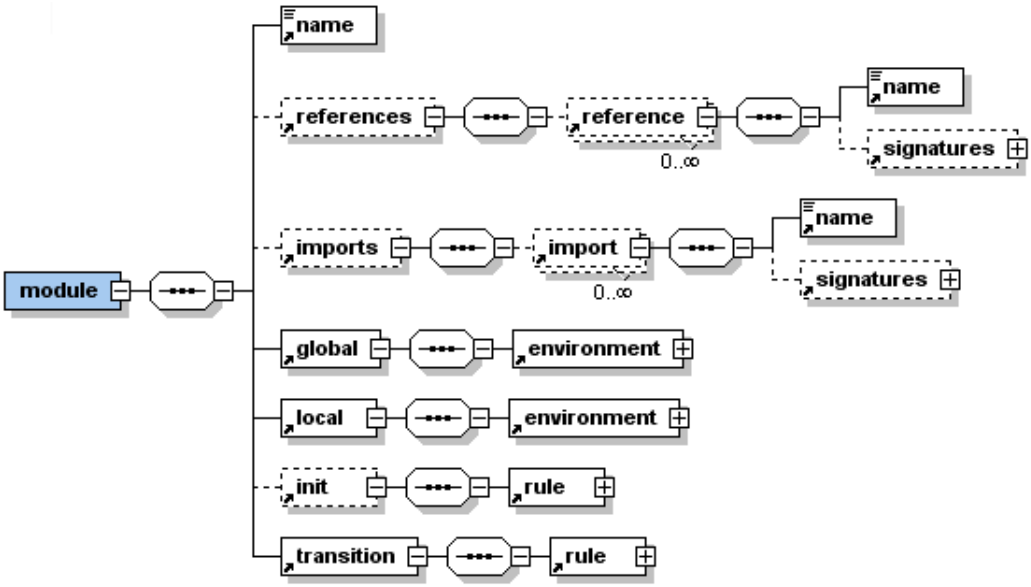

Figura 4.6: Estrutura da marca `<module>`.

Nome: A marca `<name>` representa uma cadeia de caracteres começadas por uma letra, seguida por letras, números e *underscores* em qualquer quantidade. Não são permitidos *underscores* duplos. No exemplo da Figura 4.7, o nome do módulo é `foo`.

Referências de Módulos: Os módulos referenciados são listados na marca `<references>`. Um módulo precisa ser referenciado quando o conhecimento de sua existência, e possivelmente da sintaxe de algum de seus componentes, se faz necessário em algum ponto do módulo referenciador. Um exemplo desta necessidade é a criação de agentes do tipo do módulo referenciado na regra `<create>` do módulo referenciador. Outro exemplo é o acesso de uma função dinâmica de um módulo ou agente externo ao módulo referenciador. Cada referência é feita por meio da marca `<reference>`, que contém obrigatoriamente o nome do módulo referenciado e opcionalmente as assinaturas dos componentes referenciados. As assinaturas possuem a sintaxe adequada ao tipo do elemento referenciado, conforme mostrado na Figura 4.8. No exemplo da Figura 4.7, o módulo definido referencia um outro módulo de nome `goo` e acessa uma função estática 0-ária de nome `gooFun` e tipo de retorno inteiro.

Importações de Módulos: Os módulos importados para a definição de um outro módulo são agrupados sob a marca `<imports>`. Cada importação é uma marca `<import>` que contém o nome do módulo a ser importado, e implica que todo este módulo importado será incluído como parte da definição do módulo importador, de forma semelhante a uam composição de classes em lingaugens orientadas por objetos. Cada importação contém obrigatoriamente o nome do módulo referenciado e opcionalmente as assinaturas dos componentes referenciados. As assinaturas possuem a sintaxe adequada ao tipo do elemento referenciado, conforme mostrado na Figura 4.8. No exemplo da Figura 4.7, o módulo definido importa um módulço de nome `doo` e acessa uma de suas funções dinâmicas, i.

```
<module>
  <name>foo</name>
  <references>
    <reference>
      <name>goo</name>
      <staticfunctionsignature>
        <name>gooFun</name>
        <type><integer/></type>
      </staticfunctionsignature>
    </reference>
  </references>
  <imports>
    <import>
      <name>doo</name>
      <dynamicfunctionsignature>
        <name>i</name>
        <type><integer/></type>
      </dynamicfunctionsignature>
    </import>
  </imports>
  <global>
    <environment>
       ...
    </environment>
  </global>
  <local>
    <environment>
       ...
    </environment>
  </local>
  <init>
    <update>
      <dynamicfunctioncall>
        <name>i</name>
      </dynamicfunctioncall>
      <expression>
        <literalinteger>1</literalinteger>
      </expression>
    </update>
  </init>
  <transition>
    ...
  </transition>
</module>
```

Figura 4.7: Um exemplo de um módulo

Ambiente global de nomes: A marca `<global>` contém um `<environment>`, definido a seguir. Sua função é representar a contribuição do módulo ao espaço de nomes globais, a saber, funções, ações, submáquinas e semáforos que são comuns a todos os agentes de um módulo. Em uma analogia com a orientação por objetos, se os módulos são as classes e os agentes são objetos destas classes, então os componentes globais são os membros estáticos das classes associadas aos módulos. Um exemplo de definição de um ambiente á apresentado na Figura 4.9, e é discutido mais adiante.

Ambiente local de nomes: A marca `<local>` contém um `<environment>`, definido a seguir. Sua função é representar os elementos internos aos agentes. Ainda na analogia com a orientação por objetos, os componentes locais são equivalentes aos membros não-estáticos das classes associadas aos módulos.

Regra de inicialização: A regra de inicialização, opcional, é definida dentro da marca `<init>`. Esta regra é executada uma única vez, antes da execução cíclica da regra de transição, e tem por função efetuar as atualizações iniciais necessárias em uma definição de um módulo. No exemplo da Figura 4.7, a regra de inicialização atribui o valor 1 a uma função dinâmica 0-ária de nome i.

Regra de transição: A regra de transição do módulo é definida sob a marca `<transition>`, cujo conteúdo é uma regra, definida pela marca `<rule>`. Esta regra é executada ciclicamente, até que seu término seja indicado por uma regra `<stop>`. As regras de transição são discutidas e exemplificadas mais à frente.

Ambiente de Nomes

Um ambiente de nomes, definido pela marca `<environment>`, nada mais é do que um agrupamento das coleções de funções, ações, submáquinas e semáforos de um módulo. Ou seja, um `<environment>` é uma coleção de entidades nomeáveis do módulo. Seu escopo pode ser tanto global quanto local, dependendo do contexto onde esta marca se encontre. Seu conteúdo, conforme mostrado na Figura 4.10, é a tabela de funções estáticas e derivadas, a tabela de funções dinâmicas, a tabela de funções externas, a tabela de ações, a tabela de submáquinas e a tabela de semáforos, todas opcionais. A definição destas tabelas é feita a seguir. A Figura 4.9 apresenta um exemplo de ambiente. Neste exemplo, estão ilustradas as tabelas que compõem um ambiente, omitindo-se detalhes da descrição de cada elemento que popula cada tabela. Estes elementos serão descritos mais adiante.

Tabela de funções estáticas e derivadas: A marca `<staticandderivedfunctions>` corresponde à tabela de funções estáticas e derivadas. As entradas desta tabela podem ser funções estáticas ou derivadas, em qualquer quantidade, conforme apresentado na Figura 4.11. Uma função estática é representada pela marca `<staticfunction>`. Seus componentes são, nesta ordem: o nome da função, o tipo da função e a expressão que denota a função. O tipo da função é provavelmente o tipo funcional, representado pela marca `<functional>`, à exceção de quando a função é 0-ária. Não são permitidas funções de alta-ordem. Além disso, não é permitido a chamada de funções derivadas, dinâmicas ou externas na definição de funções estáticas, pois estas devem permancer imutáveis ao longo de toda a execução da especificação ASM. A representação de uma função derivada, feita por meio da marca `<derivedfunction>`, é a mesma de uma função estática.

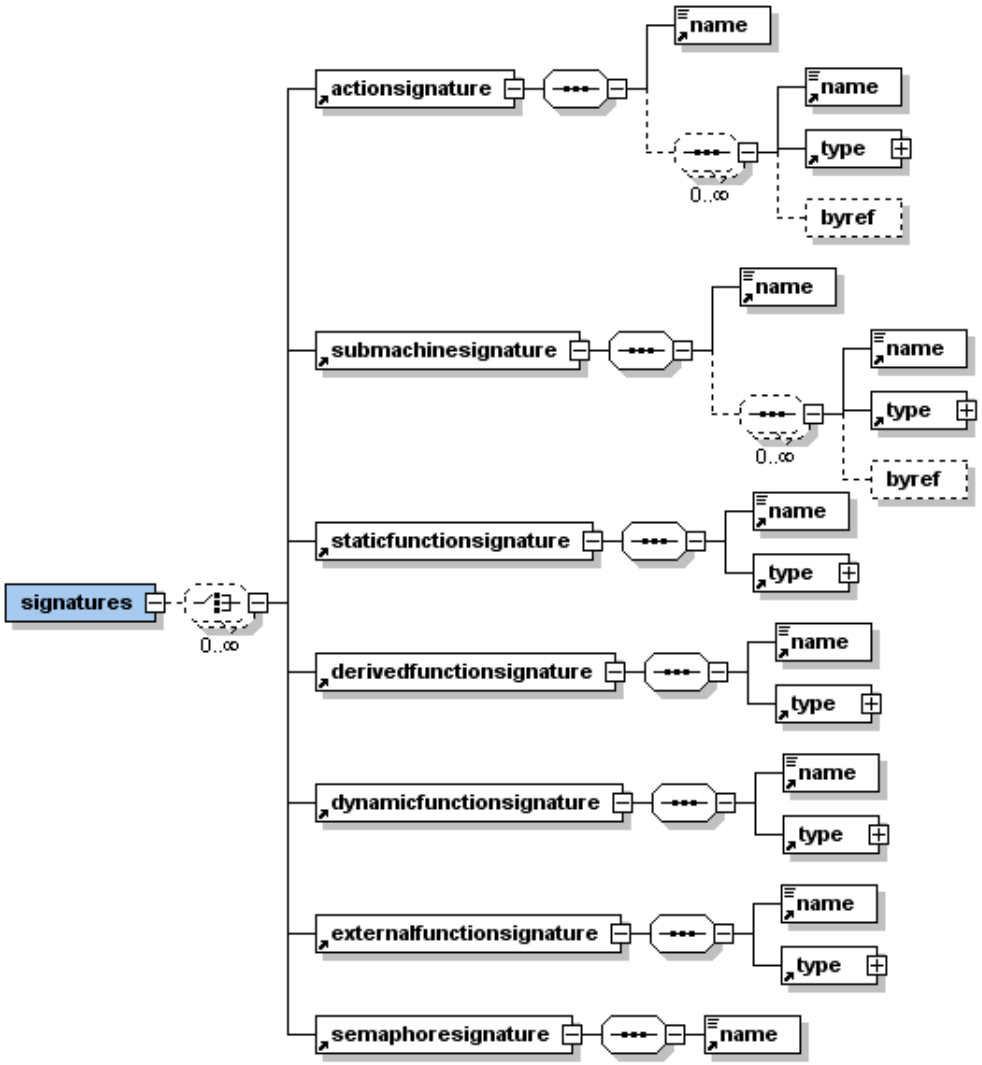

Figura 4.8: Estrutura da marca `<signatures>`.

```
<environment>
  <staticandderivedfunctions>
    <staticfunction>
      ...
    </staticfunction>
    ...
    <derivedfunction>
      ...
    </derivedfunction>
    ...
  </staticandderivedfunctions>
  <dynamicfunctions>
    <dynamicfunction>
      ...
    </dynamicfunction>
    ...
  </dynamicfunctions>
  <externalfunctions>
    <externalfunction>
      ...
    </externalfunction>
    ...
  </externalfunctions>
  <actions>
    <action>
      ...
    </action>
    ...
  </actions>
  <submachines>
    <submachine>
      ...
    </submachine>
    ...
  </submachines>
  <semaphores>
    <semaphore>
      ...
    </semaphore>
    ...
  </semaphores>
</environment>
```

Figura 4.9: Um exemplo de um ambiente

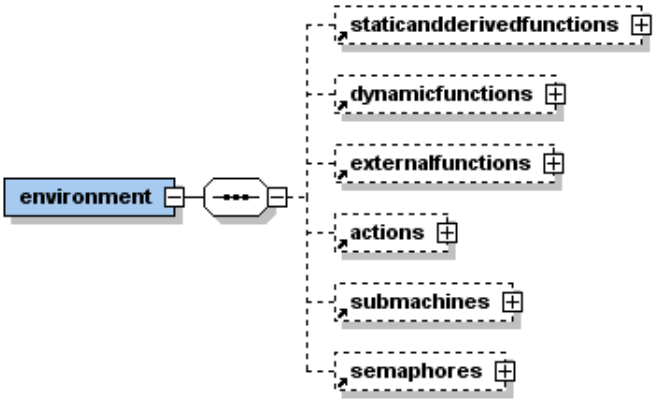

Figura 4.10: Estrutura da marca `<environment>`.

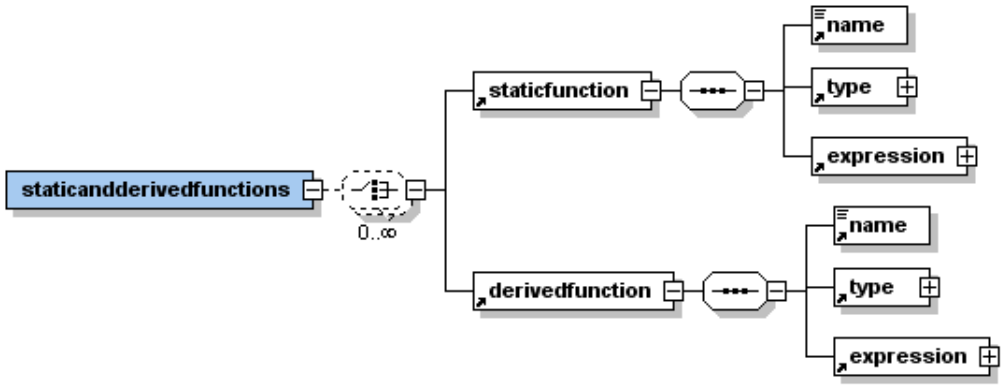

Figura 4.11: Estrutura da marca `<staticandderivedfunctions>`.

A diferença é que, semanticamente, uma função derivada pode conter consultas a funções externas ou dinâmicas, enquanto que isto é vedado a funções estáticas. Isto significa que funções derivadas não são garantidamente imutáveis ao longo da execução de uma especificação ASM.

Tabela de funções dinâmicas: A marca `<dynamicfunctions>` corresponde à tabela de funções dinâmicas. As entradas desta tabela são funções dinâmicas, em qualquer quantidade, conforme apresentado na Figura 4.12. Uma função dinâmica é representada por seu nome e seu tipo, que devem aparecer nesta ordem dentro da marca `<dynamicfunction>`. Opcionalmente, pode-se definir um valor *default* para a função dinâmica, definido por uma expressão após o tipo da função. Não há necessidade da definição do seu corpo, pois funções dinâmicas são definidas por meio de atualizações em tempo de execução. Funções de mais alta ordem não são permitidas.

Tabela de funções externas: A marca `<externalfunctions>` corresponde à tabela de funções externas. As entradas desta tabela são funções externas, em qualquer quantidade, conforme apresentado na Figura 4.13. Uma função externa também é repre-

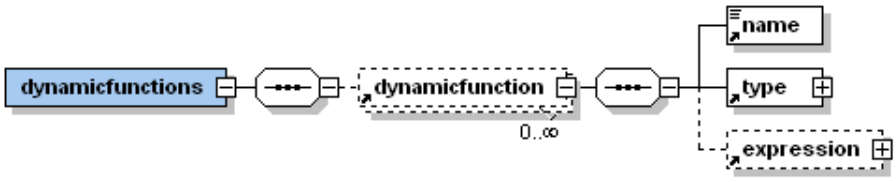

Figura 4.12: Estrutura da marca `<dynamicfunctions>`.

Figura 4.13: Estrutura da marca `<externalfunctions>`.

sentada por seu nome e seu tipo, que devem aparecer nesta ordem dentro da marca `<externalfunction>`. A definição de seu corpo não é necessária neste ponto, pois, como o próprio nome indica, esta é externa à especificação. Mais uma vez, funções de mais alta ordem não são permitidas.

Tabela de ações: A marca `<actions>` corresponde à tabela de ações. As entradas desta tabela são ações, representadas pela marca `<action>`, em qualquer quantidade, conforme apresentado na Figura 4.14. Uma ação, definida pela marca `<action>`, possui um nome, uma lista de parâmetros e uma regra de transição. Opcionalmente, cada parâmetro pode ser marcado com um indicador de que a passagem é por referência, sendo que na ausência deste indicador a passagem é feita por cópia. Uma ação apenas nomeia e parametriza uma determinada regra, de modo que esta pode ser definida uma vez e então utilizada em vários pontos da definição de outra regra, de forma equivalente a uma substituição textual. Não há execução cíclica da ação no ponto de chamada.

Tabela de submáquinas: A marca `<submachines>` corresponde à tabela de submáquinas. As entradas desta tabela são submáquinas, representadas pela marca `<submachine>`, em qualquer quantidade, conforme apresentado na Figura 4.15. A marca `<submachine>`

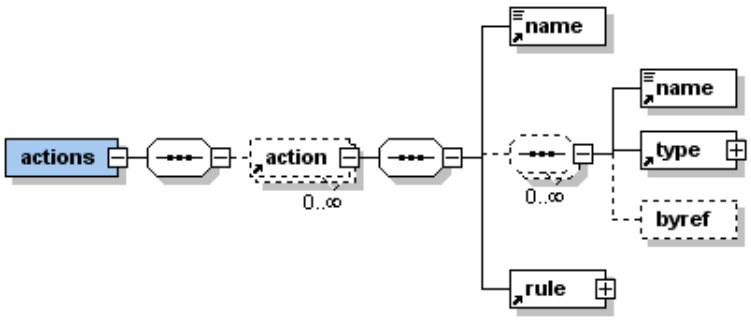

Figura 4.14: Estrutura da marca `<actions>`.

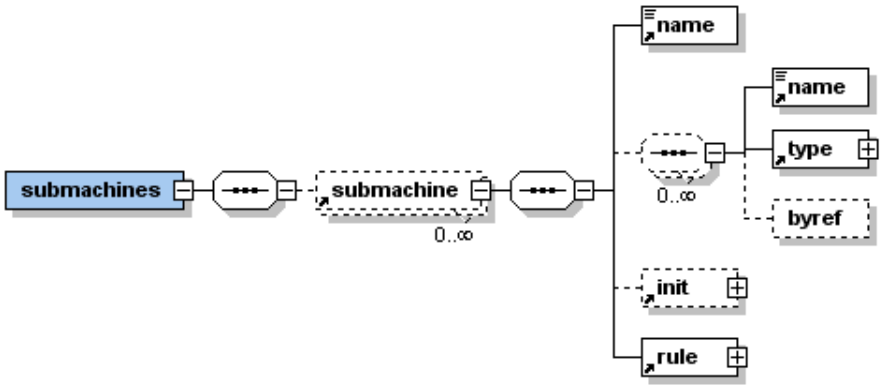

Figura 4.15: Estrutura da marca `<submachines>`.

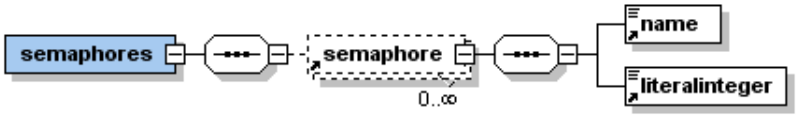

Figura 4.16: Estrutura da marca `<semaphores>`.

define uma outra forma de se construir abstrações de regras. Uma abstração do tipo `<submachine>` implementa a idéia de *submáquina*: a regra de transição é executada repetidamente até que uma regra `<return>` é encontrada. Uma submáquina é definida por meio de um nome, uma lista de parâmetros, uma regra de inicialização e uma regra de transição. Opcionalmente, cada parâmetro pode ser marcado com um indicador de que a passagem é por referência, sendo que na ausência deste indicador a passagem é feita por cópia. No ponto de chamada de uma submáquina, a regra de inicialização é executada uma vez, seguida da execução cíclica da regra de transição.

Tabela de semáforos: Semafóros são definidos dentro da marca `<semaphores>`. Um semáforo, representado pela marca `<semaphore>`, é composto por um nome e por um inteiro, que indica o valor inicial do semáforo.

Regras

A marca `<rule>` representa uma regra na linguagem intermediária. O conteúdo desta marca pode ser uma dentre vinte e uma opções, divididas em dois grupos, a saber:

- regras simples, que são `<update>`, `<immediateupdate>`, `<create>`, `<dispatch>`, `<destroy>`, `<stop>`, `<return>`, `<actioncall>`, `<submachinecall>`, `<abstractioncall>`, `<lambda>`, `<reset>`, `<signal>` e `<wait>`;
- regras construídas a partir de outras regras, que são `<conditional>`, `<forall>`, `<choose>`, `<let>`, `<case>`, `<with>` e `<block>`.

Estas regras são apresentadas na Figura 4.17. Todas estas marcas são detalhadas mais adiante. Além disso, a marca `<rule>` possui dois componentes opcionais. O primeiro, chamado `<label>`,

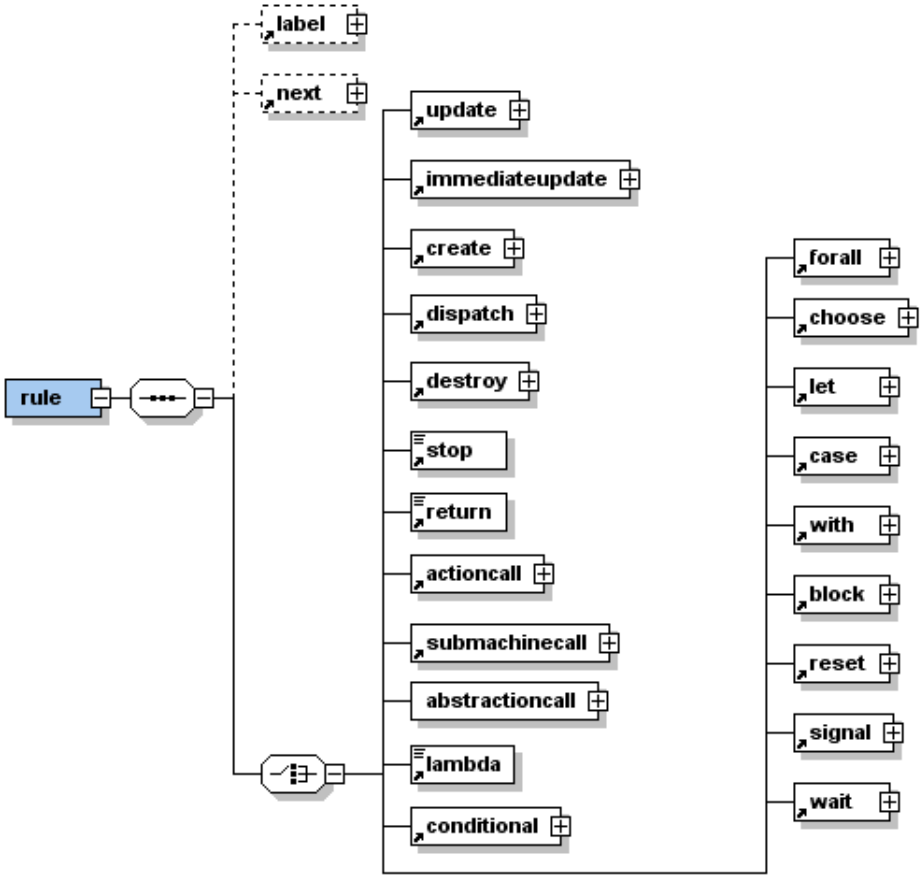

Figura 4.17: Estrutura da marca `<rule>`.

nomeia um elemento `<rule>` unicamente por meio de um inteiro. O segundo parâmetro é chamado de `<next>`, e define qual é o nó por onde a próxima iteração da execução da regra de transição deve se iniciar caso a execução da iteração corrente termine nesta regra. Caso não haja esta informação, assume-se o comportamento padrão que o próximo passo da execução se inicia pelo nó raiz da regra de transição. Este mecanismo permite a realização de otimizações de fluxo de controle.

Regras Simples As regras simples são apresentadas na Figura 4.18. Exemplos destas regras são apresentados na Figura 4.20. Uma atualização de função dinâmica é representada pela marca `<update>`. Seus componentes são uma chamada a uma função dinâmica e uma expressão. Ressalta-se que embora sintaticamente a mesma marca `<dynamicfunctioncall>` seja utilizada tanto neste contexto quanto no contexto de expressões, a sua semântica é um pouco diferente. No contexto de expressões, esta marca tem o significado de *right value*, ou seja, representa o valor armazenado em um ponto de uma função dinâmica. Já no contexto de atualização, a marca `<dynamicfunctioncall>` representa um *location*, ou seja, tem o significado de *left value*. A expressão fornecida denota o valor a ser armazenado nesta *location*. Atualizações definidas por meio da marca `<update>` são inseridas na lista de atualizações construída e processada a cada iteração.

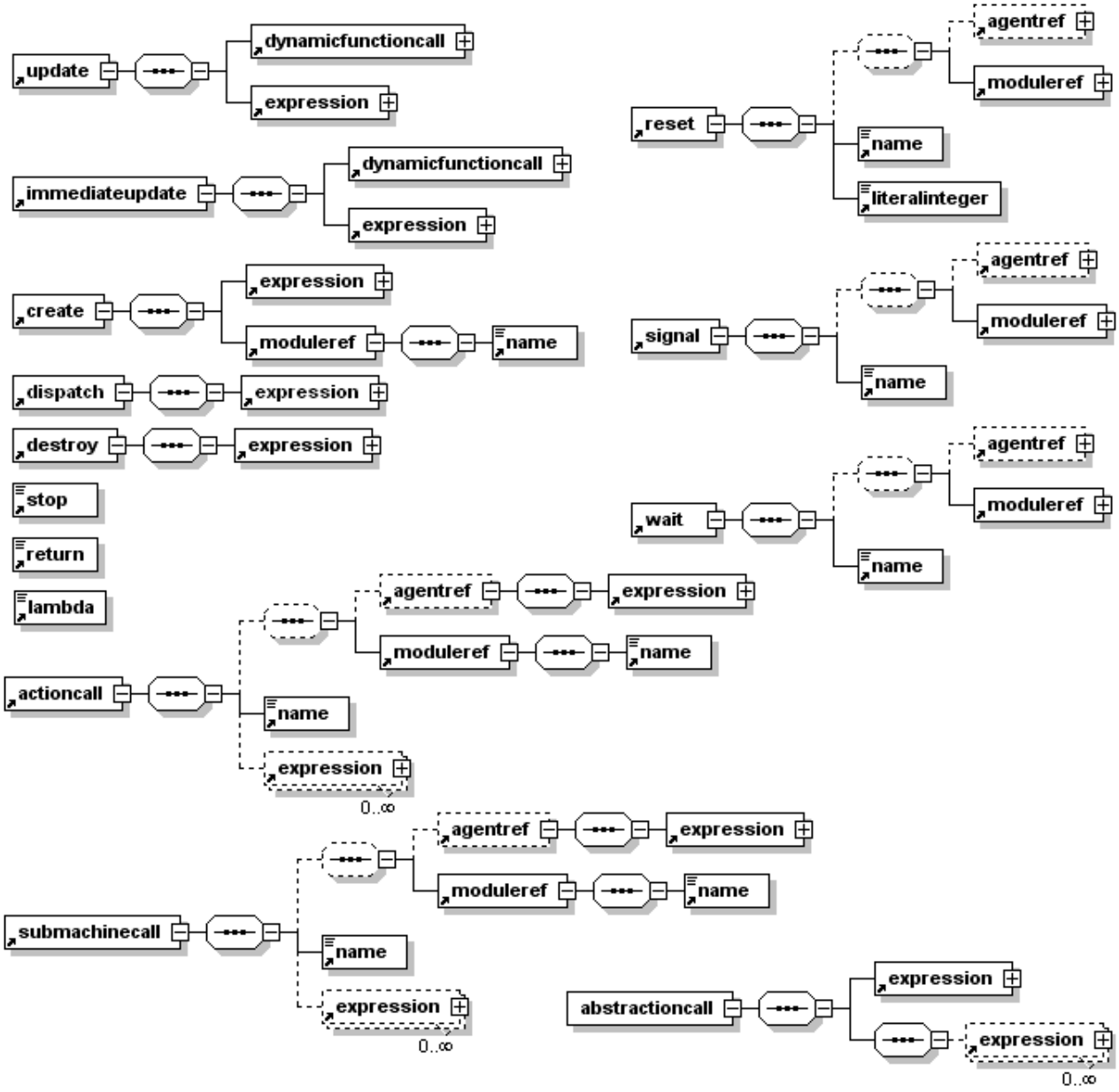

Figura 4.18: As regras simples.

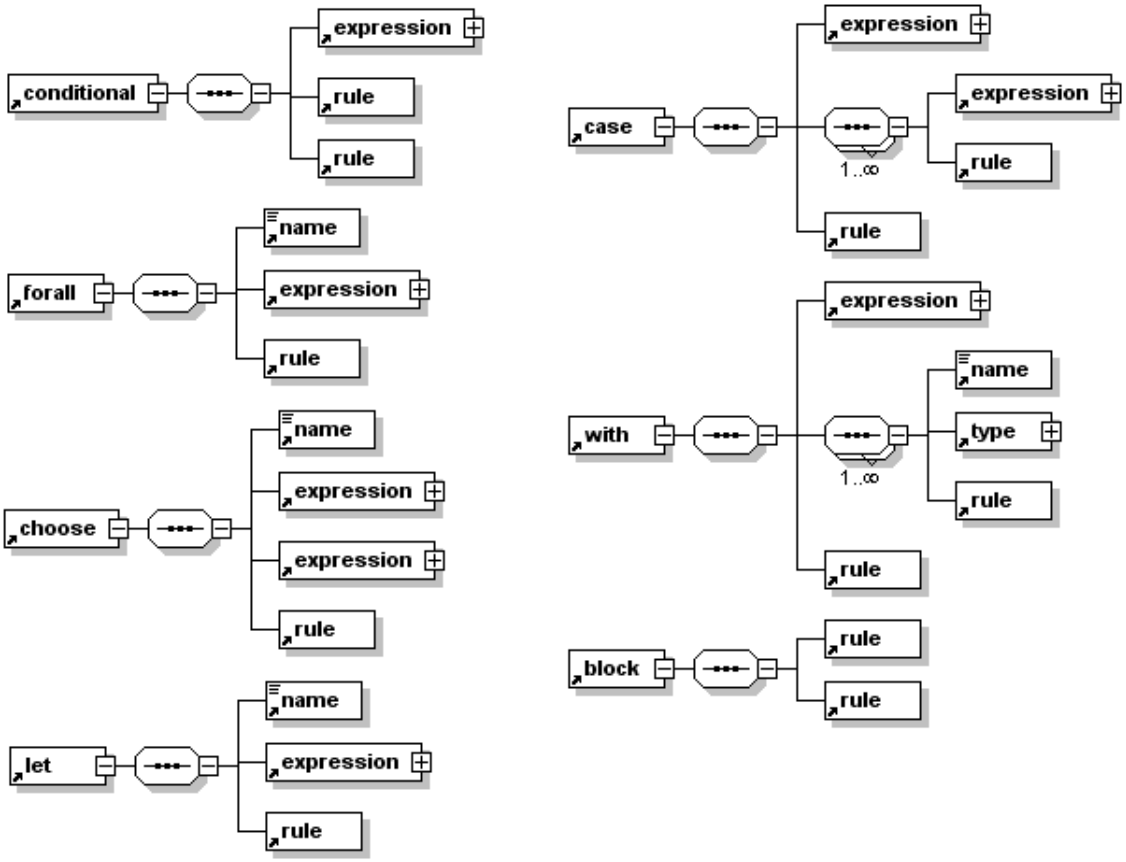

Figura 4.19: As regras compostas.

```
<update>
  <dynamicfunctioncall>
    <name>f</name>
    <expression><literalinteger>0</literalinteger></expression>
  </dynamicfunctioncall>
  <expression><literalinteger>1</literalinteger></expression>
</update>
<immediateupdate>
  <dynamicfunctioncall>
    <name>f</name>
    <expression><literalinteger>0</literalinteger></expression>
  </dynamicfunctioncall>
  <expression><literalinteger>1</literalinteger></expression>
</immediateupdate>
<create>
  <expression><literalstring>"umAgente"</literalstring></expression>
  <moduleref><name>UmModulo</name></moduleref>
</create>
<dispatch>
  <expression><literalstring>"umAgente"</literalstring></expression>
</dispatch>
<destroy>
  <expression><literalstring>"umAgente"</literalstring></expression>
</destroy>
<stop/>
<return/>
<actioncall>
  <name>UmaAcao</name>
  <expression><literalstring>"Olá Mundo!"</literalstring></expression>
</actioncall>
<submachinecall>
  <name>UmaAcao</name>
  <expression><literalstring>"Olá Mundo, várias vezes!"</literalstring></expression>
</submachinecall>
<lambda/>
<reset>
  <name>UmSemáforo</name>
  <literalinteger>1</literalinteger>
</reset>
<signal>
  <name>UmSemáforo</name>
</signal>
<wait>
  <name>UmSemáforo</name>
  <literalinteger>1</literalinteger>
</wait>
```

Figura 4.20: Exemplos de regras simples.

A marca <immediateupdate> também representa uma atualização de uma função dinâmica, e por isto sua estrutura é a mesma da marca <update>. Entretanto, atualizações definidas por meio desta marca são processadas imediatamente, ao contrário de serem inseridas em uma lista de atualizações. Esta marca deve ser usada com cautela, pois ela pode gerar efeitos colaterais que desrespeitam o modelo de máquinas de estado abstratas. A sua existência se justifica pelo seu uso em otimizações de atualizações.

A marca <create> representa a criação de um agente, e é constituída do nome deste agente e de uma referência ao módulo que o agente deve executar. O nome do módulo é fixo, sendo conhecido em tempo de compilação, ao passo que o nome do agente é determinado pela avaliação de uma expressão em tempo de execução. Depois de criado, o agente é disparado por meio de <dispatch>, que recebe como argumento o nome do agente a ser disparado. A separação entre o momento da criação e o momento do disparo é importante, pois em certos contextos pode ser necessário garantir que as estruturas associadas a um agente sejam criadas antes que outro agente que as utiliza seja disparado. A destruição de um agente de nome dado é feita por meio da marca <destroy>, que recebe o nome do agente a ser destruído.

A execução de um agente é interrompida por meio da marca <stop>, que não possui componentes. Já a indicação fim da execução de uma ação ou de uma submáquina, e conseqüentemente o retorno do controle à regra principal, é feito por meio da marca <return>, que não possui componentes. Em uma ação, esta marca não é necessária, pois o controle volta ao ponto de chamada assim que a execução da regra é concluída. Cabe lembrar que um agente pode interromper um outro agente por meio da regra <destroy>.

A chamada de uma ação é feita por meio da marca <actioncall>, que contém o nome da ação a ser executada e a lista de expressões que são associadas aos parâmetros da ação. Além disso, opcionalmente, uma ação possui uma referência a um agente e a um módulo. Se apenas a referência ao módulo é existente, então esta ação deve pertencer ao escopo global do módulo referenciado. Se ambas as referências estão presentes, então a ação pode pertencer ao escopo local, e é associada ao agente referenciado, que deve ser do tipo indicado na referência de módulo. Quanto à passagem de parâmetros, o modo como esta acontece é determinado pela declaração da ação. Se na declaração o parâmetro foi marcado como byref, então o parâmetro é de entrada e saída. Caso contrário, ele é somente de entrada. A chamada de uma submáquina é feita por meio da marca <submachinecall>, cuja sintaxe é igual à chamada de uma ação. O modo da passagem de parâmetros também é determinado pela marcação dos parâmetros formais na declaração da abstração, à semelhança das ações. A marca <abstractioncall> é semelhante às marcas <actioncall> e <submachinecall>. A diferença é que enquanto nestas duas a abstração a ser invocada é dada pelo seu nome qualificado, na marca <abstractioncall> a abstração é dada pela avaliação da primeira expressão, que deve resultar em um valor do tipo <abstraction>, que é uma referência a uma abstração qualquer.

A marca <lambda> representa uma regra que não faz nada. A sua função é servir como marcador em regras do tipo *if*, *case*, etc., onde certas avaliações de guardas devem ter a possibilidade de resultar em uma ação nula, mas sintaticamente a especificação de uma regra é necessária. Esta marca não possui componentes.

As regras <reset>, <wait> e <signal> são as regras que cuidam da utilização dos semáforos de um módulo. A regra <reset> restaura o valor de um semáforo cujo nome é indicado em <name> para o valor representado pelo inteiro em <literalinteger>. Já a regra <wait> coloca o agente em espera pelo semáforo indicado nesta, caso o valor deste seja igual a zero, ou decrementa o valor do semáforo em uma unidade, caso contrário. Finalmente, a regra <signal> sinaliza que o valor do semáforo especificado deve ser incrementado em uma unidade ou que um

agente na espera deve ser acordado, caso haja agentes na espera. Todas estas regras possuem, opcionalmente, uma referência a um módulo e a um agente, à semelhança das chamadas a ações e a submáquinas.

A Figura 4.21 apresenta um exemplo do uso de um semáforo para controlar o acesso a uma função dinâmica f. Para tal, o acesso é feito em três passos, devidamente guardados dentro da regra de transição por uma regra *case* ou por regras condicionais. No passo 0, o agente, por meio da regra `<wait>`, tenta obter acesso à função dinâmica consultando o semáforo `sem`, criado para ser utilizado neste controle de acesso. Este semáforo foi inicializado com o valor 1 no momento de sua declaração. Se no momento em que o `<wait>` é executado o semáforo vale 1, este é decrementado em uma unidade, assumindo o valor 0. Caso contrário, o agente fica bloqueado (sem espera ocupada) até que algum outro agente sinalize sua liberação. Quando o agente bloqueado é liberado, ele passa para o passo 2, onde pode fazer uso da função dinâmica f. Finalmente, quando ele termina de usar a função, o agente passa para o passo 3, onde o uso da função é explicitamente liberado por meio da execução da regra `<signal>` sobre o semáforo `sem`. Cabe ressaltar que não existe uma amarração explícita entre a exclusão mútua no acesso a f e a declaração do semáforo `sem`. Este comportamento deve ser programado pelo programador do módulo. Conforme dito anteriormente, o modelo é genérico o bastante para permitir acessos não-controlados, se esta for a vontade do programador.

Regras Compostas As regras compostas, conforme já foi dito, são regras construídas a partir de outras regras, e são apresentadas na Figura 4.19. As Figuras 4.22 e 4.23 apresentam alguns exemplos de regras compostas.

A marca `<conditional>` representa um *if-then-else* para regras de transição. Ela é composta por uma expressão e duas regras. A expressão deve denotar um valor do tipo booleano, que determina quais das duas regras é executada. Se o valor da expressão for verdadeiro, a primeira regra é executada; caso contrário, a segunda regra será executada.

A marca `<forall>` é composta por um nome, uma expressão e uma regra. A expressão é um agregado (conjunto ou lista), e a semântica desta construção é executar a regra de transição dada para cada elemento do conjunto, em paralelo, de tal forma que em cada vez o nome aponte para um elemento do agregado. Estes conjuntos e listas podem, inclusive, ser o conjunto de nomes de um subconjunto dos agentes.

A marca `<choose>` é composta por um nome, duas expressões e uma regra. A primeira expressão é um agregado (conjunto ou lista), e a semântica desta construção é executar a regra de transição uma única vez, de tal forma que o nome aponte para um elemento aleatório do agregado que atenda à condição imposta pela segunda expressão. Caso nenhum elemento atenda à condição, então a regra não é executada.

A marca `<let>` é composta por um nome, uma expressão e uma regra. A semântica desta marca é executar a regra dada em um ambiente de nomes onde o nome dado denota a expressão também dada.

A marca `<case>` tem a seguinte composição: uma expressão, seguida de pares *expressão* $\times$ *regra*, seguidos de uma regra. A semântica de `<case>` é avaliar a primeira expressão dada e então executar a regra cuja expressão é equivalente à primeira. Caso nenhum casamento ocorra, a última regra é executada. Nesta regra, casamento significa que o valor da expressão avalia inicialmente é o mesmo daquele denotado pela expressão da alternativa.

A marca `<with>` tem a seguinte composição: uma expressão, seguida de triplas *nome* $\times$ *tipo* $\times$ *regra*, seguidas de uma regra. A semântica de `<with>` é avaliar a primeira expressão dada e

```
<module>
  ...
  <dynamicfunctions>
    <dynamicfunction>
      <name>f</name>
      <type><integer/></type>
    </dynamicfunction>
  </dynamicfunctions>
  ...
  <semaphores>
    <semaphore>
      <name>sem</name>
      <literalinteger>1</literalinteger>
    </semaphore>
  </semaphores>
  ...
  <transition>
    <rule>
       ... if step == 0
       <wait>
         <name>sem</name>
       </wait>
       ... faz step = 1
       ...
       ... if step == 1
       <update>
          ... atualiza ou consulta f
       </update>
       ... faz step = 2
       ...
       ... if step == 2
       <signal>
         <name>sem</name>
       </signal>
       ... faz step = 1
       ...
    </rule>
  </transition>
</module>
```

Figura 4.21: Exemplo do uso de um semáforo.

```
<conditional>
  <expression>
    ... aqui vai a expressão que denota a condição
  </expression>
  <rule>
    ... aqui vai a regra que deve ser executada se a condição avaliar para verdadeiro
  </rule>
  <rule>
    ... aqui vai a regra que deve ser executada se a condição avaliar para falso
  </rule>
</conditional>

<forall>
  <name>oNomeDoAlias<name>
  <expression>
    ... aqui vai a expressão que denota o conjunto
  </expression>
  <rule>
    ... aqui vai a regra que deve ser executada para cada elemnto do conjunto
  </rule>
</forall>

<choose>
  <name>oNomeDoAlias<name>
  <expression>
    ... aqui vai a expressão que denota o conjunto
  </expression>
  <expression>
    ... aqui vai a expressão que denota a condição que deve ser atendida
  </expression>
  <rule>
    ... aqui vai a regra que deve ser executada para o elemnto escolhido
  </rule>
</choose>

<let>
  <name>oNomeDoAlias<name>
  <expression>
    ... aqui vai a expressão cujo valor é associado ao alias
  </expression>
  <rule>
    ... aqui vai a regra que deve ser executada
  </rule>
</let>
```

Figura 4.22: Exemplos de regras compostas (1/2).

```
<case>
  <expression>
    ... aqui vai a expressão que denota o valor usado nas comparações
  </expression>
  <expression>
    ... se o valor desta expressão for igual ao da primeira expressão ...
  </expression>
  <rule>
    ... então esta regra é executada
  </rule>
  <expression>
    ... caso contrário, se o valor desta expressão for igual ao da primeira expressão ...
  </expression>
  <rule>
    ... então esta regra é executada
  </rule>
  ...
  <rule>
    ... se nenhuma regra foi executada até aqui, então esta regra padrão é executada
  </rule>
</case>

<with>
  <expression>
    ... aqui vai a expressão que denota o valor usado nas comparações
  </expression>
  <name>nome1</name>
  <type>
    ... se o valor da expressão avaliada inicialmente for deste tipo ...
  </type>
  <rule>
    ... então esta regra é executada, onde nome1 denota o valor da expressão avaliada
  </rule>
  <name>nome2</name>
  <type>
    ... caso contrário, se o valor da expressão avaliada inicialmente for deste tipo ...
  </type>
  <rule>
    ... então esta regra é executada, onde nome2 denota o valor da expressão avaliada
  </rule>
  ...
  <rule>
    ... se nenhuma regra foi executada até aqui, então esta regra padrão é executada
  </rule>
</with>
```

Figura 4.23: Exemplos de regras compostas (2/2).

então executar a regra cujo tipo associado na tripla seja equivalente àquele do valor resultante da avaliação da expressão. Caso nenhum casamento ocorra, a última regra é executada. Nesta regra, casamento significa que o valor da expressão avalida inicialmente possui o mesmo tipo daquele especificado na alternativa. Dentro de uma regra de uma tripla, o valor da expressão avaliada é acessível por meio do nome que compõe a tripla.

A marca `<block>` permite a composição paralela de regras, e é composta por duas regras. Múltiplas regras podem ser compostas por meio da utilização recursiva da marca `<block>`.

Tipos

A marca `<type>` representa um tipo na linguagem intermediária proposta. O conteúdo desta marca pode ser uma dentre quatorze opções, conforme apresentado na Figura 4.24. Estas opções se dividem basicamente em dois conjuntos: tipos básicos e construtores de tipos.

- Os tipos básicos existentes são `<any>`, `<boolean>`, `<character>`, `<integer>`, `<real>`, `<string>` e `<node>`.
- Os construtores de tipo permitem a definição de novos tipos a partir de tipos existentes ou de outras informações, e são `<disjointunion>`, `<interval>`, `<tuple>`, `<list>`, `<set>`, `<functional>` e `<abstraction>`.

Tipos Simples A marca `<any>` é um dos tipos básicos e representa o tipo que é a união disjunta de todos os tipos. A marca `<boolean>` representa o tipo dos valores booleanos. A marca `<character>` representa o tipo dos caracteres. Um caractere possui 8 bits, à semelhança d tipo `char` de C++. A marca `<integer>` representa o tipo dos inteiros. Um inteiro possui 32 bits, à semelhança do tipo `int` de C++. A marca `<real>` representa o tipo dos valores reais. Um número real possui 64 bits, à semelhança do tipo `double` de C++. A marca `<string>` representa o tipo das cadeias de caracteres. A marca `<node>` representa o tipo nó, utilizado para a representação de tipos recursivos dentro do contexto de Machĭna, sendo introduzido para permitir a implementação de *pattern matching* segundo proposto na linguagem Machĭna [BTIB05].

Tipos Compostos A marca `<disjointunion>` representa o tipo que é a união disjunta de dois outros tipos. Seus componentes são os tipos que participam da união. A marca `<interval>` representa um intervalo de valores discretos. Os componentes desta marca são duas expressões que determinam o início e o fim do intervalo. Estas expressões devem ser do tipo inteiro ou do tipo caracter. A marca `<tuple>` é um dos construtores de tipo apresentados na Figura 4.24. Esta marca é constituída de uma seqüência de tipos, que são os tipos dos componentes da tupla, na ordem especificada. É necessário a definição de ao menos um componente da tupla. A marca `<list>` define o tipo das listas e o conteúdo da marca é o tipo dos elementos da lista. A marca `<set>` define o tipo dos conjuntos e seu conteúdo é o tipo dos elementos do conjunto. Finalmente, a marca `<functional>` representa os tipos da funções. Esta marca possui três componentes: o nome do parâmetro da função, o tipo do domínio da função e o tipo da imagem da função. Não são permitidas funções de alta ordem. Para a representação de funções n-árias, utiliza-se uma recursão à direita, ou seja, a segunda expressão da função é outra marca `<functional>`. Finalmente, a marca `<abstraction>` designa o tipo dos valores que

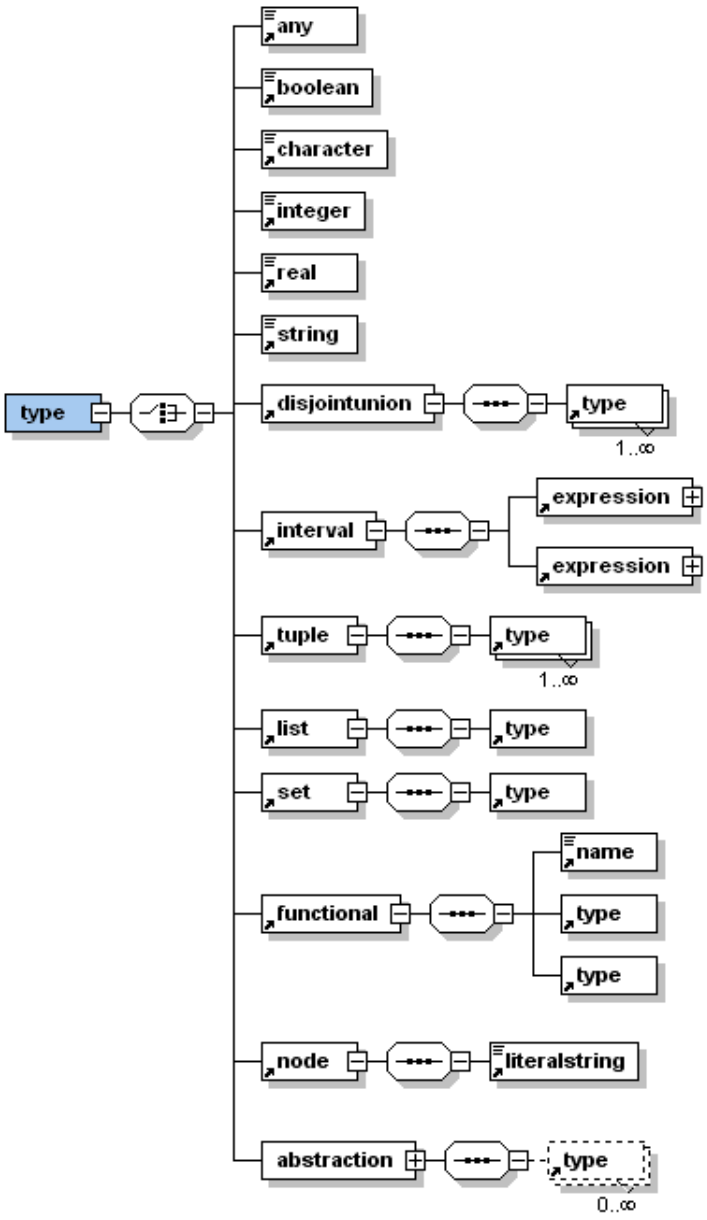

Figura 4.24: Estrutura da marca `<type>`.

representam as ações e sub-máquinas, permitindo que estas sejam passadas como parâmetro umas às outras.

Expressões

A marca `<expression>` representa uma expressão na linguagem intermediária proposta. O conteúdo desta marca pode ser uma dentre cinqüenta e seis opções, conforme apresentado na Figura 4.25. Estas opções se dividem basicamente em alguns conjuntos, a saber:

- literais, constituídos pelas marcas `<literalboolean>`, `<literalcharacter>`, `<literalinteger>`, `<literalreal>` e `<literalstring>`;
- o identificador individual do agente, `<self>`;
- a expressão que denota a ausência de um valor, `<undef>`;
- operadores unários, constituídos pelas marcas `<minusunary>`, `<not>`, `<typecast>`, `<tail>`, `<head>`, `<initial>` e `<last>`;
- operadores binários, constituídos pelas marcas `<mult>`, `<div>`, `<mod>`, `<add>`, `<minus>`, `<intersection>`, `<setconstructor>`, `<listconstructor>`, `<equal>`, `<diff>`, `<lessthan>`, `<morethan>`, `<lessequalthan>`, `<moreequalthan>`, `<and>`, `<or>`, `<xor>`, `<cons>`, `<in>`, `<tupleprojection>`, `<nodelabel>` e `<nodecontent>`;
- chamadas de funções, constituídas pelas marcas `<staticfunctioncall>`, `<derivedfunctioncall>`, `<dynamicfunctioncall>` e `<externalfunctioncall>`;
- *aliases*, constituídos pelas marcas `<parametercall>`, `<letalias>`, `<forallalias>`, `<choosealias>`, `<withalias>`, `<letexpalias>` e `<withexpalias>`;
- construção de agregados, constituídos pelas marcas `<tupleagregate>`, `<setagregate>`, `<listagregate>` e `<nodeagregate>`;
- expressões mais complexas, constituídas pelas marcas `<ifexp>`, `<letexp>`, `<caseexp>` e `<withexp>`;
- referências a abstrações, dada pela marca `<abstractionreference>`.

Todas estas marcas são detalhadas a seguir. Exemplos das marcas de expressão são apresentados nas Figuras 4.27, 4.28, 4.29 e 4.30.

Literais As marcas para literais são apresentadas na Figura 4.26. A marca `<literalboolean>` representa um literal booleano. O seu conteúdo é o texto `true` ou o texto `false`. Nenhum outro texto é permitido, nem é permitido deixar o conteúdo vazio.

A marca `<literalcharacter>` representa um caractere literal. O seu conteúdo é o caractere representado, cercado de aspas simples.

A marca `<literalinteger>` representa um inteiro literal. O seu conteúdo é um inteiro sinalizado de 4 bytes, ou seja, pertencente ao intervalo $-2.147.483.648$ a $2.147.483.647$, à semelhança de um valor do tipo `int` de C++.

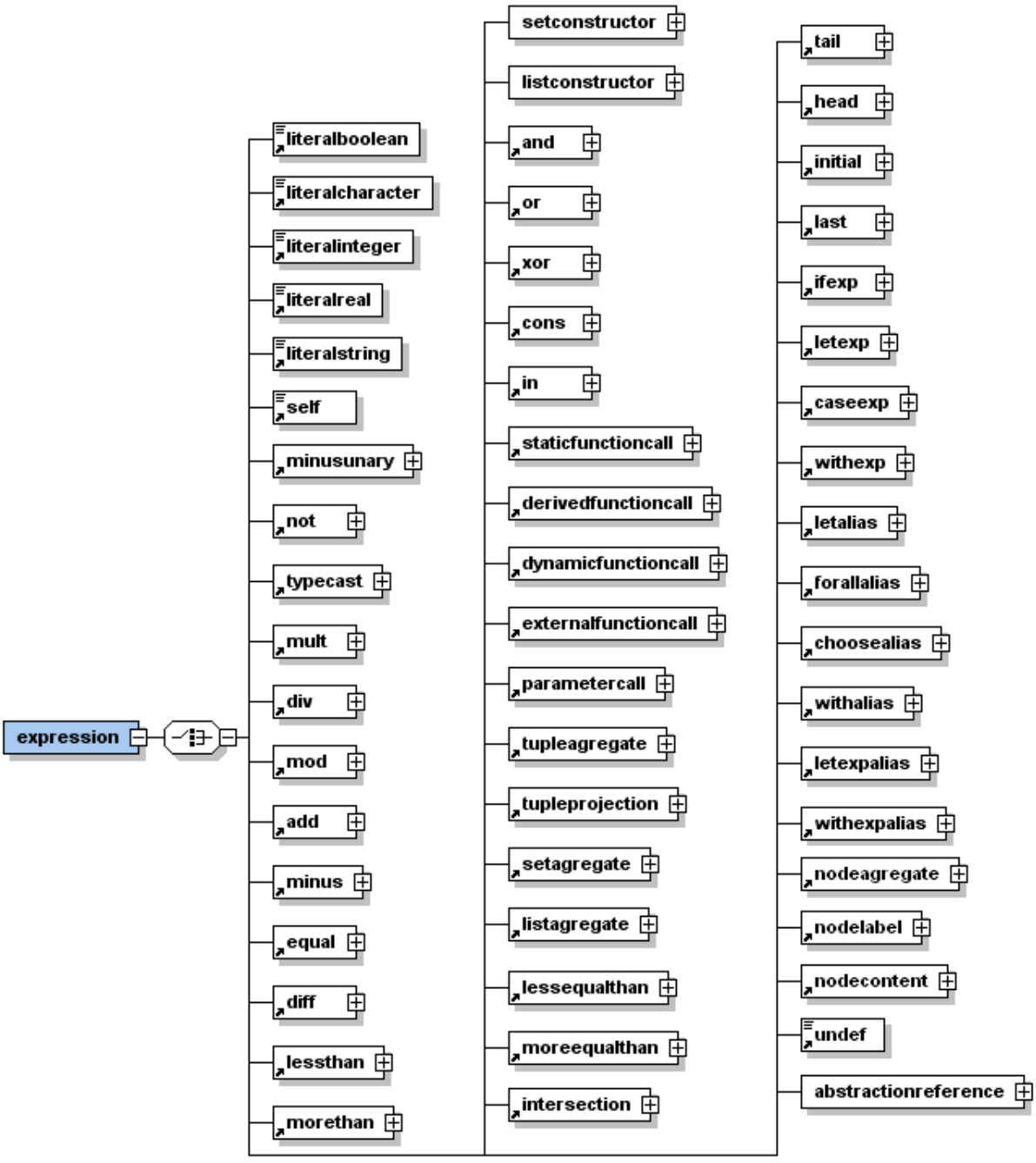

Figura 4.25: Estrutura da marca `<expression>`.

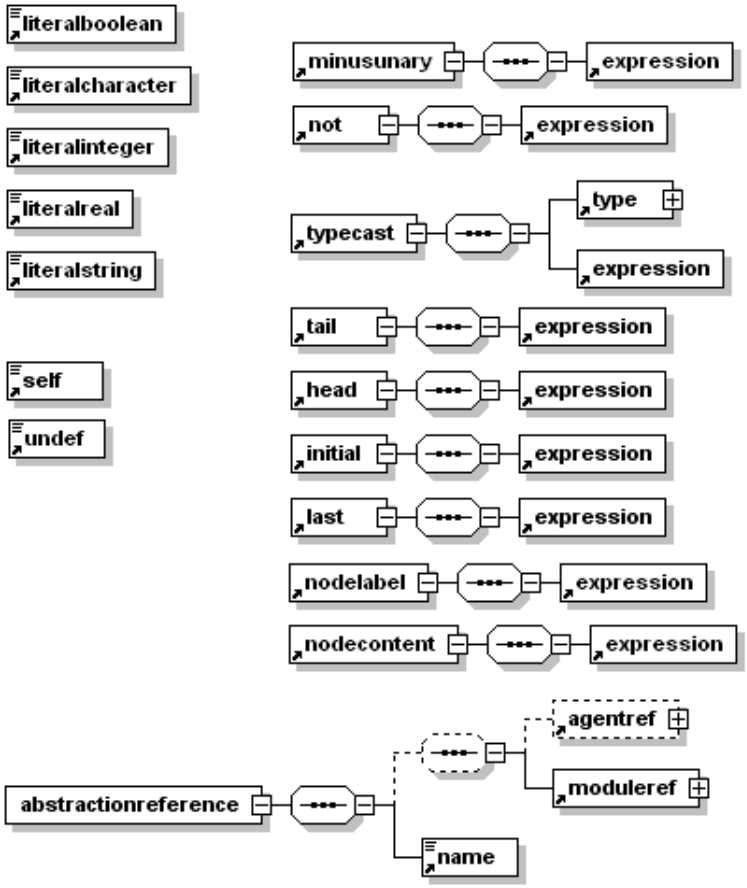

Figura 4.26: Algumas alternativas para a marca `<expression>`: literais, operadores unários, `<self>` e `<undef>`.

```
<literalboolean>true</literalboolean>

<literalinteger>1</literalinteger>

<self/>

<not>
  <expression>
    <literalboolean>true</literalboolean>
  </expression>
</not>

<mult>
  <expression>
    <literalreal>3.14159</literalreal>
  </expression>
  <expression>
    <literalreal>2.71</literalreal>
  </expression>
</mult>

<staticfunctioncall>
  <name>sin</name>
  <expression>
    <literalreal>1.57</literalreal>
  </expression>
</staticfunctioncall>

<tupleagregate>
  <expression>
    <literalreal>1.57</literalreal>
  </expression>
  <expression>
    <literalinteger>1</literalinteger>
  </expression>
  <expression>
    <literalboolean>true</literalboolean>
  </expression>
</tupleagregate>


<tupleprojection>
  <expression>
    ... a expressão que denota a tupla
  </expression>
  <literalinteger>0</literalinteger>
</tupleprojection>
```

Figura 4.27: Exemplos de expressões (1/4).

```
<nodeagregate>
  <expression>
    ... a expressão que denota o nome do nó
  </expression>
  <expression>
    ... a expressão que denota o valor do nó
  </expression>
</nodeagregate>

<nodelabel>
  <expression>
    ... a expressão que denota o nó cujo nome será obtido
  </expression>
</nodelabel>

<nodecontent>
  <expression>
    ... a expressão que denota o nó cujo valor será obtido
  </expression>
</nodecontent>

<ifexp>
  <expression>
    ... aqui vai a expressão que denota a condição
  </expression>
  <expression>
    ... aqui vai a expressão que deve ser avaliada caso a condição denote verdadeiro
  </expression>
  <expression>
    ... aqui vai a expressão que deve ser avaliada caso a condição denote falso
  </expression>
</ifexp>

<letexp>
  <name>oNomeDoAlias<name>
  <expression>
    ... aqui vai a expressão cujo valor é associado ao alias
  </expression>
  <expression>
    ... aqui vai a expressão onde o alias é utilizado
  </expression>
</letexp>
```

Figura 4.28: Exemplos de expressões (2/4).

```
<case>
  <expression>
    ... aqui vai a expressão que denota o valor usado nas comparações
  </expression>
  <expression>
    ... se o valor desta expressão for igual ao da primeira expressão ...
  </expression>
  <expression>
    ... então esta expressão é avaliada
  </expression>
  <expression>
    ... caso contrário, se o valor desta expressão for igual ao da primeira expressão ...
  </expression>
  <expression>
    ... então esta expressão é avaliada
  </expression>
  ...
  <expression>
    ... se nenhuma expressão foi escolhida e avaliada até aqui, então esta expressão padrão
        é escolhida e avaliada
  </expression>
</case>
```

Figura 4.29: Exemplos de expressões (3/4).

```
<with>
  <expression>
    ... aqui vai a expressão que denota o valor usado nas comparações
  </expression>
  <name>nome1</name>
  <type>
    ... se o valor da expressão avaliada inicialmente for deste tipo ...
  </type>
  <expression>
    ... então esta expressão é avaliada, onde nome1 denota o valor da primeira expressão
  </expression>
  <name>nome2</name>
  <type>
    ... caso contrário, se o valor da expressão avaliada inicialmente for deste tipo ...
  </type>
  <expression>
    ... então esta expressão é avaliada, onde nome2 denota o valor da primeira expressão
  </expression>
  ...
  <expression>
    ... se nenhuma expressão foi escolhida e avaliada até aqui, então esta expressão padrão
        é escolhida e avaliada
  </expression>
</with>
```

Figura 4.30: Exemplos de expressões (4/4).

A marca `<literalreal>` representa um número real literal. O seu conteúdo é um número de ponto flutuante de 8 bytes, ou seja, pertencente ao intervalo $\pm 2.2E-308$ a $\pm 1.8E308$, à semelhança de um valor do tipo `double` de C++.

A marca `<literalstring>` representa uma cadeia de caracteres. O seu conteúdo é a cadeia de caracteres representada, cercada por aspas duplas. Caracteres de escape, tais como `\n` e `\t`, são reconhecidos.

Self O operador especial `<self>` retorna o nome do agente onde a instância desta marca é encontrada, e possui tipo string.

Undef O operador especial `<undef>` representa a expressão que denota a ausência de valor.

Operadores Unários Os operadores unários são apresentados na Figura 4.26. O operador unário de inversão do sinal é representado pela marca `<minusunary>`. A expressão que constitui seu conteúdo deve denotar um valor do tipo inteiro ou do tipo real.

O operador unário de negação lógica é representado pela marca `<not>`. A expressão que constitui seu conteúdo deve denotar um valor do tipo booleano.

A marca `<typecast>` é utilizada para efetuar a conversão de tipos. Dada uma expressão, seu tipo é convertido para o tipo dado, sempre que possível. É útil para obter um tipo mais específico a partir de um tipo mais geral. Esta situação aparece particularmente quando se trabalha com união disjunta ou com o tipo `<any>`.

As marcas `<tail>`, `<head>`, `<initial>` e `<last>` costituem as operações tradicionais sobre listas. Elas recebem como argumento uma expressão que representa uma lista e retornam, respectivamente, a lista sem o primeiro elemento, o primeiro elemento da lista, a lista sem o último elemento e o último elemento da lista. Um erro é lançado se a lista passada como parâmetro está vazia.

O operador unário `<nodelabel>` recebe como parâmetro um valor do tipo nó, e retorna uma string que representa o seu nome. Por sua vez, o `<nodecontent>` recebe como parâmetro um valor do tipo nó, e retorna o valor do seu conteúdo.

Operadores Binários Os operadores binários são apresentados na Figura 4.31. As marcas `<mult>`, `<div>`, `<add>` e `<minus>` denotam, respectivamente, os operadores de multiplicação, divisão, adição e subtração, e as expressões que constituem seu conteúdo devem denotar operandos do tipo inteiro ou do tipo real. Adicionalmente, as marcas `<add>` e `<minus>` aceitam expressões de tipo `Set`, caso em que denotam a união e a diferença entre conjuntos, respectivamente. A marca `<intersection>` realiza a interseção entre dois conjuntos.

A marca `<mod>` denota o operador binário de resto da divisão. As expressões que constituem seus operandos devem ser do tipo inteiro.

As marcas `<equal>`, `<diff>`, `<lessthan>`, `<morethan>`, `<lessequalthan>` e `<moreequalthan>` denotam, respectivamente, os operandos de comparação igual a, diferente de, menor que, maior que, igual ou menor que e igual ou maior que. Enquanto `<equal>` e `<diff>` comparam os valores de quaisquer expressões válidas da linguagem, bastando apenas que sejam do mesmo tipo, os demais comparadores se aplicam a expressões que denotam inteiros, reais ou caracteres. A exceção é a marca `<lessthan>`, que também pode ser utilizada para comparar se um conjunto está contido em outro conjunto, mas estes não são iguais entre si.

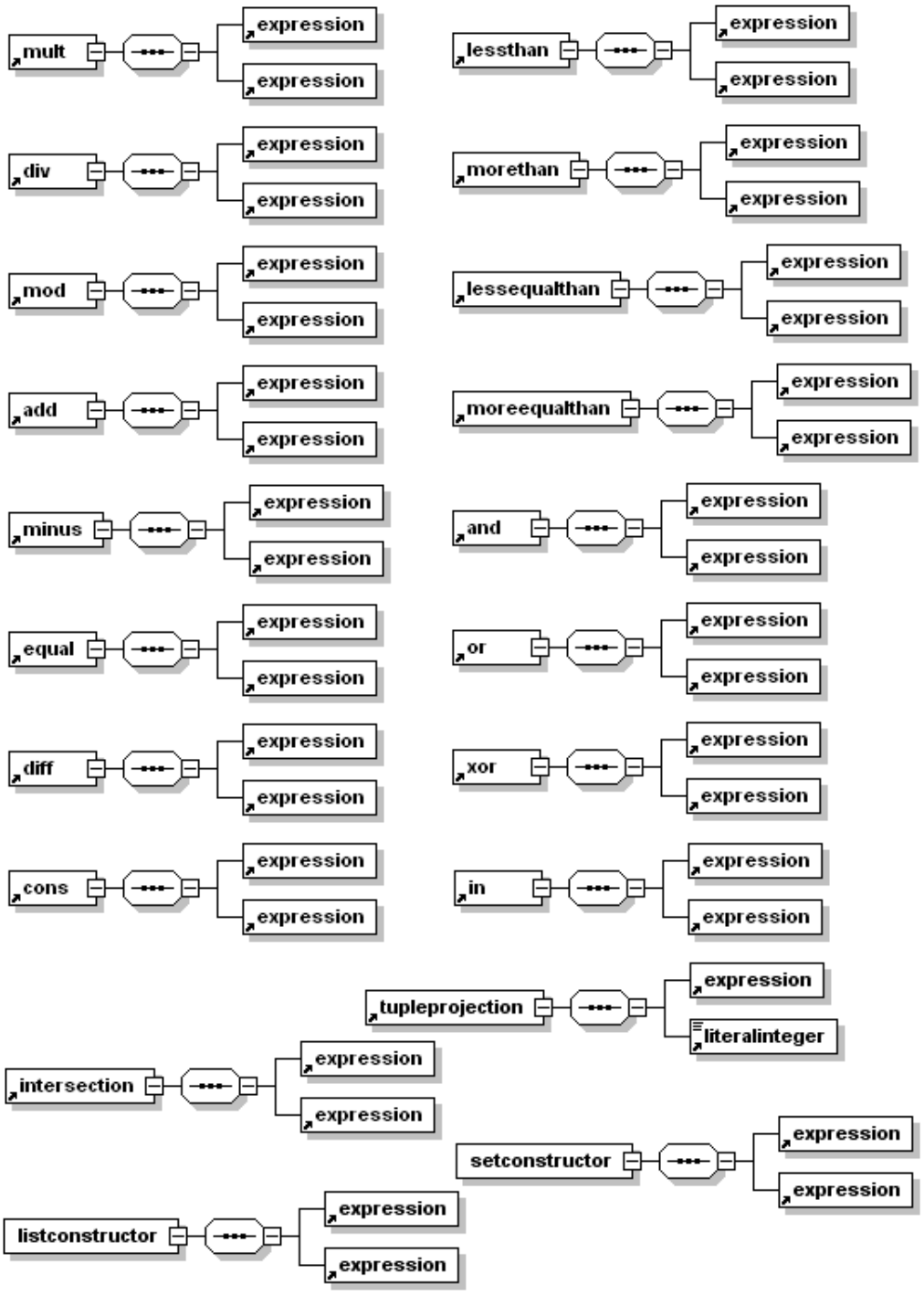

Figura 4.31: Algumas alternativas para a marca `<expression>`: operadores binários

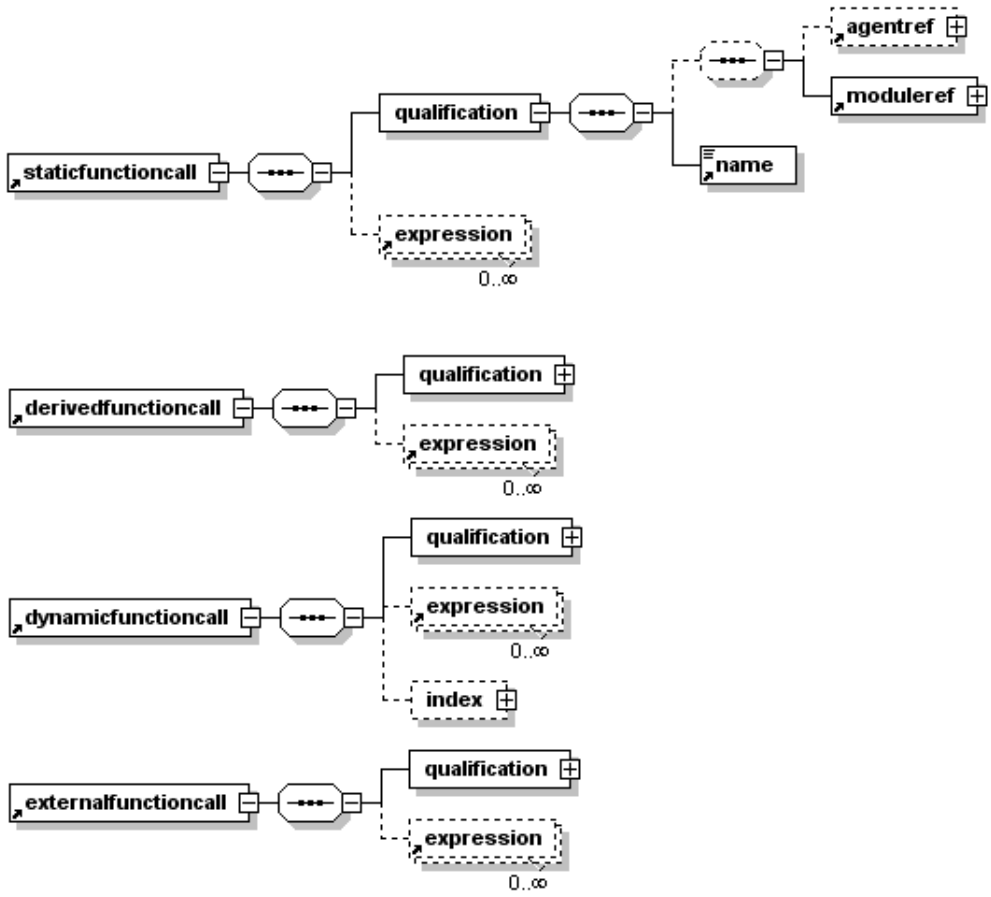

Figura 4.32: Algumas alternativas para a marca `<expression>`: chamadas de funções.

As marcas `<and>`, `<or>` e `<xor>` representam, respectivamente, os operadores E-lógico, OU-lógico e OU-EXCLUSIVO-lógico. Seus operandos devem ser elementos do tipo booleano.

A marca `<cons>` denota o operador binário de construção de listas. O tipo do valor denotado pela segunda expressão é uma lista cujos elementos são de tipo equivalente ao tipo do valor denotado pela primeira expressão. A lista resultante é uma lista cuja cabeça é o elemento denotado pela primeira expressão, e cuja cauda é o elemento denotado pela segunda expressão. Esta marca também pode ser utilizada para adicionar um elemento a um conjunto existente.

A marca `<in>` denota o operador binário de pertinência. Dado uma expressão, verifica se o valor denotado por esta está contido em uma lista ou em um conjunto denotado pela segunda expressão.

A marca `<tupleprojection>` faz a projeção da tupla. Dada uma expressão que denota uma tupla e um literal inteiro i, a projeção retorna o i-ésimo elemento da tupla.

As marcas `<setconstructor>` e `<listconstructor>` recebem duas expressões que denotam um par de inteiros ou um par de caracteres e retornam, respectivamente, um conjunto ou uma lista contendo os valores naquele intervalo fechado.

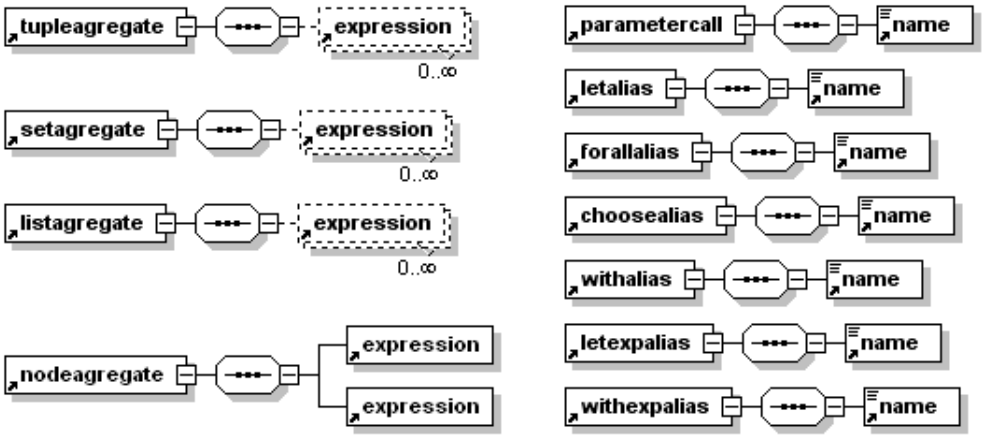

Figura 4.33: Algumas alternativas para a marca `<expression>`: agregados e *aliases*.

Chamadas de Funções A marca `<staticfunctioncall>` representa uma chamada a uma função estática. Seu conteúdo consiste no nome da função e em uma seqüência de expressões que denotam os valores a serem passados como parâmetros à função, como apresentado na Figura 4.32. Opcionalmente, pode-se especificar o nome de um agente e um módulo. Se são especificados tanto o nome do agente quanto do módulo, é invocada a função especificada de um agente do nome e tipo dados. Caso apenas o módulo seja especificado, esta função é procurada no espaço global deste módulo e então é invocada. Caso contrário, a função a ser chamada está no próprio espaço do agente chamador. Esta maneira de se especificar o escopo onde a função é encontrada é o mesmo daquele apresentado anteriormente para ações, submáquinas e semáforos.

A passagem de parâmetros para chamadas de funções é feita por cópia, pois parêmetros de função são sempre de entrada.

A marca `<derivedfunctioncall>` representa uma chamada a uma função derivada. Seu conteúdo consiste no nome da função e em uma seqüência de expressões que denotam os valores a serem passados como parâmetros à função. À semelhança da chamada de funções estáticas, o contexto onde a função derivada é encontrada pode ser especificado por meio da definição do nome do agente e do módulo.

A marca `<dynamicfunctioncall>` representa uma chamada a uma função dinâmica. Seu conteúdo consiste no nome da função e em uma seqüência de expressões que denotam os valores a serem passados como parâmetros à função. À semelhança da chamada de funções estáticas, o contexto onde a função dinâmica é encontrada pode ser especificado por meio da definição do nome do agente e do módulo. Adicionalmente, uma chamada a uma função dinâmica pode conter um índice inteiro, caso se deseje acessar apenas um componente de uma tupla.

A marca `<externalfunctioncall>` representa uma chamada a uma função externa. Seu conteúdo consiste no nome da função e em uma seqüência de expressões que denotam os valores a serem passados como parâmetros à função. À semelhança da chamada de funções estáticas, o contexto onde a função externa é encontrada pode ser especificado por meio da definição do nome do agente e do módulo.

Agregados Neste trabalho, compreende-se por agregados as listas, os conjuntos e as tuplas. A marca `<setagregate>` representa um conjunto de valores, e é constituída da seqüência de expressões que denotam o conjunto, como pode ser visto na Figura 4.33. A marca `<listagregate>` representa uma lista de valores, e é constituída da seqüência de expressões que denotam a lista. De forma semelhante, uma tupla é a seqüência das expressões que denotam seus componentes, envolvidas pela marca `<tuplegregate>`. Um nó é construído por meio da marca `nodeagregate`, que recebe como parâmetros duas expressões: a primeira denota o nome do nó, enquanto a segunda denota o seu valor.

Aliases Na linguagem proposta, existem construções que associam um nome a um valor ou expressão. Chama-se de *alias* o uso de uma expressão por meio de seu nome associado.

Uma das associações mais tradicionais é aquela que amarra o nome de um parâmetro formal à expressão que denota o valor passado como parâmetro de fato. Dentro do corpo de uma função ou abstração de regra, um parâmetro é acessível por meio da marca `<parametercall>`.

Uma expressão vinculada a um nome em uma regra `<let>` ou em uma expressão `<letexp>` é utilizada por meio das marcas `<letlias>` e `<letexplias>`, respectivamente.

Para as regras `<forall>` e `<choose>`, o elemento do conjunto associado ao nome é obtido respectivamente por meio das marcas `<forallalias>` e `<choosealias>`. E em uma regra `<with>` ou expressão `<withexp>`, o valor casado no tipo oferecido por cada alternativa é acessado por meio das marcas `<withalias>` e `<withexpalias>`, respectivamente.

Todas estas marcas recebem como parâmetro o nome do *alias* a ser acessado.

Outras Expressões Outras expressões são apresentadas na Figura 4.34. Uma expressão condicional é representada pela marca `<ifexp>`. Seu conteúdo consiste em três expressões, sendo que a primeira denota um valor de tipo booleano, e as outras duas denotam valores de mesmo tipo. Caso a avaliação da primeira expressão resulte no valor lógico *true*, o valor da expressão condicional é o valor denotado pela segunda expressão; caso contrário, o valor da expressão condicional é o valor denotado pela terceira expressão.

A marca `<letexp>` é composta por uma seqüência de pares *nome* × *expressão*, seguidos de uma expressão. O valor denotado por esta marca é o valor denotado pela última expressão que a constitui, sendo esta avaliada em um ambiente de nomes onde os nomes que aparecem nos pares *nome* × *expressão* denotam a expressão associada.

A marca `<caseexp>` tem a seguinte composição: uma expressão, seguida de pares *expressão* × *expressão*, seguidos de uma expressão. A semântica de `<caseexp>` é avaliar a primeira expressão dada, obtendo um valor. Em seguida, são verificados os pares *expressão* × *expressão* para verificar se algum destes pares possui em seu primeiro componente uma expressão que denota o mesmo valor. Em caso afirmativo, o valor da expressão `caseexp` é o valor denotado pela expressão do segundo componente. Caso não aconteça o casamento, o valor da expressão `caseexp` é o valor da expressão definida no final. Nesta regra, casamento significa que o valor da expressão avalida inicialmente é o mesmo daquele denotado pela expressão da alternativa. Pode-se explicar a semântica desta construção ainda de outra forma. Seja

$$caseexp = \{e, (e_1, e_{r1}), (e_2, e_{r2}), \ldots, (e_k, e_{rk}), e_{default}\}$$

A semântica de *caseexp*, $[caseexp]$, é dada por

$$[caseexp] = \begin{cases} e_i & \text{se } \exists i | (e_i = e) \wedge (e_j \neq e | \forall j < i) \\ e_{default} & \text{caso contrário} \end{cases}$$

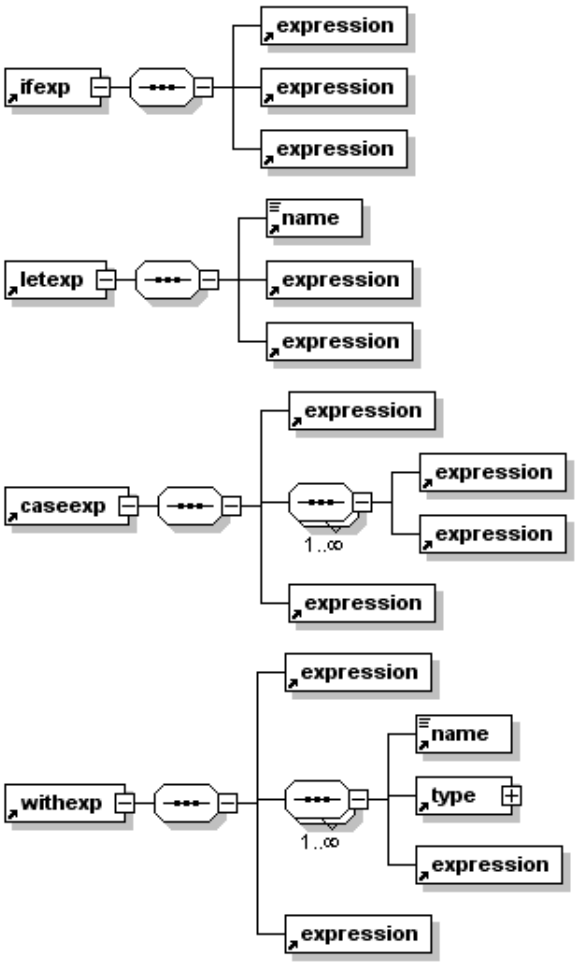

Figura 4.34: Algumas alternativas para a marca `<expression>`: expressões complexas.

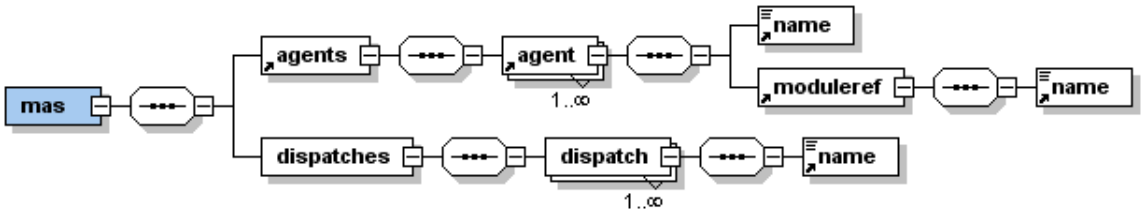

Figura 4.35: Estrutura da marca `<mas>`.

A marca `<withexp>` tem a seguinte composição: uma expressão, seguida de triplas *tipo* × *nome* × *expressão*, seguidos de uma expressão. A semântica de `<withexp>` é avaliar a primeira expressão dada, obtendo um valor. Em seguida, são verificadas as triplas *tipo* × *nome* × *expressão* para verificar se alguma destas triplas possui em seu primeiro componente um tipo que é equivalente ao tipo do valor obtido. Em caso afirmativo, o valor da expressão `withexp` é o valor denotado pela expressão do terceiro componente. Na avaliação desta expressão, pode-se acessar o valor comparado por meio do nome fornecido no segundo componente da tripla. Caso não aconteça o casamento, o valor da expressão `withexp` é o valor da expressão definida no final. Nesta regra, casamento significa que o valor da expressão avalida inicialmente possui o mesmo tipo daquele especificado na alternativa. Pode-se explicar a semântica desta construção ainda de outra forma. Seja

$$withexp = \{e, (n_1, t_1, e_{r1}), (n_2, t_2, e_{r2}), \ldots, (e_k, t_k, e_{rk}), e_{default}\}$$

A semântica de $withexp$, $[withexp]$, é dada por

$$[withexp] = \begin{cases} e_i\Big|_{n_i=e} & \text{se } \exists i | (t_i = t_e) \wedge (t_j \neq t_e | \forall j < i) \\ e_{default} & \text{caso contrário} \end{cases}$$

onde t_e é o tipo de e.

MIR Agents Starter

A marca `<mas>` é a raiz de um arquivo de especificação de *MIR Agents Starter*. Basicamente, seu conteúdo é uma seqüência de declarações de agentes, como mostrado na Figura 4.35.

Agentes

Os agentes disparados no início da execução da especificação são declarados dentro da marca `<agents>`, apresentada na Figura 4.35. Pelo menos um agente deve ser declarado em uma especificação válida.

A função de um agente é executar a regra de transição de um módulo dentro de seu contexto. Por isso, a estrutura da marca `<agent>` consiste de uma referência a um módulo, que nada mais é do que o nome do módulo em si, como pode ser visto na Figura 4.35. Diz-se que o tipo do agente é o seu módulo. Agentes de um mesmo tipo compartilham o escopo global do módulo, ao mesmo tempo que possuem cópias individuais do escopo local do módulo. Depois de criados, os agentes são explicitamente disparados por meio da regra `<dispatch>`.

A marca `<moduleref>` representa uma referência a um módulo, e é constituída de um nome, que é o nome do módulo referenciado, como apresentado na Figura 4.35.

4.5 Exemplos

O Apêndice B apresenta um conjunto de exemplos de programas escritos na linguagem MIR. Cada exemplo é explicado módulo a módulo e foi utilizado na validação do trabalho como um todo. Além disso, estes exemplos foram projetados de modo a cobrir todas as construções da linguagem MIR, conforme apresentado nas Tabelas 7.2, 7.3 e 7.4.

4.6 Conclusões sobre a Linguagem MIR

A linguagem MIR, proposta por Tirelo, Oliveira *et alii* [Tir00, Oli04], e aqui modificada, provê a expressividade necessária à especificação de programas segundo o paradigma de máquinas de estado abstratas. MIR se mostra adequada, portanto, à utilização como linguagem a ser entendida pelo arcabouço kℓar para a otimização de especificações ASM, uma vez que foi especificamente projetada para permitir otimizações específicas deste modelo. Além disso, recursos da linguagem tais como a presença de módulos, de escopos locais e globais e a composição de módulos garantem a escalabilidade da linguagem, permitindo que problemas maiores possam ser decompostos em partes, abordados e então recombinados.

A nossa contribuição à linguagem MIR consiste no fato desta prover a capacidade de se especificar sistemas multi-agentes. Este agentes possuem as propriedades de autonomia, sensibilidade ao contexto e proatividade, conforme apresentadas por Zambonelli *et al* [ZJW03].

Do ponto de vista da sintaxe, utilizou-se uma representação textual baseada em XML para a representação de especificações de programas em linguagem MIR. Esta representação baseada em XML é adequada para uma representação intermediária, conforme os motivos apresentados oportunamente na Seção 4.4. Outras vantagens que merecem ser destacadas são a possibilidade de se verificar a sintaxe de um programa por meio do *XML Schema Definition (XSD)* fornecido junto com o arcabouço[3] e a existência de várias ferramentas para a edição de arquivos XML que poderiam ser utilizados na programação nesta linguagem.

[3]Este arquivo é apresentado também no Apêndice A.

Capítulo 5

Implementação da Linguagem MIR

Uma das vantagens do uso de ASM para a especificação da semântica de um algoritmo é que esta especificação pode ser executada. Neste trabalho, usa-se MIR para definir as especificações semânticas, e para que as mesmas sejam executadas, faz-se uso de um *"visitor"* de geração de código, o qual gera código C++ que reflete a semântica de módulos e agregadores. Este capítulo tem por objetivo detalhar como as construções em linguagem MIR, após carregadas, são representadas em memória, e como estas são compiladas para o código C++ correspondente.

5.1 Mapeamento de ASM para OOP

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 de elementos ASM em entidades da programação orientada por objetos (OOP).

Os primeiros elementos a se considerar da linguagem MIR são os módulos e os agentes. Para a representação em memória de módulos de programas MIR, foram implementadas as classes da Figura 5.1. Lá estão apresentadas também as classes que representam os elementos do ambiente de um módulo, que são as diversas tabelas de funções, abstrações de regras e semáforos. É percorrendo esta hierarquia que o visitor de geração de código é capaz de gerar o código C++ equivalente. As assinaturas importadas e referenciadas são apresentadas na Figura 5.2.

Um ambiente, por sua vez, é representado em memória por uma classe chamada Environment, cujos componentes são as tabelas das entidades que compõem um ambiente. Estas tabelas são representadas pelas classes StaticAndDerivedFunctionTable, DynamicFunctionTable, DerivedFunctionTable, ExternalFunctionTable, ActionTable, SubMachineTable e SemaphoreTable, que nada mais são do que tabelas das entidades correspondentes a cada uma das classes. Cabe ressaltar que as funções foram organizadas em uma hierarquia tal que funcionalidades comuns são sempre agrupadas em classes abstratas ancestrais a serem derivadas pelas classes que necessitam de tal funcionalidade, evitando, assim, duplicação desnecessária de código.

Todas as classes em questão possuem também o método accept(Visitor* v). Este método é necessário à utilização do padrão de projeto *Visitor*, apresentado na Seção 3.3.7. É por meio deste padrão que as funcionalidades de persistência e geração de código, apresentadas neste capítulo, são implementadas. A presença deste padrão também abre a possibilidade de que novas funcionalidades sejam adicionadas de maneira modular.

As listas de referências e importações são apresentadas na Figura 5.2. Basicamente, estas

listas são constituídas de assinaturas de funções, abstrações de regras e semáforos, agrupadas pelo módulo de origem. A funcionalidade comum das referências e das importações são agrupadas em uma classe abstrata ancestral comum.

Uma especificação MAS (*MIR Agents Starter*) é representada por um objeto da classe MASSpecification, apresentado na Figura 5.3. Esta classe contém, basicamente, um conjunto de agentes a serem instanciados.

No processo de tradução para C++, a compilação de um módulo dá origem a uma classe. Os componentes do módulo, ou seja, as tabelas com as funções, abstrações de regras e semáforos são traduzidos nos membros desta classe, conforme detalhado adiante. A regra de transição do módulo é um método de nome especial, executeTransitionRule. A Figura 5.5 mostra um exemplo de interface da classe resultante da compilação de um módulo. Toda classe que representa um módulo deve herdar de Module, apresentado na Figura 5.7. Esta classe define o método executeTransitionRule como um método puramente virtual. Desta forma, 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 *a) que cria um agente e então o dispara. Esta função é necessária porque agentes são executados concorrentemente, por meio de *threads*, e uma *thread* na biblioteca de concorrência utilizada é uma função com tal assinatura e tipo de retorno.[1] A implementação desta função pode ser vista na Figura 5.4.

Um agente, por sua vez, é um objeto do módulo do seu tipo. O disparo de um agente é a execução de seu método executeTransitionRule por meio de uma *thread*. A compilação de um disparador de agentes, anteriormente chamado de MAS, dá origem a uma classe que cria as instâncias dos agentes especificados de acordo com seu tipo e então dispara estes agentes. As Figuras 5.8 e 5.9 mostram um exemplo da tradução de um disparador de agentes. No construtor da classe, MAS::MAS() na Figura 5.9, são criados os agentes como objetos dos módulos correspondentes, e então estes são inseridos em uma tabela *hash* que associa cada agente criado a seu nome. Este ambiente onde os agentes habitam é acessível por qualquer agente que nela esteja, o que torna os agentes acessíveis entre si por meio de seu nome. Por sua vez, os agentes, após criados, são disparados no método run desta mesma classe. Para cada agente criado, é associada uma *thread* que recebe o agente como parâmetro, assim como uma referência à função runModule. Esta função está definida junto com a classe de base Module, como pode ser visto na Figura 5.4, e sua função basicamente é executar, dentro da *thread*, o método executeTransitionRule do módulo.

A compilação de um arquivo que representa um disparador de agentes (de extensão `mas`) resulta em um par de arquivos de extensão `h` e `cc` de mesmo nome, conforme os modelos apresentados nas Figuras 5.8 e 5.9, respectivamente. Após a inclusão dos cabeçalhos dos módulos utilizados, o arquivo de extensão `h` define a classe MAS, cujo papel é instanciar e disparar os agentes especificados no disparador de agentes, e então servir de contexto para estes agentes. A classe segue o padrão *Singleton*, definido na Seção 3.3.4, 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(). No momento em que um objeto desta classe é instanciado, os agentes são criados. O disparo destes agentes ocorre por meio da chamada ao método run(). 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

[1] A biblioteca utilizada é a `pthread`, que é a biblioteca de *threads* padrão de sistemas operacionais *unix-like*. Maiores detalhes sobre esta biblioteca podem ser encontrados em [But97].

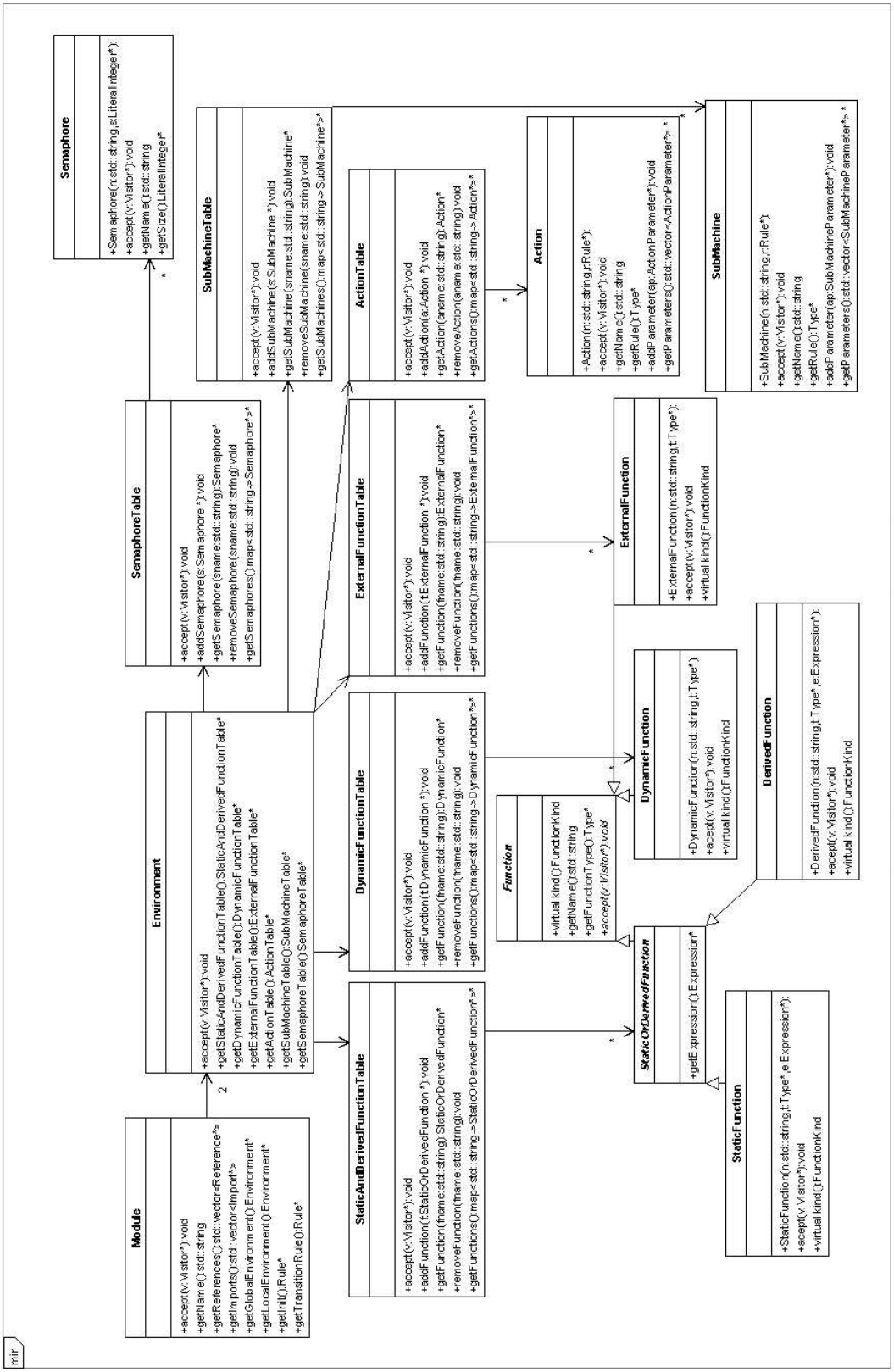

Figura 5.1: Diagrama de classes: a estrutura de um módulo.

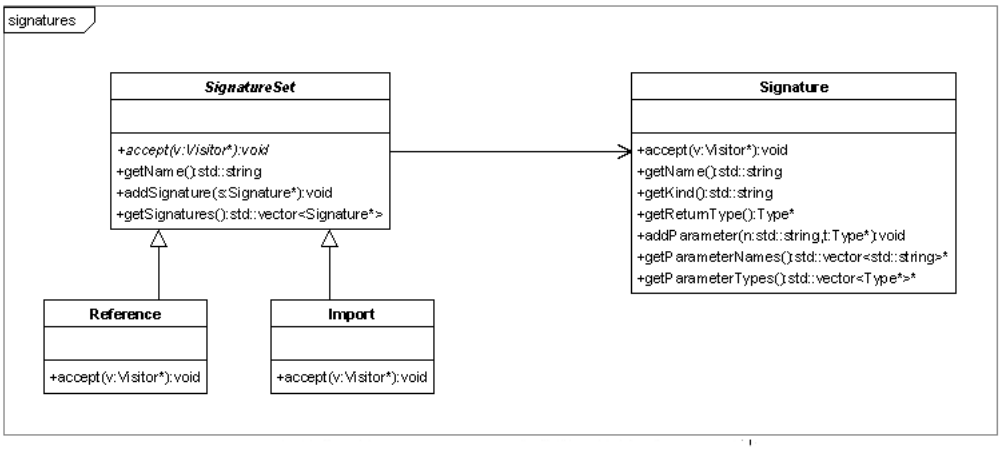

Figura 5.2: Diagrama de classes: referências e importações.

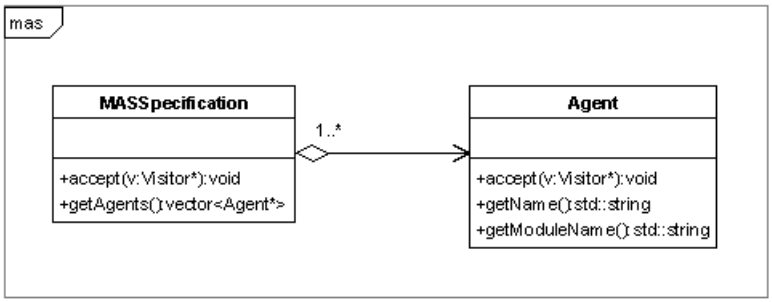

Figura 5.3: Diagrama de classes: especificação MAS.

```
#include "Module.h"
// The function that starts a specific agent, used by the thread
void *runModule(void* a) {
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
  Module* agent = (Module*) a;
  agent->executeTransitionRule();
  return 0; }
```

Figura 5.4: A função runModule, que dispara um agente de um módulo, definida em Module.cc.

```
#ifndef MODULE1_H
#define MODULE1_H
// Some needed includes
#include "Module2.h"
#include "Module3.h"
...
#include "ModuleN.h"
// The base class of all modules
#include "Module.h"
class Module1 : public Module {
  public:
    // The module constructor. It receives a pointer to the context
    // where the other agents can be found.
    Module1(std::map<std::string, Module*>* agents, std::string agName);
    // The imports
    Module2* __Modulo2;
    Module3* __Modulo3;
    ...
    ModuleN* __ModuloN;
    // The global environment
    ...
    // The local environment
    ...
    // The transition rule
    virtual void executeTransitionRule();
  private:
    // The update list and some auxiliary methods
    ... };
#else
class Module1 : public Module;
#endif
```

Figura 5.5: Exemplo de uma classe criada para representar um módulo genérico.

dispara estes agentes. Note que na criação dos agentes no construtor privado MAS::MAS(), cada agente recebe como parâmetro, no momento de sua criação, o contexto no qual ele é inserido, de modo que é possível a um agente acessar as funções dinâmicas de outro agente, o que caracteriza a memória compartilhada. O método run() dispara a execução da regra de transição dos agentes criados como *threads* independentes.

Um módulo é compilado em código C++ por meio da classe CodeGenerator. Esta classe implementa os métodos da classe abstrata Visitor, de modo que a geração de código em si se dá por meio da passagem de um objeto do tipo CodeGenerator para o método visit(Visitor* v) do objeto que representa o módulo a ser compilado. Para cada componente deste módulo, o código correspondente é gerado, e esta seção tem por objetivo apresentar os critérios utilizados para a geração de código em cada elemento do módulo.

A compilação de um módulo dá origem a um par de arquivos de extensões `h` e `cc`, cujas estruturas são semelhantes àquelas apresentadas nas Figuras 5.5 e 5.6, 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.

```
#include "Module1.h"
//--------------------------------------------------------------------------------
Module1::Module1(std::map<std::string, Module*>* a) {
  __Modulo2 = new Module2(a);
  __Modulo3 = new Module3(a);
  ...
  __ModuloN = new ModuleN(a);
  // regra init
  // ... }
//--------------------------------------------------------------------------------
void Module1::executeTransitionRule() {
  bool __stop = false;
  while (!__stop) { // The transition rule... }
}
//--------------------------------------------------------------------------------
```

Figura 5.6: Exemplo de uma classe criada para representar um módulo.

Dentro destas expressões, os primeiros elementos da definição são os `include`'s necessários à compilação deste módulo. Estes `include`'s 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 comum 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 o contexto no qual o agente ora criado será inserido e o nome do agente a ser criado. Este contexto serve também ao propósito de permitir que outros agentes estejam acessíveis.

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

O ambiente global do módulo é então declarado. Abstrações de regras são declaradas como métodos que retornam void, enquanto funções são declaradas como métodos de retorno diferente de void. Semáforos são declarados como estruturas `sem_t` da biblioteca `semaphore` utilizada para prover os semáforos e as operações sobre estes. Os detalhes de como cada um destes métodos é implementado de fato para cada tipo de elemento do ambiente é apresentado mais à frente, quando da explicação do arquivo de extensão `cc` associado a um módulo. O ambiente local do módulo é definido a seguir, à semelhança do ambiente global. A regra de transição do módulo é compilada para um método de nome especial, a saber, void executeTransitionRule().

Após a geração de código de cada módulo utilizado e a geração de código do disparador de agentes, deve-se compilar cada módulo utilizado e então compilar o código do disparador dos agentes. A compilação é direta, e um `makefile` de exemplo é apresentado na Figura 5.10.

A Figura 5.11 apresenta a forma geral de um construtor de um módulo. Em um primeiro momento são inicializados os módulos que porventura constituem este módulo em si. Neste

```
#ifndef MODULE_H
#define MODULE_H
#include <iostream>
//------------------------------------------------------------------------------
class Module {
  public:
    // The transition rule, to be executed
    virtual void executeTransitionRule() = 0;
    // the thread
    pthread_t thread;
    // returns the name of the agent
    std::string getAgentName() { return agentName; };
  protected:
    // the name of the agent
    std::string agentName; };
//------------------------------------------------------------------------------
// The function that starts a specific agent, used by the thread
void *runModule(void* a);
//------------------------------------------------------------------------------
#else
class Module;
void *runModule(void* a);
#endif
```

Figura 5.7: Classe de base de um módulo.

exemplo, o módulo `MyModule` é composto por dois outros módulos, a saber, `A` e `B`. Em seguida, funções dinâmicas locais de aridade 0 são inicializadas. Esta inicialização é necessária apenas para funções 0-árias, pois para funções de aridade superior é utilizada a classe `std::map` junto com um método acessor que garante o valor padrão para pontos ainda não definidos. A mesma inicialização existe para funções dinâmicas 0-árias globais, porém esta inicialização não é feita no construtor, pois trata-se de componentes estáticos da classe do módulo. Finalmente, o código correspondente à regra de inicialização `init` é gerado. Após este código, a função `flush()` é chamada, que levará a termo as atualizações eventualmente geradas por regras de atualização.

5.2 Representação em Memória e Compilação das Tabelas

A compilação das diversas tabelas presentes em um módulo de MIR resulta nos membros da classe que representa o módulo.

5.2.1 Tabelas de Funções

Funções são compiladas diferentemente de acordo com seu tipo. Funções estáticas e derivadas são compiladas em métodos segundo o exemplo apresentado na Figura 5.12. A passagem de parâmetros, do ponto de vista da função, é sempre por cópia, pois funções não possuem efeitos colaterais.

A compilação de uma função dinâmica definida no ambiente local produz como resultado uma tabela *hash* que associa os pontos no domínio da função aos valores da função nestes pon-

```
#ifndef MAS_H
#define MAS_H
// The includes for the needed modules
#include "Module1.h"
#include "Module2.h"
#include "Module3.h"
...
#include "ModuleN.h"
// The base class of all modules
#include "Module.h"
// Some needed auxiliary elements
#include <string>
#include <map>
#include <vector>
// The MAS class, which is a singleton.
class MAS {
  public:
    // Provides the unique instance of this class
    static MAS* getInstance();
    // Starts the agents
    void run();
  protected:
    // The private constructor
    MAS();
    // The unique instance of this class
    static MAS* mas;
    // The agents
    static std::map<std::string, Module*>* agents; };
#else
class MAS;
#endif
```

Figura 5.8: O arquivo de extensão `h` gerado a partir de um MAS.

```
// Includes the header
#include "MAS.h"
//-----------------------------------------------------------------------------
// Static initialization for MAS::mas
MAS* MAS::mas = 0;
//-----------------------------------------------------------------------------
// Static initialization of agents
std::map<std::string, Module*>* MAS::__agents = new std::map<std::string, Module*>();
//-----------------------------------------------------------------------------
MAS* MAS::getInstance() {
  if (mas == 0) mas = new MAS();
  return mas; }
//-----------------------------------------------------------------------------
void MAS::run() {
  // The threads for the agents
  pthread_t agentName1_t;
  pthread_t agentName2_t;
  ...
  pthread_t agentNameN_t;

  // Creates and starts each thread
  pthread_create(&agentName1_t, NULL, runModule, (*MAS::__agents)["agentName1"]);
  pthread_create(&agentName2_t, NULL, runModule, (*MAS::__agents)["agentName2"]);
  ...
  pthread_create(&agentNameN_t, NULL, runModule, (*MAS::__agents)["agentNameN"]);

  // Waits until all is over
  pthread_join(agentName1_t, NULL);
  pthread_join(agentName2_t, NULL);
  ...
  pthread_join(agentNameN_t, NULL);

  return; }
//-----------------------------------------------------------------------------
MAS::MAS() {
  // The agents (instances of the modules) are created
  (*MAS::_agents)["agentName1"] = new Module1(MAS::__agents);
  (*MAS::_agents)["agentName2"] = new Module2(MAS::__agents);
  ...
  (*MAS::_agents)["agentNameN"] = new ModuleN(MAS::__agents); }
//-----------------------------------------------------------------------------
// The main function, where everything starts...
int main(int argc, char *argv[]) {
  MAS* mas = MAS::getInstance();
  mas->run();
  return 0; }
```

Figura 5.9: O arquivo de extensão `cc` gerado a partir de um MAS.

```
all:  MAS.cc MAS.h Module1.cc Module1.h ... ModuleN.cc ModuleN.h Module.cc Module.h
      g++ Module.cc -c
      g++ Module1.cc -c
      g++ Module2.cc -c
      ...
      g++ ModuleN.cc -c
      g++ MAS.cc ./Module1.o ./Module2.o ... ./ModuleN.o ./Module.o -o mas -lpthread
```

Figura 5.10: O arquivo `makefile`.

```
MyModule::MyModule(std::map<std::string, Module*> agents, std::string agName) {
  // the initialization of the components of the module
  A __A = new A(agents, agName);
  B __B = new B(agents, agName);
  ...
  // the initialization of loacal 0-ary dynamic functions
  fa = NULL;
  fb = NULL;
  ...
  // the init rule
  ...
  flush(); }
```

Figura 5.11: O construtor de um módulo.

```
tipoRetorno nomeFuncao(tipoParametro1 &p1, ..., tipoParametroN &pN) {
  ... // código equivalente a expressao que define a funcao
  tipoRetorno v = ultimoValor;
  return v; }
```

Figura 5.12: Resultado da compilação de uma função estática ou derivada.

```
// a funcao
std::map<tipoParametro1, ... std::map<tipoParametroN, tipoRetorno> ... > > nomeFuncao;
// o metodo acessor
tipoRetorno acessor__nomeFuncao(tipoParametro1 &p1, ..., &tipoParametroN pN) {
  if (nomeFuncao.find(p1, ..., pN)) return nomeFuncao[p1]...[pN];
  else return valorPadrao; }
```

Figura 5.13: Resultado da compilação de uma função dinâmica.

tos e também um método acessor para esta tabela, conforme apresentado na Figura 5.13. Cabe ressaltar que, desta forma, para cada função dinâmica 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 de uma variável membro cujo tipo é o tipo de retorno da função. Existe também um método acessor para cada função dinâmica. A função do método acessor no lugar do acesso direto se justifica pelo fato que funções dinâmicas 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 dinâmica, conforme visto no Capítulo 4. A compilação de uma função dinâmica definida no ambiente global é de quase todo semelhante à compilação de uma função dinâmica definida no ambiente local. A diferença é que a tabela *hash* gerada e o método acessor são marcados como estáticos, e a inicialização não se dá no construtor, sendo feita diretamente no arquivo de extensão `cc` gerado. Note que a noção de escopo local e escopo global não diz respeito à visibilidade (se público ou privado), mas sim ao compartilhamento entre os agentes de mesmo tipo. Uma função dinâmica global é compartilhada por todos os agentes daquele módulo, ao passo que cada agente possui uma cópia individual de uma função dinâmica local.

Finalmente, funções externas são compiladas para métodos que apenas fazem a chamada à função definida externamente por meio da MNI, como apresentado na Seção 4.3.2.

5.2.2 Tipos

As classes utilizadas para a representação em memória dos tipos da linguagem MIR são apresentadas na Figura 5.14. À semelhança das regras, as classes para representação dos tipos são organizadas em uma hierarquia onde existe uma classes abstrata básica, Type, a partir da qual todas as classes de tipos devem derivar. Entretanto, esta derivação não é direta, mas se dá por meio duas classes abstratas, BasicType e TypeConstructor, de modo a agrupar de um lado os tipos simples e do outro os construtores de tipos. Esta estrutura faz uso dos padrões de projeto *Composite* e *Visitor*.

O mapeamento entre os tipos da linguagem MIR e os tipos em C++ é dado na Tabela 5.1.

Optou-se por desenvolver um ambiente de execução, a ser referenciado em cada módulo compilado, que provesse classes em C++ para cada tipo da linguagem MIR. No processo de análise semântica anterior à geração de código C++, a equivalência de tipos adotada é a *equivalência estrutural* [Car97].

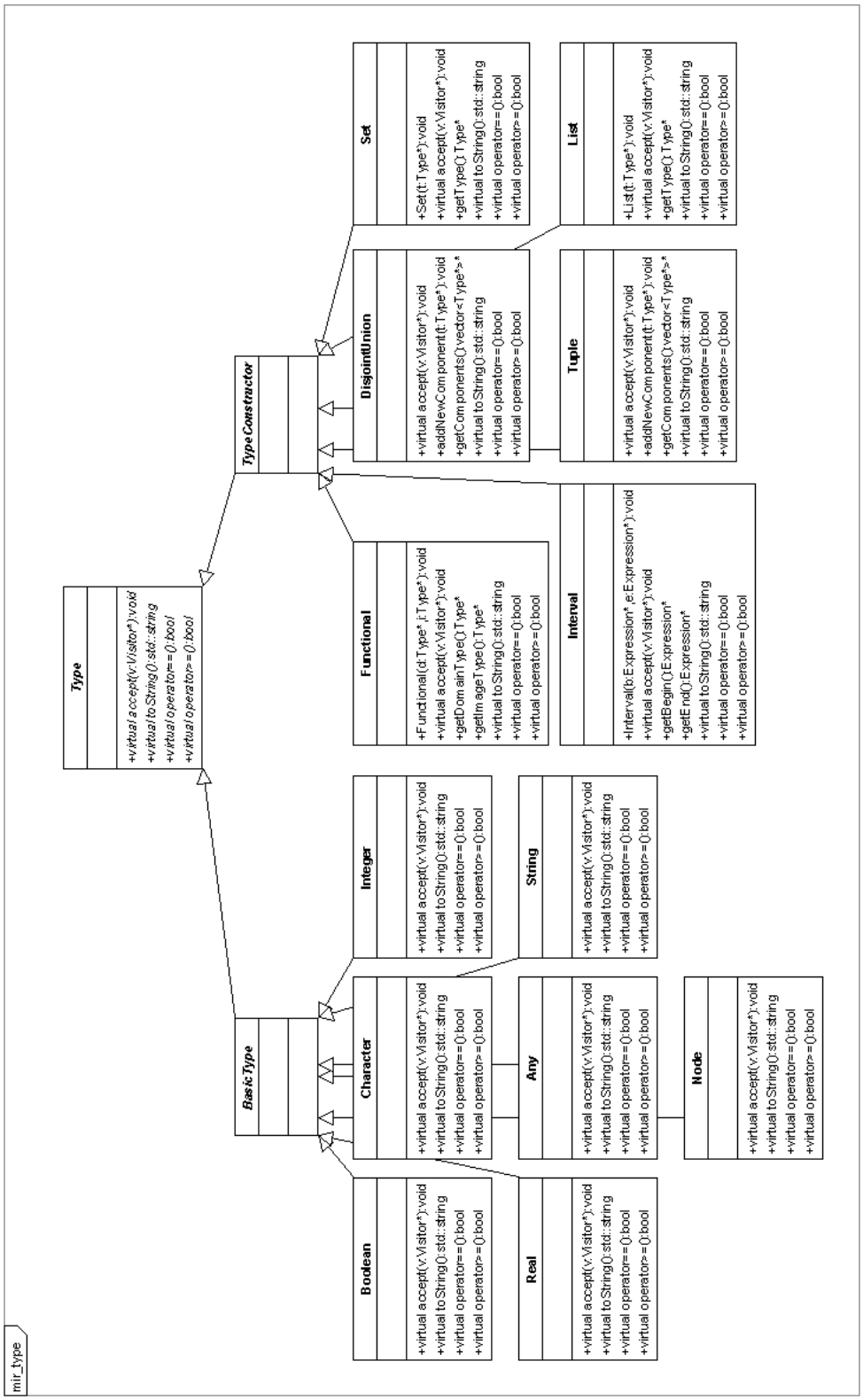

Figura 5.14: Diagrama de classes: tipos.

Linguagem MIR	**C++**
any	`Klar__Any`
boolean	`Klar__Boolean`
character	`Klar__Character`
integer	`Klar__Integer`
real	`Klar__Real`
string	`Klar__String`
disjoint union	`Klar__Any`
interval	`Klar__Interval`
$(T_1,\ldots,T_n)$	`Klar__Tuple`
$[T]$	`Klar__List<class T>`
$\{T\}$	`Klar__Set<class T>`
node	`Klar__Node`
abstraction	`Klar__Abstraction`

Tabela 5.1: Mapeamento de tipos previsto na compilação de MIR para C++.

```
void nomeSubMaquina(tipoParametro1 &p1, ..., tipoParametroN &pN) {
  bool __return = false;
  while (!__return)
  { // código equivalente à regra da submáquina
    ... }
  return; }
```

Figura 5.15: Resultado da compilação de uma submáquina.

5.2.3 Tabelas de Ações e Submáquinas

Ações e submáquinas também são traduzidas para métodos da classe correspondente ao módulo. Submáquinas são convertidas em métodos com tipo de retorno void, e possuem uma estrutura semelhante àquela apresentada na Figura 5.15.

Note que, como dito anteriormente, uma chamada a uma submáquina dispara uma regra de transição que é executada repetidamente até que uma regra *return* seja encontrada, cuja semântica é fazer `__return = true`, o que causa o término da execução da submáquina no final da transição corrente. Diferentemente, uma ação é compilada para um método à semelhança daquele apresentado na Figura 5.16, também com tipo de retorno void. Neste caso, não existe a noção de submáquina. Os parâmetros podem ser de entrada ou de entrada e saída, o que altera a forma como parâmetros são passados, se por cópia ou por referência. Esta classificação dos parâmetros é dada pela declaração da abstração na linguagem MIR, conforme visto no Capítulo 4.

5.2.4 Tabela de Semáforos

Um semáforo é declarado como uma estrutura do tipo `sem_t` da biblioteca `semaphore`. Esta é a biblioteca em C++ utilizada para prover os semáforos e as operações sobre estes. Semáforos são inicializados no construtor do módulo com o valor indicado na declaração do mesmo em

```
void nomeAcao(tipoParametro1 &p1, ..., tipoParametroN &pN) {
  // código equivalente à regra da ação
  // ...
  return; }
```

Figura 5.16: Resultado da compilação de uma ação.

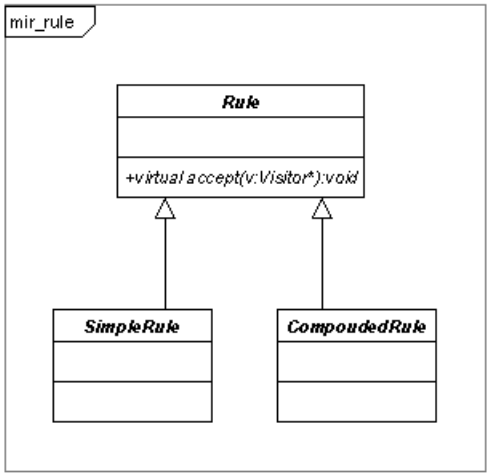

Figura 5.17: Diagrama de classes: regras de transição.

linguagem MIR.

5.3 Representação em Memória e Compilação de Regras

As regras de transição são representadas em memória por objetos instanciados a partir das classes mostradas nas Figuras 5.17, 5.18 e 5.19. Para cada regra presente na linguagem MIR existe uma classe correspondente que possui na sua interface pública os métodos acessores adequados.

A estrutura das classes que representam as regras respeita uma certa hierarquia. Toda classe de regra deriva da classe abstrata Rule, cujo único método é um método puramente virtual, a saber, accept(Visitor* v). Desta forma, toda classe de regra deve implementar este método que, como dito anteriormente, é utilizado pelo padrão de projeto *Visitor*. Fica assim estabelecida uma interface comum a todas as regras, o que é útil em contextos onde uma regra qualquer é esperada. Além disso, as regras são ainda separadas na estrutura em dois conjuntos: regras simples e regras compostas. Esta separação se dá por meio da derivação a partir da classe abstrata correspondente. Regras simples são apresentadas na Figura 5.18, ao passo que as regras compostas são apresentadas na Figura 5.19.

A estrutura das classes de regras segue o padrão de projeto *Composite*, apresentado na Seção 3.3.6. Este padrão define uma hierarquia de objetos como uma árvore e é particularmente útil quando deseja-se tratar uniformemente tanto objetos individuais quanto composições de

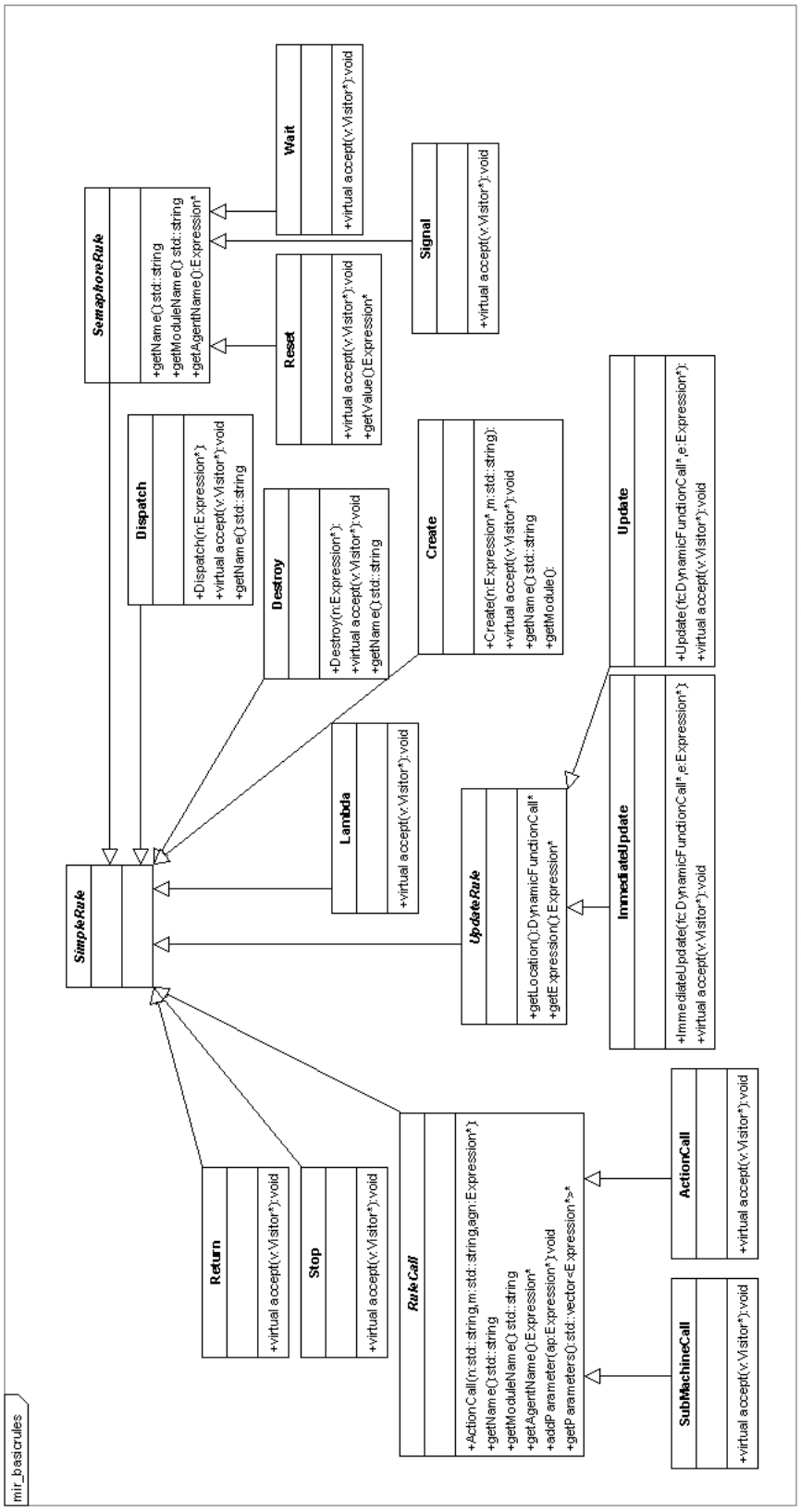

Figura 5.18: Diagrama de classes: regras básicas.

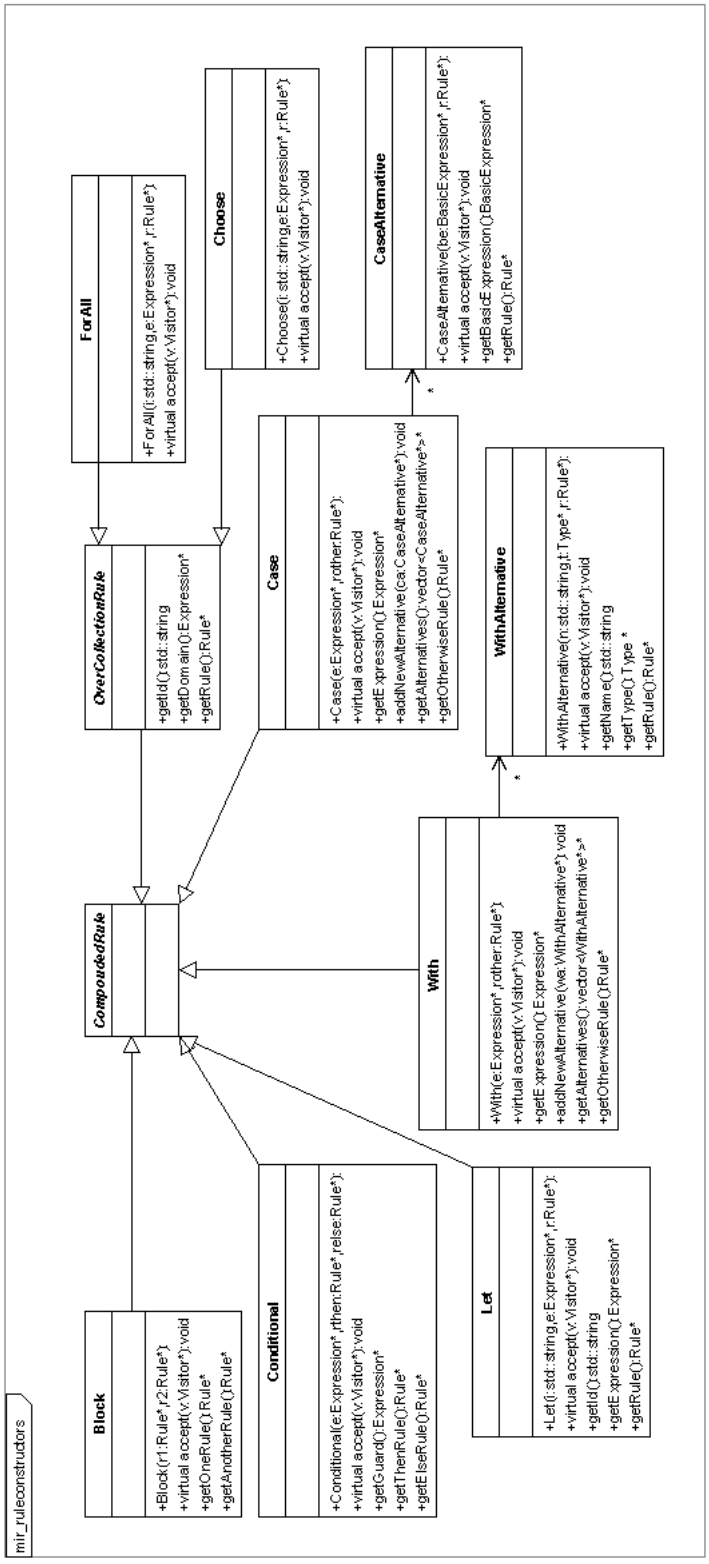

Figura 5.19: Diagrama de classes: construtores de regras (regras compostas).

```
// update
this->addNewUpdate((Klar__Any**)&__t, (exp));
// immediate update
__t = exp;
// create
if ((*agents)["nome"] != NULL) {
  pthread_cancel((*agents)["nome"]->thread);
  agents->erase("nome"); }
TipoAgente* __module__to__create = new TipoAgente(agents, "nome");
(*agents)["nome"] = __module__to__create;
// dispatch
TipoAgente* __module__to__dispatch = (*agents)["nome"];
pthread_create(&(__module__to__dispatch->thread), NULL, runModule, __module__to__create);
// destroy
if ((*agents)["nome"] != NULL) {
   pthread_cancel((*agents)["nome"]->thread);
   agents->erase("nome"); }
// stop
__stop__agent = true;
// return
__return__submachine = true;
// action call
Modulo::__actioncall__Nome(p1, ..., pN);
agente->__actioncall__Nome(p1, ..., pN);
// submachine call
Modulo::__submachine__Nome(p1, ..., pN);
agente->__submachine__Nome(p1, ..., pN);
// lambda
```

Figura 5.20: Exemplos de geração de código para as regras (1/4).

objetos.

A tradução de regras para código C++ é apresentada por meio de exemplos nas Figuras 5.20, 5.21, 5.22 e 5.23.

Na Figura 5.20 são apresentados exemplos de tradução das regras básicas. Uma regra de *update* 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, e é este artifício que permite simular o paralelismo intrínseco de uma regra de transição em um programa seqüencial. A tradução *immediate update* permite que as atualizações sejam percebidas imediatamente, constituindo-se de uma simples atribuição direta em C++. Esta regra deve ser utilizada apenas nas situações onde a localização atualizada não é consultada adiante. Caso contrário, o modelo pode ser desrespeitado. A criação de agentes, definida pela regra *create*, primeiro verifica se já existe um agente com o nome proposto para o agente a ser criado. Caso exista, este agente tem sua execução interrompida e é destruído. Em seguida, o agente é criado com o tipo adequado, a saber, o seu módulo, recebendo o contexto onde será inserido e o seu nome. Finalmente, o agente criado é colocado no contexto de agentes comuns a todos os agentes. Note que o agente é criado e situado no contexto, mas ainda não é disparado. Para tanto, existe a regra *dispatch*, que é traduzida em dois comandos C++: o primeiro recupera uma referência ao agente com o nome dado; o segundo dispara este agente

```
// conditional
if (exp) { ... // codigo equivalente ao  "then" }
else { ... // codigo equivalente ao  "else" }
// forall
tipoDoConjuntoOuLista __fa__nome__domain = ... /* expressao que define o conjunto ou lista */;
for (int __i__nome = 0; __i__nome < __fa__nome__domain->size(); __i__nome++)
{ // associa o elemento da vez com o nome do alias
  tipoElemento nome = (*__fa__nome__domain)[__i__nome];
  // a regra R do forall
  ... }
// choose
bool __ch__nome__found = false; // indica se encontrou elemento
tipoDoConjuntoOuLista __ch__nome__domain = ... /* expressao que define o conjunto ou lista */;
std::vector<int> __ch__nome__alreadychosen(); // vetor de ja escolhidos, inicialmente vazio
while (!__ch__nome__found & __ch__nome__domain->size() > __ch__nome__alreadychosen.size())
{ // obtem uma posicao aleatoria no dominio, desconsiderando ja escolhidos
  int __ch__nome__random = getRandom(__ch__nome__domain->size(), __ch__nome__alreadychosen);
  // associa o elemento escolhido com o nome do alias
  tipoElemento nome = (*__ch__nome__domain)[__ch__nome__random];
  // verifica se condicao eh satisfeita
  if ( ... /* expressao da condicao */) {
    __ch__nome__found = true;
    // a regra R do choose
    ... }
  else {
    // se nao achou, remove a posicao das consideradas para sorteio,
    // adicionando-a a ja escolhidos
    __ch__nome__alreadychosen.push_back(__ch__nome__random); }
}
```

Figura 5.21: Exemplos de geração de código para as regras (2/4).

propriamente, iniciando a execução de sua regra de transição dentro de uma *thread*. Um agente em execução é destruído por meio da regra *destroy*, cujo código correspondente consiste em procurar o agente de nome dado no contexto de agentes, interromper a sua execução e então apagá-lo. A execução de uma regra de transição pode ser interrompida ao se executar a regra *stop*, cuja tradução resulta em uma atribuição de *true* à variável reservada `__stop__agent`. Esta variável é consultada no início de cada transição para verificar se o agente deve ou não executar a transição mais uma vez. De forma semelhante, *return* é traduzido na atribuição de *true* à variável reservada `__return__submachine`, que é consultada por uma submáquina a cada iteração para saber se a execução deve continuar ou não. Uma *action call* e uma *submachine call* são convertidas nas chamadas dos métodos correspondentes, passando-se os parâmetros atuais. A tradução de *lambda* não produz nenhum código.

A Figura 5.21 apresenta os primeiros exemplos da tradução das regras compostas. Uma *conditional* é traduzida para um comando `if` de C++, onde a guarda é avaliada como a expressão do `if` e os códigos para as regras correpondentes aos braços `then` e `else` são produzidos dentro dos blocos correspondentes ao `then` e `else` do `if`. O *forall* tem a semântica de disparar em paralelo a sua regra de transição para cada elemento de um conjunto dado, de tal forma que em cada instância da regra o elemento do conjunto para aquela instância está associado

ao nome especificado. A tradução desta regra segue o exemplo apresentado na Figura 5.21. A expressão que define o domínio é avaliada e seu valor é associado a um nome especial. Este nome especial, e outros a seguir, são constituídos de prefixos e sufixos que cercam o nome do *alias*, de modo que aninhamentos destes comandos não sofrem com choque de nomes. Em seguida, para cada elemento do domínio, a regra do *forall* é executada, mas antes o elemento atual á associado com seu *alias*. 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 regra *choose* tem uma tradução um pouco mais complexa. Inicialmente, é gerada uma variável que indica se foi encontrado um elemento que satisfaça a condição dada. Em seguida, a expressão que denota o domínio de busca é avaliada e associada a um identificador. É também criado um conjunto que indica quais elementos do domínio já foram examinados. Estabelecido este 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. A busca começa por meio de uma chamada a getRandom, que fornece um inteiro aleatório entre 0 e o número especificado no primeiro argumento, e que não esteja na lista passada como segundo argumento, que contém os inteiros já gerados anteriormente. Este artifício evita o exame repetido de elementos do domínio. Em seguida, o elemento escolhido é associado ao *alias* do *choose*, e a condição é avaliada. Caso esta 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 adicionado à lista de índices de elementos já escolhidos e então uma nova tentativa de escolha é feita. Esgotado-se os elementos de modo que nenhum satisfaz a condição dada, nada é executado.

A Figura 5.22 apresenta exemplos de tradução de mais algumas regras compostas. A tradução da regra *let* cria um bloco C++ e dentro dele avalia a expressão de *let*, associando o valor desta ao *alias* especificado na regra. Em seguida, o código da regra é gerado, tendo acesso ao valor denotado pela expressão de *let* por meio do *alias*. O término da regra do *let* é seguido pelo fechamento do bloco C++, determinando-se o fim do escopo da expressão do *let*. A tradução da regra *case* resulta primeiramente na avaliação da expressão do *case* e sua associação com 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. Finalmente, a regra padrão é traduzida dentro de um bloco else sem guarda, que será executado caso todas as guardas anteriores falhem. A tradução da regra *with* é semelhante à tradução da regra *case*, porém duas diferenças devem ser ressaltadas. A primeira 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, desenvolvida como parte da biblioteca de *runtime*, e que se utiliza das funcionalidades disponibilizadas pela RTTI (*Real Time Type Information*) de C++. O critério de comparação é, via de regra, a comparação estrutural. A exceção fica por conta dos tipos construídos por meio de *node*, que são comparados de acordo com o seu rótulo. A segunda diferença é que, em cada alternativa, o primeiro código gerado é o *typecast* da expressão avaliada para o tipo encontrado e sua associação com o identificador dado. Desta forma, o valor da expressão se torna disponível dentro da alternativa. A tradução de uma regra *block* 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.

```
// let
{ // associa expressao com o nome do alias
  tipoExpressao nome = ... /* avaliacao da expressao */;
  // a regra R do let
  ... }
// case
tipoExpressao caseExpression = ... /* avaliacao da expressao */;
if (caseExpression == ... /* expressao 1*/) {
  // a regra correspondente
  ... }
else if (caseExpression == ... /* expressao 2*/) {
  // a regra correspondente
  ... }
...
else {
  // a regra padrao caso nenhuma alternativa corresponda a expressao
  ... }
// with
tipoExpressao withExpression = ... /* avaliacao da expressao */;
if (typeEquivalent(withExpression, type1)) {
  // associa a expressao com um nome dado, fazendo o type cast apropriado
  type1 __wa__nome = *(dynamic_cast<type1*>(withExpression));

  // a regra correspondente
  ... }
else if (typeEquivalent(withExpression, type2)) {
  // associa a expressao com um nome dado, fazendo o type cast apropriado
  type2 __wa__nome = *(dynamic_cast<type2*>(withExpression));
  // a regra correspondente
  ... }
...
else {
  // a regra padrao caso nenhuma alternativa corresponda a expressao
  ... }
// block
... ; // código correspondente a primeira regra
... ; // código correspondente a segunda regra
```

Figura 5.22: Exemplos de geração de código para as regras (3/4).

```
// reset
sem_init(&(NomeModulo::__semaphore__NomeSemaforo), 0, valorSemaforo);
sem_init(&(agente->__semaphore__NomeSemaforo), 0, valorSemaforo);
// signal
sem_post(&(NomeModulo::__semaphore__NomeSemaforo));
sem_post(&(agente->__semaphore__NomeSemaforo));
// wait
sem_wait(&(NomeModulo::__semaphore__NomeSemaforo));
sem_wait(&(agente->__semaphore__NomeSemaforo));
```

Figura 5.23: Exemplos de geração de código para as regras (4/4).

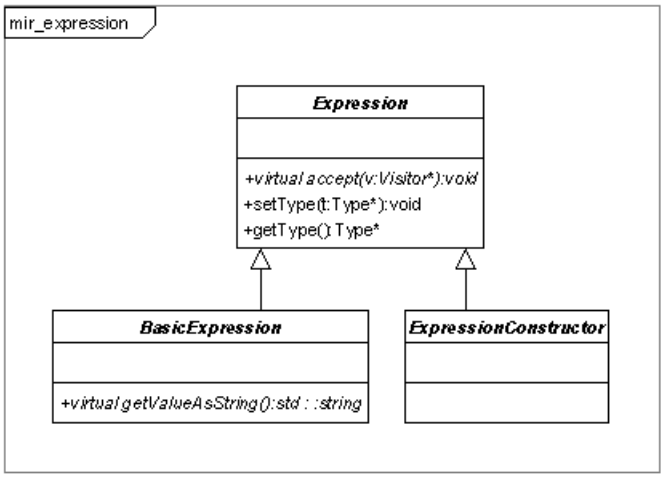

Figura 5.24: Diagrama de classes: expressões.

A Figura 5.23 apresenta exemplos da tradução das regras relativas às operações com os semáforos. A regra *reset* é traduzida em uma chamada à função sem_init da biblioteca de semáforos utilizada. Esta chamada recebe como argumento uma referência ao semáforo a ser inicializado, um indicador de que o semáforo é compartilhado por várias *threads* (no caso, este indicador é o valor 0), e o valor com o qual o semáforo será inicializado. Na Figura 5.23 são apresentados dois exemplos de cada um dos comandos de semáforos: o primeiro acessa um semáforo declarado como global, que está associado à classe do módulo, enquanto o segundo acessa um semáforo local, associado ao objeto que representa o agente do módulo. Uma regra *signal* é traduzida em uma chamada a uma função sem_post da biblioteca de semáforos utilizada. Esta função recebe como argumento uma referência ao semáforo, e caso nenhuma *thread* esteja bloqueada esperando o semáforo, incrementa o valor do semáforo em uma unidade. Caso contrário, uma das *threads* bloqueadas neste semáforo é liberada para execução. A regra *wait* é traduzida para uma chamada à função sem_wait, que recebe como argumento uma referência ao semáforo a ser consultado. Caso o valor do semáforo seja 0, a *thread* da execução do agente é bloqueada. Caso contrário, o valor do semáforo é decrementado e a *thread* da execução do agente prossegue normalmente.

5.4 Representação em Memória e Compilação de Expressões

A Figura 5.24 mostra a estrutura básica das classes para a representação em memória de expressões. O mesmo recurso de fornecer uma classe básica a partir da qual as classes de expressão derivem indiretamente por meio das classes BasicExpression e ExpressionConstructor é utilizado.

As expressões básicas são os literais,a expressão *self* e a expressão *undef*, apresentados na Figura 5.25. Estas sempre figuram como folhas em uma árvore que representa uma expressão.

Os construtores de expressão são apresentados na Figura 5.26. Constituem este grupo os operadores unários, os operadores binários, as chamadas a funções, os agregados, os *alia-*

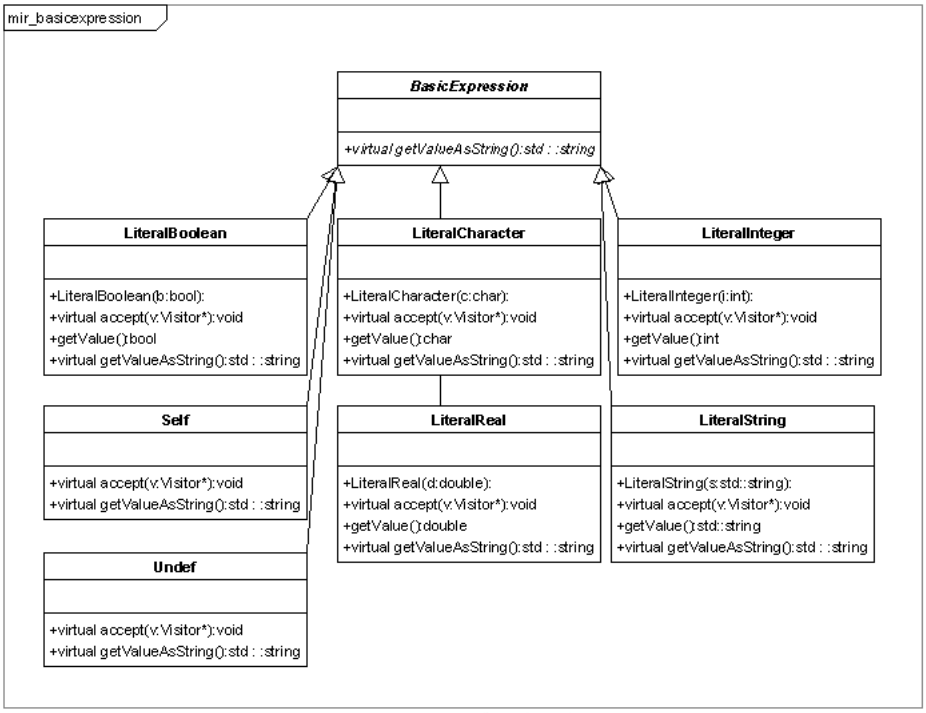

Figura 5.25: Diagrama de classes: expressões básicas (literais).

ses, a expressão *if*, a expressão *case*, a expressão *with* e a expressão *let*, correspondentes às construções da linguagem MIR.

As classes para os operadores unários são apresentadas na Figura 5.28, e derivam de UnaryOp. As classes para os operadores binários são apresentadas na Figura 5.27, e todo operador binário deriva de BinaryOp. As classes de chamadas a funções são apresentadas na Figura 5.29, e derivam de FunctionCall. Por sua vez, os agregados possuem uma classe comum, Agregate, apresentada na Figura 5.31. Finalmente, os *aliases* são agrupados sob a classe abstrata Alias, como mostrado na Figura 5.30. Esta estratégia permite que código comum seja delegado às classes de base, além de fornecer uma interface comum a grupos relacionados de classes.

As Figuras 5.32, 5.33, 5.34 e 5.35 apresentam exemplos da tradução de expressões em MIR para código C++. A Figura 5.32 apresenta como as expressões mais simples são traduzidas. A tradução de um literal resulta na declaração de um valor do tipo do literal. A expressão *self* é traduzido para uma chamada ao método que retorna o nome do agente, ao passo que *undef* é mapeado para o valor NULL. Um *alias* é traduzido para a chamada à variável associada ao alias, obtendo o seu valor. 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 chamada a uma função é traduzida para a avaliação de cada parâmetro atual, seguida da chamada ao método que está associado à função. Note que a informação se a passagem é por valor ou por referência está na declaração da função, e não em sua chamada. O literal de um agregado é traduzido para a seguinte seqüência de comandos: primeiramente, a expressão de cada componenente é avaliada. Em seguida, o literal do agregado é declarado, e cada componente é adicionado. A projeção de uma tupla é traduzida primeiramente para a avaliação da expressão que resulta na tupla. Em seguida, uma variável com o tipo do i-ésimo componente projetado é declarada, e esta recebe a projeção do i-ésimo componente.

Dando continuidade, a Figura 5.33 apresenta mais alguns exemplos de tradução de expressões. Uma expressão condicional *if exp* é traduzida de forma semelhante à tradução de uma regra *conditional*. A diferença é que neste caso existe um cuidado extra com a obtenção do valor da expressão *if exp*. Inicialmente, é criada uma variável com o tipo de retorno da expressão condicional. 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 condicional 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 condicional. A geração de código para o braço else é análoga.

A tradução da expressão *let exp* acontece da seguinte forma. Inicialmente, é criada uma variável com o tipo de retorno da expressão *let exp* para conter o resultado da expressão como um todo. Em seguida, é criado um bloco C++ e dentro dele é avaliada a expressão que denota o valor associado ao nome dado, e então esta associação é feita. Em seguida, o código da avaliação da expressão principal é gerado, tendo acesso ao valor denotado pela expressão associada ao nome por meio do *alias*. O término da avaliação da expressão principal do *let exp* é seguido pela atribuição do valor resultante à variável criada para conter o valor da expressão como um todo. O fechamento do bloco C++ determina o fim do escopo da *alias*.

A Figura 5.34 apresenta um exemplo da tradução de uma expressão *case exp*. Inicialmente, é criada uma variável com o tipo de retorno da expressão *case exp* para conter o resultado da

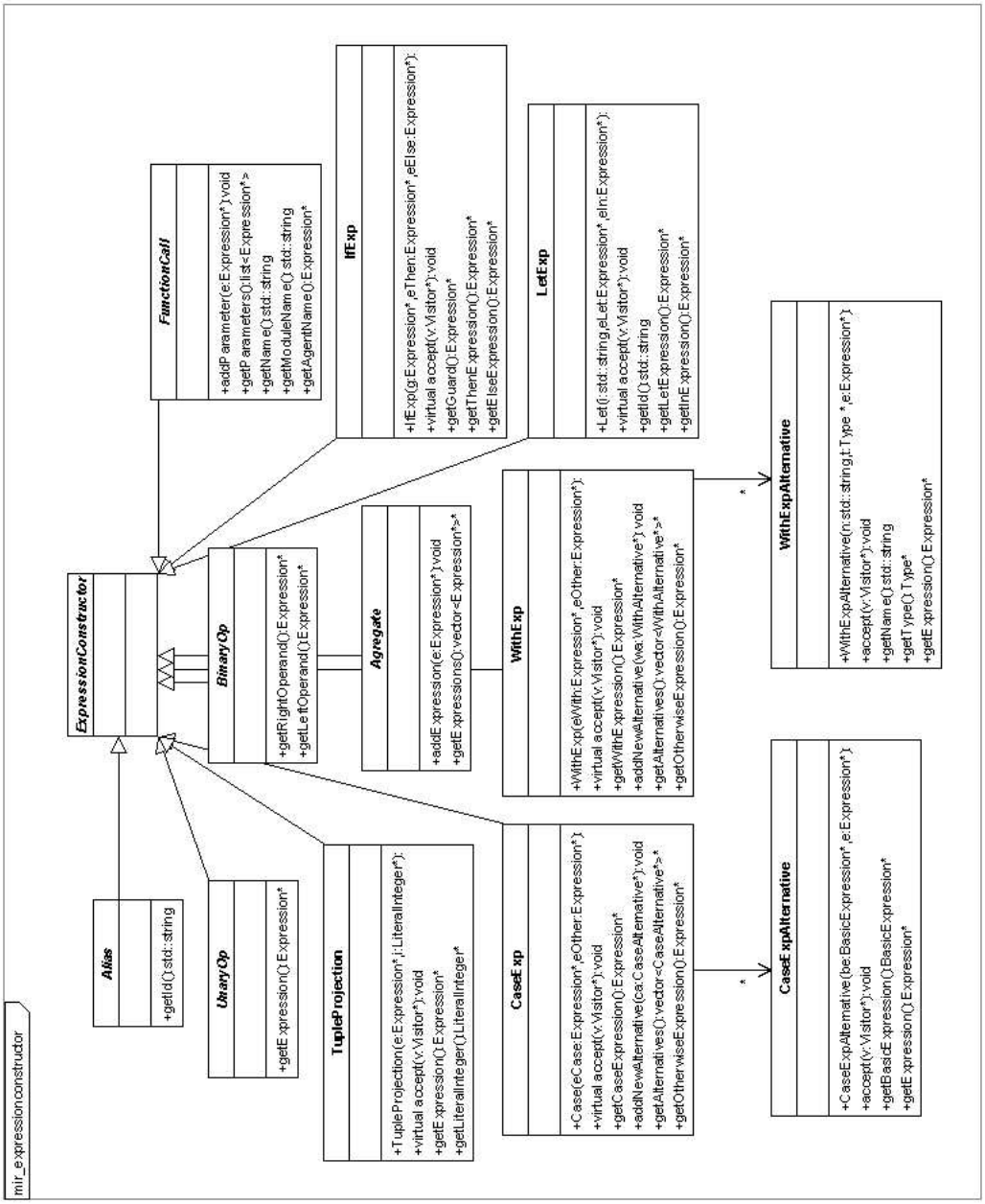

Figura 5.26: Diagrama de classes: construtores de expressões (expressões compostas).

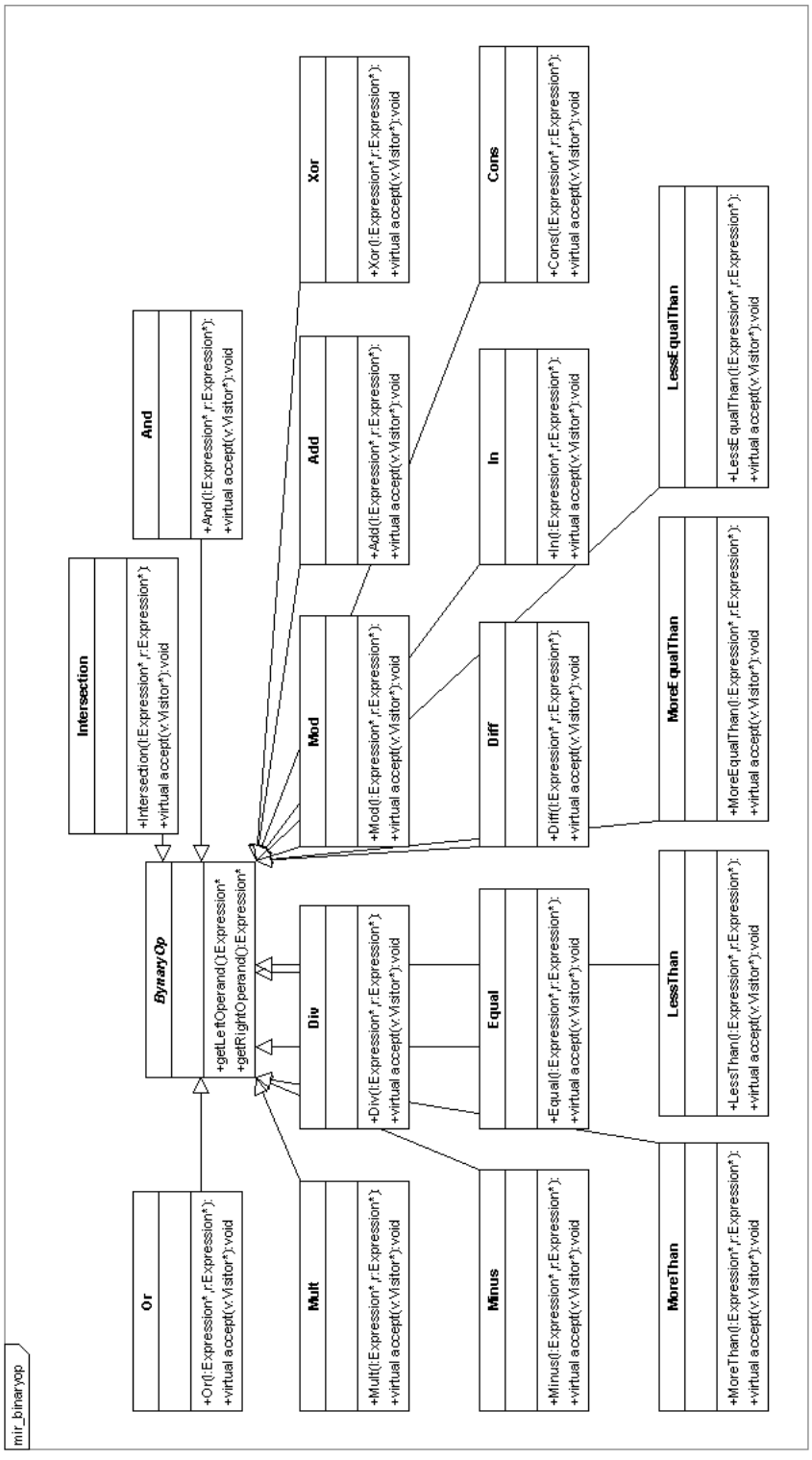

Figura 5.27: Diagrama de classes: operadores binários.

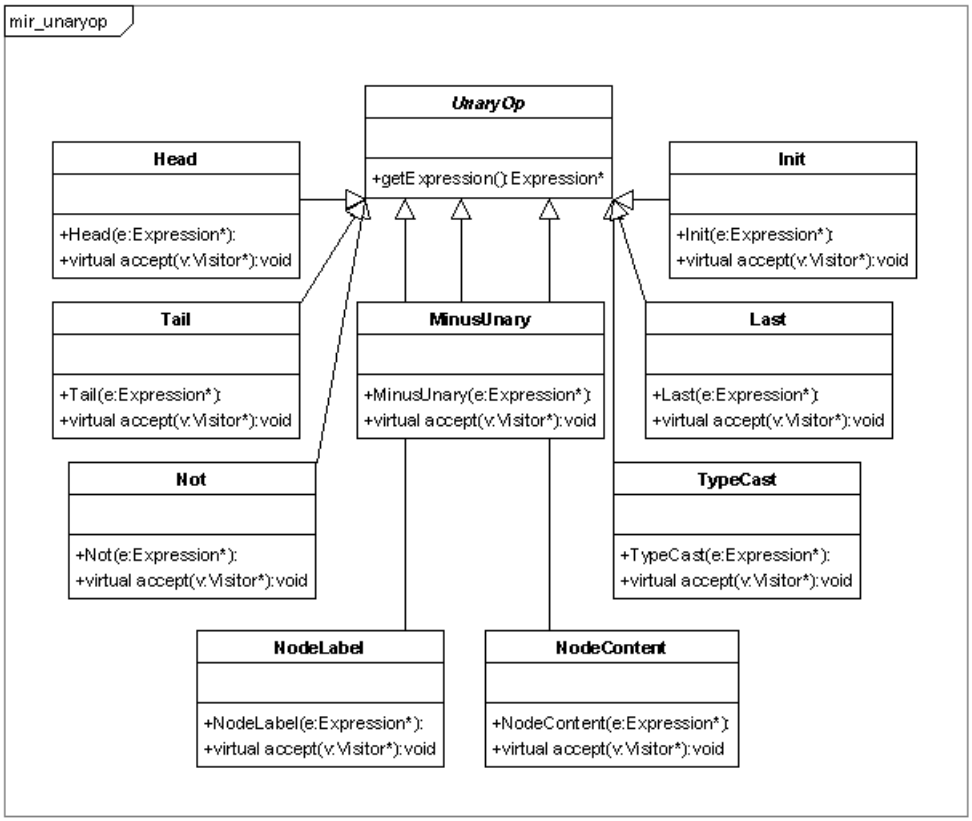

Figura 5.28: Diagrama de classes: operadores unários.

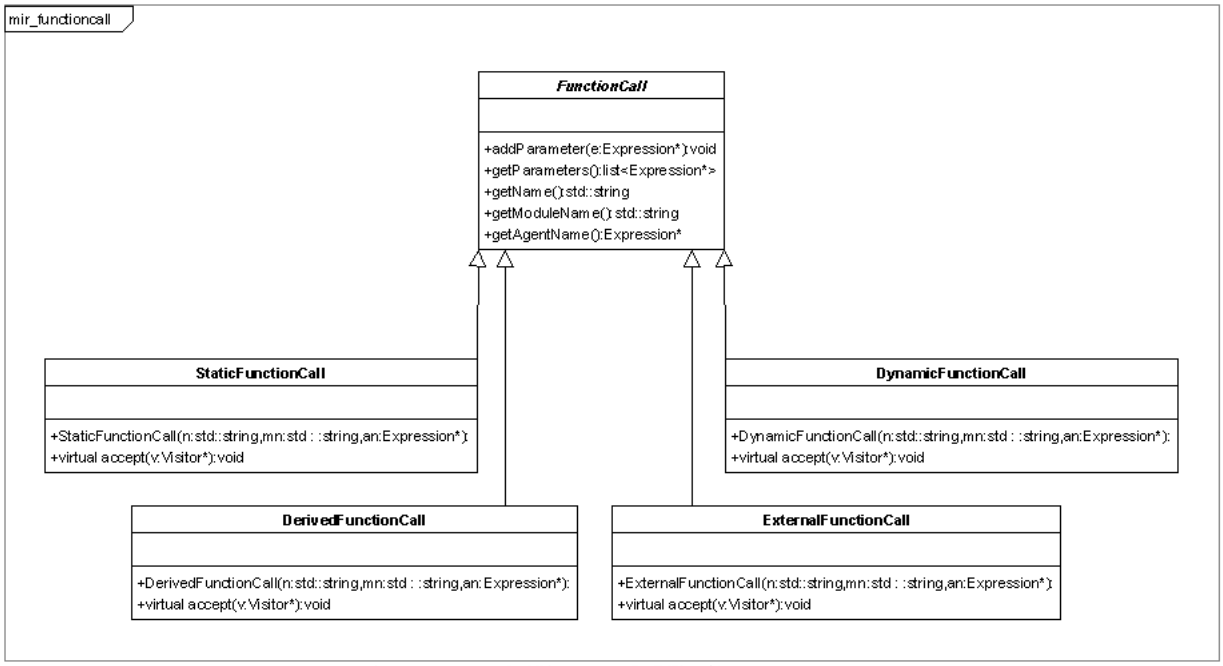

Figura 5.29: Diagrama de classes: chamadas de função.

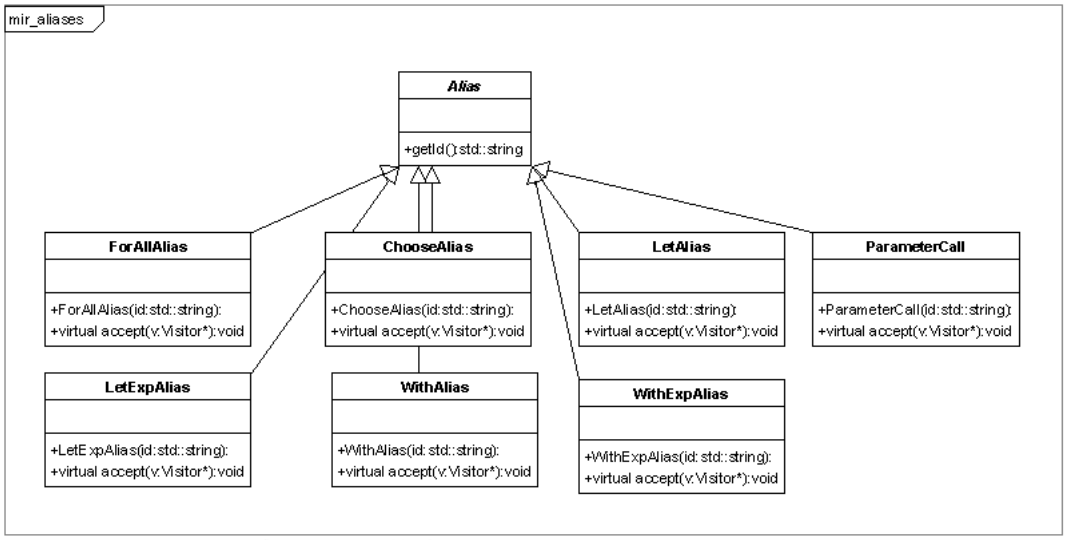

Figura 5.30: Diagrama de classes: *aliases*.

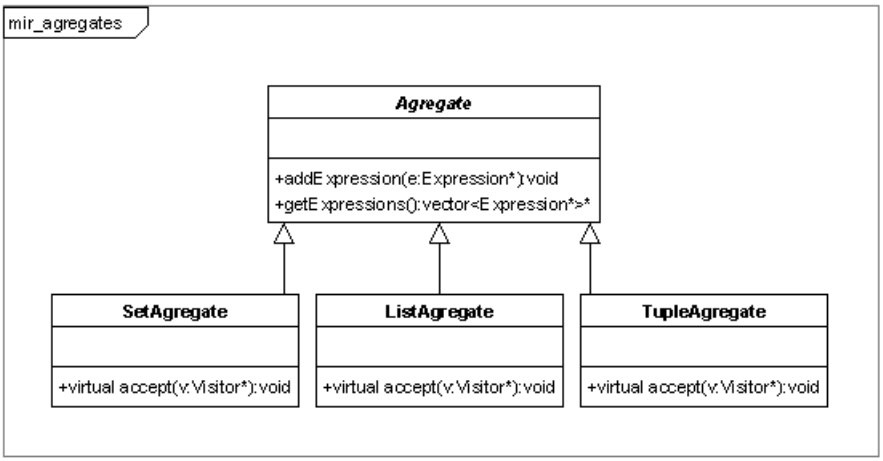

Figura 5.31: Diagrama de classes: agregados.

expressão como um todo. 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 diponibilizado externamente por meio da atribuição à variável declarada para conter o resultado da expressão *case exp* como um todo. Além disso, é gerado também um bloco else que corresponde ao caso em que nenhuma alternativa foi escolhida.

A Figura 5.35 apresenta um exemplo da tradução de uma expressão *with exp*. A tradução da expressão *with exp* é semelhante à tradução da expressão *case exp*, porém duas diferenças devem ser ressaltadas. A primeira 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, desenvolvida como parte da biblioteca de *runtime*, e que se utiliza das funcionalidades disponibilizadas pela RTTI (*Real Time Type Information*) de C++. O critério de comparação é, via de regra, a comparação estrutural. A exceção fica por conta dos tipos construídos por meio de *node*, que são comparados de acordo com o seu rótulo. A segunda diferença é que, em cada alternativa, o primeiro código gerado é o *typecast* da expressão avaliada para o tipo encontrado e sua associação com o identificador dado. Desta forma, o valor da expressão se torna disponível dentro da alternativa.

5.5 Persistência

Uma estrutura em memória construída com objetos instanciados a partir das classes para representação de programas da linguagem MIR é passível de ser *persistida*[2]. A persistência,

[2]Por *persistência* entende-se a capacidade de colocar as informações de um objeto ou estrutura de objetos em um formato seqüencial, de modo que este possa ser gravado em disco, transmitido via rede, gravado em um banco de dados etc. Uma representação textual de um objeto, como um arquivo XML, por exemplo, constitui uma forma de persistência. Em alguns contextos, a persistência também é conhecida como *serialização*.

```
// literais
Klar__Integer t1(0);
// self
Klar__String t1((std::string)(this->getAgentName()));
// undef
Klar__Any *t1 = NULL;
// alias (neste exemplo, um alias de um forall)
Klar__Int* t1 = (*__fa__nome__domain)[__i__nome];
// unary op (neste exemplo, uma inversão de sinal)
Klar__Integer* t1 = ... ; // avalia a expressão do operando
Klar__Integer t2 = ( - (*t1)); // o resultado está disponível em t2
// binary op (neste exemplo, uma soma)
Klar__Integer* t1 = ... ; // avalia a expressão do primeiro operando
Klar__Integer* t2 = ... ; // avalia a expressão do segundo operando
Klar__Integer t3 = ((*t1) + (*t2)); // o resultado está disponível em t3
// function call
tipoParametro1 t1 = ... ; // avalia expressao do primeiro parametro
...
tipoParametro1 tN = ... ; // avalia expressao do ultimo parametro
Klar__Real tN+1 = __funcao((t1), ..., (tN)); // chama a função, diponibilizando
                                             // o resultado em tN+1
// agregate (neste exemplo, uma lista)
tipoElemento1 t1 = ... ;  // avalia expressao do primeiro elemento
...
tipoElemento1 tN = ... ; // avalia expressao do ultimo elemento
Klar__List tN+1(); // declara a lista, diponivel em tN+1
tN+1.push_back(t1); // adiciona o primeiro elemento
...
tN+1.push_back(tN); // adiciona o ultimo elemento
// tuple projection
Klar__Tuple t1 = ... ; // avalia a expressao que resulta na tupla
int t2 = ... ; // o inteiro que indica qual componente da tupla deve ser projetado
tipoDoElementoProjetado* t3 = dynamic_cast<tipoDoElementoProjetado*>(t1..getElementAt(t2));
```

Figura 5.32: Exemplos de geração de código para as expressões (1/4).

```
// ifexp
tipoIfExp* t1; // declara variavel com o tipo de retorno da expressão if
Klar__Boolean t2 = ... ; // avalia a guarda
if (t2.getValue()) {
  t3 = ... ; // avalia a parte "then" da expressao
  t1 = t3; // copia resultado da avaliacao para resultado do if
}
else {
  t4 = ... ; // avalia a parte "else" da expressao
  t1 = t4; // copia resultado da avaliacao para resultado do if
}
... // neste ponto, o valor da expressao esta disponivel em t1
// letexp
tipoLetExp* t1; // declara variavel com o tipo de retorno da expressão let
{ // associa expressao com o nome do alias
  tipoExpressao nome = ... /* avaliacao da expressao */;
  // avalia expressao do let
  t2 = ... ;
  // torna o resultado da avaliacao disponivel
  t1 = t2; }
... // neste ponto, o valor da expressao esta disponivel em t1
```

Figura 5.33: Exemplos de geração de código para as expressões (2/4).

```
// casexp
tipoCaseExp* t1; // declara variavel com o tipo de retorno da expressão case
tipoExpressaoGuarda t2 = ... ; // avalia expressao a ser comparada
if (t2 == ... /* expressao 1*/) {
  // avalia a expressao correspondente
  t3 = ... ;
  // torna o resultado da avaliacao disponivel em t1
  t1 = t3;
}
else if (t2 == ... /* expressao 2*/) {
  // avalia a expressao correspondente
  t3 = ... ;
  // torna o resultado da avaliacao disponivel em t1
  t1 = t3;
}
...
else {
  // a expressao padrao caso nenhuma alternativa corresponda a expressao a ser comparada
  t3 = ... ;
  // torna o resultado da avaliacao disponivel em t1
  t1 = t3;
}
... // neste ponto, o valor da expressao esta disponivel em t1
```

Figura 5.34: Exemplos de geração de código para as expressões (3/4).

```
// withexp
tipoWithExp* t1; // declara variavel com o tipo de retorno da expressão with
tipoExpressaoGuarda t2 = ... ; // avalia expressao a ser comparada
if (typeEquivalent(t2, type1)) {
  // associa a expressao com um nome dado, fazendo o type cast apropriado
  type1 __wa__nome = *(dynamic_cast<type1*>(withExpression));
  // avalia a expressao correspondente
  t3 = ... ;
  // torna o resultado da avaliacao disponivel em t1
  t1 = t3;
}
else if (typeEquivalent(withExpression, type2)) {
  // associa a expressao com um nome dado, fazendo o type cast apropriado
  type2 __wa__nome = *(dynamic_cast<type2*>(withExpression));
  // avalia a expressao correspondente
  t3 = ... ;
  // torna o resultado da avaliacao disponivel em t1
  t1 = t3;
}
...
else {
  // a expressao padrao caso nenhuma alternativa corresponda a expressao a ser comparada
  t3 = ... ;
  // torna o resultado da avaliacao disponivel em t1
  t1 = t3;
}
... // neste ponto, o valor da expressao esta disponivel em t1
```

Figura 5.35: Exemplos de geração de código para as expressões (4/4).

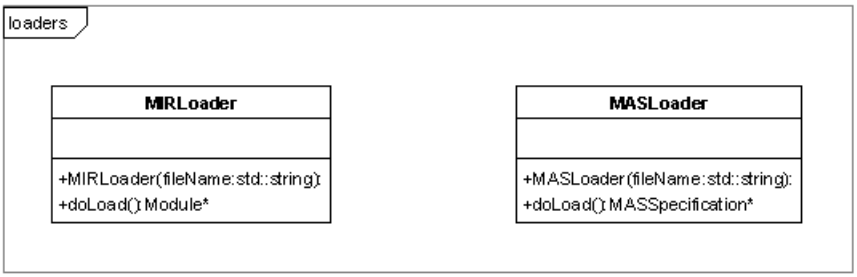

Figura 5.36: Diagrama de classes: *loaders*.

neste caso, consiste em converter uma estrutura em memória para um arquivo XML que a represente. A sintaxe deste arquivo não é outra senão aquela apresentada na Seção 4.4.1. A classe Serialize é fornecida para esta tarefa. Ela implementa a classe abstrata Visitor, que fornece os métodos de *callback* necessários à implementação do padrão nas classes para representação em memória de uma especificação em linguagem MIR.

Em complemento a este visitor, foi implementada também as classes ModuleLoader e MASLoader, que fazem o caminho inverso: a partir de um arquivo de um módulo (com extensão `mod`) ou um arquivo de disparador de agentes (com extensão `mas`) válido, estas classes fornecem uma estrutura em memória correspondente. A Figura 5.36 apresenta estas classes. Não foram utilizadas ferraments de *XML Data Binding*, pois não foram encontradas ferramentas adequadas durante a codificação realizada neste trabalho. Os *parsers* do *loaders* foram geradas com o Lex e o Bison, e as classes para representação em memória foram codificadas neste trabalho, conforme apresentado neste capítulo.

5.6 Conclusão

Neste capítulo foram apresentadas duas questões principais: a forma como programas em MIR são representados em memória e também como estes programas são traduzidos para C++ por meio do visitor CodeGenerator. A primeira informação é importante caso se deseje montar as especificações MIR diretamente e passá-las ao kℓar sem o intermédio do arquivo XML. A outra questão tem duas funções básicas. Primeiramente, mostrar como as construções de MIR são convertidas em C++ ajuda a tornar mais clara a sua semântica. Em segundo lugar, o registro destas traduções constitui um manual básico a ser seguido em caso de se desejar alterar a forma como a tradução para C++ é realizada.

De posse destas informações, pode-se agora adentrar no assunto das otimizações em si, que é o motivo principal da elaboração do kℓar.

Capítulo 6

Arquitetura do kℓar

Conforme apresentado na Seção 1.2.2, o kℓar foi desenvolvido dentro de um contexto maior. A Figura 6.1 ilustra o seu contexto.

Na parte superior esquerda encontra-se esquematizado o arcabouço ACOA [Lob05]. Na parte inferior direita, dentro do retângulo pontilhado, está representado o kℓar. O acoplamento entre os dois sistemas acontece por meio da linguagem MIR [Tir00, OBB04a, MBB+05]. Os losangos representam as otimizações para o modelo ASM [OBB04b, Oli04].

Ainda na Figura 6.1, é possível observar os elementos principais que compõem a arquitetura do kℓar. Em primeiro lugar, para que especificações MIR textuais escritas em XML sejam entendidas pelo arcabouço, é preciso que elas sejam carregadas em memória. Desta forma, um dos componentes da arquitetura do kℓar é o *carregador* de especificações, que lê uma especificação em XML e a partir dela fornece uma representação em memória equivalente. Este carregador pode inclusive ser utilizado fora do contexto do kℓar.

Um outro elemento importante da arquitetura do kℓar é o componente que cuida da *persistência* de especificações MIR. Dada uma especificação MIR em memória, este componente gera em disco a representação textual correspondente. Este componente também pode ser utilizado fora do contexto do kℓar.

O *gerador de código C++* é o componente responsável pela geração de código executável correspondente a uma especificação MIR. Dada uma especificação em memória, este componente gera os arquivos C++ com a semântica apropriada à especificação. Dentro do contexto do kℓar, este componente é utilizado para gerar código a partir de uma especificação otimizada.

O núcleo das otimizações é o componente do kℓar que recebe as otimizações acopladas. Este componente é responsável por consultar o arquivo de configurações, carregar as otimizações especificadas neste arquivo, ordená-las apropriadamente e aplicá-las à especificação original, obtendo assim uma especificação otimizada.

Novos componentes podem eventualmente ser adicionados. Para tanto, deve-se utilizar o padrão visitor. A implementação deste padrão resulta em um componente facilmente acoplável ao kℓar, e a nova funcionalidade pode ser então disponibilizada na classe de fachada por meio de herança e delegação. Este artifício constitui um *hotspot* onde novas funcionalidades podem ser acionadas.

A utilização do kℓar como o arcabouço a que ele se propõe é bastante simples. Funcionalmente, uma aplicação qualquer que queira utilizar o kℓar para a otimização de especificações ASM e geração de código executável que reflita a semântica da especificação otimizada deve seguir os seguintes passos:

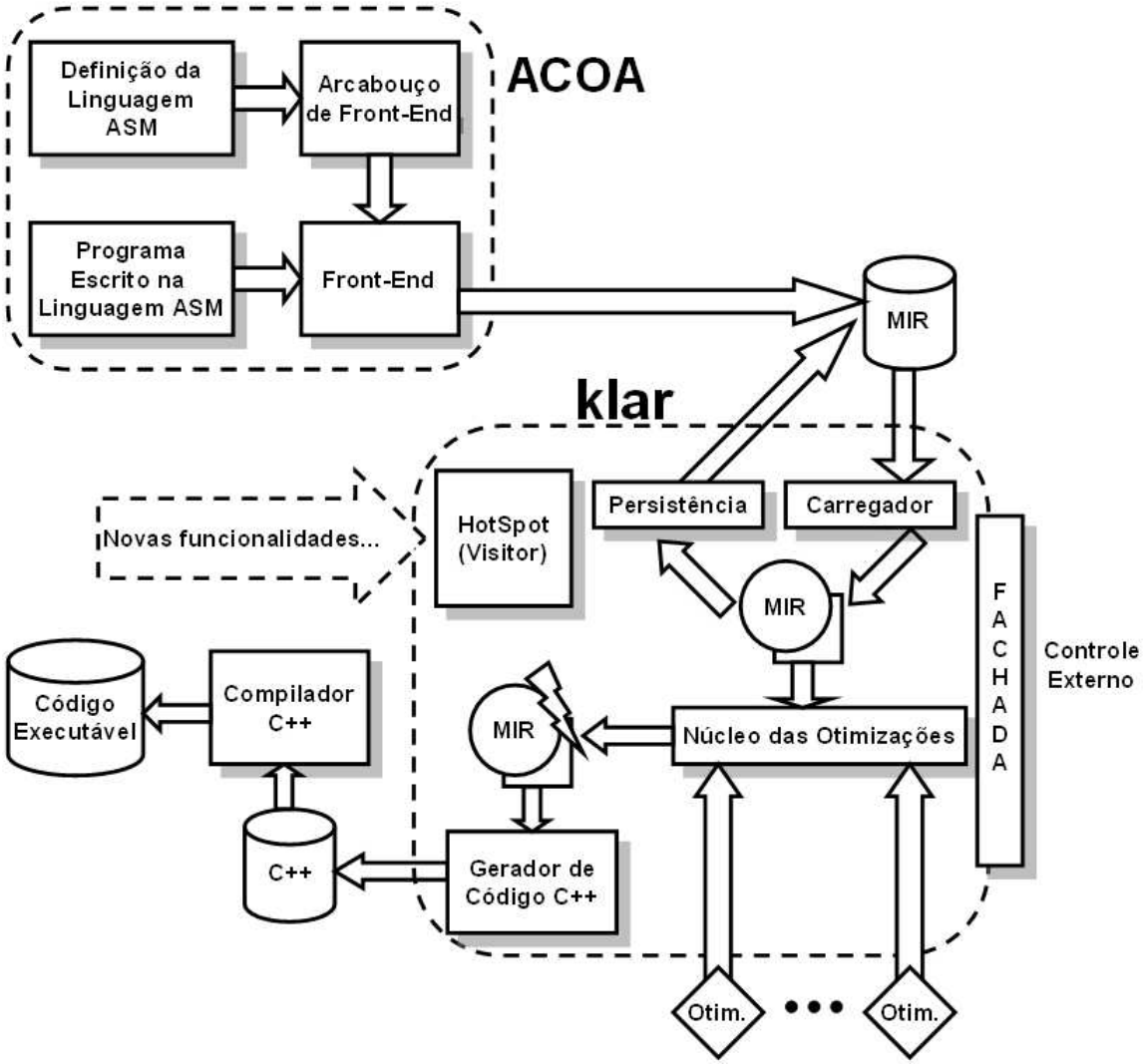

Figura 6.1: O uso do kℓar no contexto do LLP - DCC - UFMG.

1. Instanciar a classe de fachada que dá acesso ao arcabouço, permitindo o controle do mesmo.

2. Obter uma representação da especificação ASM a ser otimizada em linguagem MIR. Esta especificação pode ser obtida de duas maneiras:

   (a) Por meio do carregador de especificações fornecido pelo arcabouço, que tem a capacidade de ler um arquivo XML com a descrição da especificação em linguagem MIR e montar uma estrutura de objetos correspondentes.

   (b) Montando a estrutura de objetos que descreve a especificação diretamente por meio da instanciação apropriada das classes apresentadas no Capítulo 5.

3. Invocar a otimização da especificação por meio do método da classe de fachada destinado a este fim, passando a representação em memória obtida no passo anterior. Este método consulta a lista de otimizações, carrega-as em memória e as executa em cascata sobre a estrutura a ser otimizada, retornando como resultado a especificação otimizada. Este passo é opcional caso não se esteja interessado nas otimizações, mas apenas na geração de código.

4. De posse da especificação otimizada, instancia-se o visitor para geração de código, apresentado no Capítulo 5, e então dispara-se este visitor sobre a estrutura que representa a especificação otimizada. Este visitor gera o código C++ com a semântica equivalente à especificação, que pode então ser compilada em código executável.

A seguir são apresentados detalhes do projeto e implementação do kℓar. O capítulo finaliza mostrando um exemplo de aplicação que utiliza o kℓar. Este exemplo mostra o código necessário para os passos acima.

6.1 Estratégias e Decisões de Projeto

Uma das contribuições do kℓar consiste em fornecer um ambiente de otimizações para a representação intermediária de especificações ASM em linguagem MIR, onde as otimizações sejam facilmente adicionadas ou removidas. Para alcançar tal modularidade, faz-se necessário o estabelecimento de um padrão a ser seguido por aqueles que venham a implementar otimizações a serem adicionadas ao arcabouço. O objetivo desta seção é apresentar as decisões de projeto relevantes e o padrão a ser seguido propriamente. Em princípio, foram consideradas duas abordagens possíveis para a modularidade das otimizações no kℓar: modularidade no nível do código e modularidade no nível da aplicação.

Modularidade no nível do código: neste caso, novas otimizações devem ser adicionadas ao código-fonte, e todo o projeto deve ser recompilado. A padronização é feita por meio de padrões de projeto. Esta abordagem possui a vantagem de não envolver dificuldades ou padronizações junto a compiladores utilizados ou diretivas de compilação.

Modularidade no nível da aplicação: neste caso, o programa pode ser distribuído pré-compilado, preservando o código-fonte, e novas otimizações são *bibliotecas dinâmicas*. A padronização é feita por meio de padrões de projeto, compilador utilizado, diretivas de compilação e registro de configurações. Esta abordagem apresenta a vantagem de que o

núcleo das otimizações será sempre o mesmo, pois os desenvolvedores de novas otimizações não poderão modificá-lo, e com isso garante-se que uma otimização desenvolvida seja sempre compatível com qualquer aplicativo que faça uso do kℓar. Se a primeira abordagem fosse utilizada, alguém poderia mudar o núcleo das otimizações. Este ponto ficará mais claro com a apresentação do diagrama de classes, descrito inicialmente na Seção 6.1.1 e abordado em detalhes na Seção 6.3.

Considerando os fatos expostos, optou-se pela segunda abordagem. O kℓar foi desenvolvido em C++ padrão [Str00, EA01a, EA01b]. Ele possui cerca de 150 classes, que correspondem a quase 30 mil linhas de código. Durante o desenvolvimento, procurou-se observar também algumas estratégias de boa programação apresentadas por Meyers [Mey98, Mey96]. O compilador adotado foi o gcc, versão 3.0, que pode ser obtido sob a licença GNU no website do projeto [GCC]. Além disso, este compilador está disponível nas principais distribuições Linux existentes atualmente. A Seção 6.2 explica como desenvolver, compilar e configurar novos módulos de otimização para o kℓar.

6.1.1 Diagrama de Classes

O diagrama de classes do kℓar é apresentado na Figura 6.2. Dentre as classes apresentadas, convém destacar a classe KlarLibrary. É por meio de instâncias desta classe que o kℓar deve ser acessado. Outra classe importante é a classe Module, que representa a estrutura de um módulo em linguagem MIR, apresentada no Capítulo 4.

De modo a facilitar a leitura, foi adotada uma convenção nos diagramas de classe neste capítulo. Esta convenção consiste em anotar em cada classe os papéis que esta desempenha em algum padrão de projeto. Esta anotação possui a forma de um esteriótipo[1] segundo o formato

«dp:nome_do_padrao:papel_no_padrao»

Por exemplo, «dp:AbstractFactory:ConcreteFactory» indica que a classe que possui este esteriótipo representa o papel de uma fábrica concreta em um padrão de fábrica abstrata. Esta convenção se faz particularmente útil na Seção 6.3, onde o diagrama de classes é detalhado por meio da identificação dos padrões de projeto existentes no kℓar.

6.1.2 Documentação

Conforme destacado por Johnson [Joh97a, Joh97b], um arcabouço deve possuir uma documentação detalhada. Por este motivo, optou-se por gerar a documentação do kℓar a partir de seu código-fonte e de comentários especiais adicionados a este com esta intenção. O uso de um gerador automático de documentação ajuda a manter a documentação atualizada e completa. A ferramenta escolhida para esta tarefa foi o Doxygen [Dox], que é gratuito (sendo distribuído sob a licença GNU) e produz uma documentação de qualidade.

A Seção 6.2 descreve os dois tipos de otimização existentes no kℓar e como criar novas otimizações de cada um deste tipos, configurando-as apropriadamente. A Seção 6.4 apresenta como o kℓar pode ser utilizado por aplicações escritas em C++ por meio da elaboração de um protótipo que ilustra os aspectos envolvidos.

[1] Segundo Pilone [Pil03], *esteriótipos* são uma forma de se estender a UML por meio da definição de termos simples e de seu uso para o esclarecimento de elementos da UML e sua participação em um sistema.

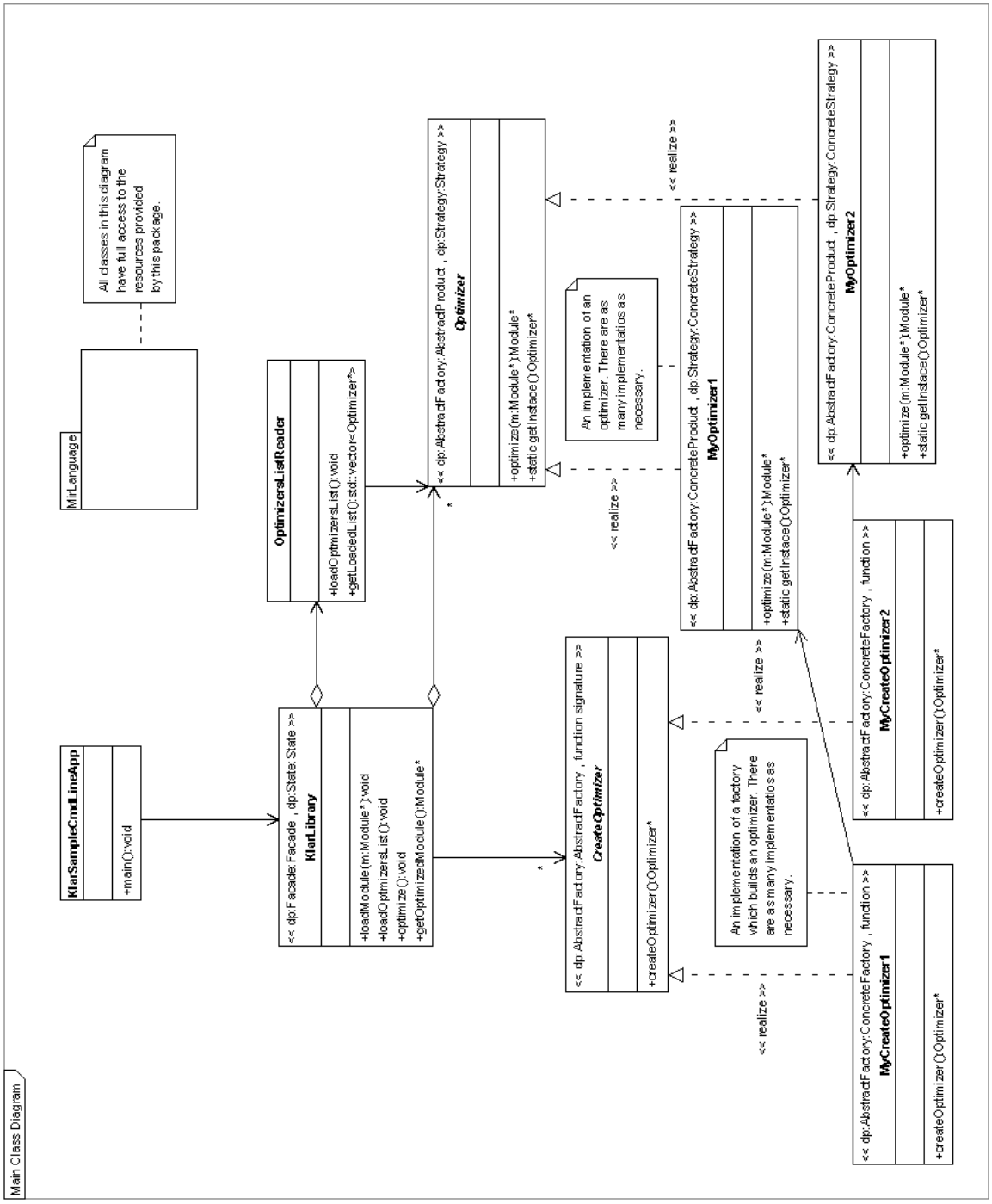

Figura 6.2: Diagrama de classes: visão geral da arquitetura do kℓar.

6.2 Classificação das Otimizações

De acordo com sua natureza e seu mecanismo de atuação, as otimizações são classificadas no contexto do kℓar em duas categorias: otimizações *intrínsecas* e otimizações *extrínsecas*, detalhadas a seguir.

6.2.1 Otimizações Intrínsecas

O termo otimizações *intrínsecas* diz repeito àquelas otimizações que alteram a forma como uma especificação ASM na linguagem MIR é traduzida para código C++ equivalente. Estas otimizações não atuam sobre especificações modificando-as, de modo a se obter uma outra especificação *equivalente* que seja mais eficiente ao mesmo tempo em que a sua semântica é preservada. O mecanismo destas otimizações se dá por meio da alteração de como construções específicas da linguagem MIR são mapeadas em elementos C++.

Conforme detalhado no Capítulo 5, funções dinâmicas em uma especificação ASM em linguagem MIR dão origem a tabelas *hash* que as representam em código C++. Mais exatamente, é utilizada a classe std::map, pertencente à *Standard Template Library* de C++, que consiste em uma implementação eficiente de uma tabela *hash*. Suponha que em um determinado contexto uma outra estrutura de dados seja mais eficiente para representar funções dinâmicas, ou que ainda uma nova implementação de uma tabela *hash* que ofereça vantagens em relação à versão padrão se torne disponível. Seria necessário alterar a forma como uma função dinâmica é convertida em código C++. Mais do que isto, acessos a pontos de funções dinâmicas poderiam ter de ser feitos de forma diferente no código C++, tanto como *right-values* quanto como *left-values*. Esta alteração visa a obtenção de código mais eficiente, mas ao mesmo tempo especificações ASM em linguagem MIR não percebem esta mudança. Especificações não são alteradas, apenas a forma como estas são convertidas em código C++ equivalente. Por isto, otimizações deste tipo são chamadas de otimizações intrínsecas.

A forma como as classes que representam módulos da linguagem MIR foram projetadas e o uso do padrão Visitor para a geração de código tornam esta tarefa consideravelmente mais simples. Para que tais otimizações sejam implementadas, deve-se herdar da classe CodeGenerator que, conforme apresentado no Capítulo 5, é a classe utilizada para a geração de código em C++ para módulos. Em seguida, na classe herdeira, deve-se sobrescrever os métodos responsáveis pela geração de código das estruturas que devem ter o seu mapeamento alterado. No exemplo acima, estes métodos seriam:

- visitDynamicFunction(DynamicFunction *df), responsável pela declaração da estrutura que representa uma função dinâmica;
- visitUpdate(Update *u) e visitImmediateUpdate(ImmediateUpdate *u), que utilizam funções dinâmicas como *left-values*;
- e finalmente visitDynamicFunctionCall(DynamicFunctionCall *fc), que utiliza funções dinâmicas como *right-values*.

Todos os outros métodos são aproveitados devido às características da herança em linguagens orientadas por objetos. Alterações em cascata podem ser obtidas por meio da derivação de classe após classe.

6.2.2 Otimizações Extrínsecas

Otimizações *extrínsecas* atuam por meio da *transformação* de uma dada especificação ASM em linguagem MIR. O resultado desta transformação deve ser uma outra especificação, semanticamente equivalente àquela originalmente dada, que porém seja sob algum aspecto mais eficiente.

Exemplos de tais otimizações são o escalonamento de instruções de modo a maximizar o número de atualizações imediatas [OBB04b] e a detecção de desvios mais eficientes [TB00], apresentadas respectivamente nas Seções 2.5.1 e 2.5.2.

Para a implementação de tais otimizações, faz-se necessária a observação de um protocolo em particular. Mais exatamente, os passos para a criação de uma otimização extrínseca no kℓar são:

1. Implementar a classe que encapsulará a otimização. Esta classe pode ter qualquer nome, contanto que ela herde da classe klar::Optimizer. Isto a obrigará a implementar o método puramente virtual optimize(Module*), que é chamado pelo arcabouço no momento da otimização, passando como parâmentro a representação em memória do módulo a ser otimizado. Este método deve retornar o módulo otimizado, e portanto ele é o ponto de partida para a implementação da otimização. Inclusive, toda a otimização pode ser implementada nele, embora métodos auxiliares ajudem a manter o código mais modular.

2. Criar a fábrica que fornecerá o otimizador da forma apropriada, conforme apresentado na Seção 6.3.2. Neste caso, a fábrica não é uma classe, mas sim uma função com nome e assinatura Optimizer *CreateOptimizer(). As Figuras 6.3 e 6.4 fornecem um modelo inicial para os Passos 1 e 2.

3. Compilar a classe com as opções de compilação adequadas. Esta compilação deve resultar em uma biblioteca de vínculo dinâmico, expondo a função Optimizer *CreateOptimizer(), que será utilizada para se criar instâncias do otimizador. Para o modelo das Figuras 6.3 e 6.4, a compilação é dada por:

```
g++ -c -fPIC CommonSubExprOtm.cc
```

```
g++ -o CommonSubExprOtm.so -shared -fPIC CommonSubExprOtm.o
```

4. Adicionar uma entrada ao arquivo de configuração `optimizers.cfg`, que contém a lista de otimizações a serem aplicadas e informações sobre cada otimização, tais como a ordem de aplicação e o local onde elas podem ser encontradas. A sintaxe deste arquivo é XML-*like*, e sua sintaxe é apresentada na Figura 6.5.

Quando o processo de otimização é iniciado, as otimizações descritas no arquivo de configuração de otimizações são carregadas e a primeira otimização é aplicada ao módulo carregado. Em seguida, a segunda otimização é aplicada ao resultado da primeira otimização, e assim sucessivamente, até que todas as otimizações sejam aplicadas. O resultado da última otimização é o módulo a ser retornado do processo de otimização.

```
#ifndef COMMONSUBEXPROTM_H
#define COMMONSUBEXPROTM_H
class CommonSubExprOtm : public Optimizer {
  private:
    // Default constructor
    CommonSubExprOtm();
    // The optimizer
    static Optimizer* optimizer;
  public:
    // The class method which provides an Optimizer
    static Optimizer* getInstance();
    // Default destructor
    virtual ~CommonSubExprOtm();
    // Method which performs the common subexpressions optimization.
    virtual mir::Module* optimize(mir::Module* input);
};
// The factory for the optimzer
extern "C" {
  Optimizer *CreateOptimizer(); }
#else
class CommonSubExprOtm : public Optimizer;
Optimizer *CreateOptimizer();
#endif
```

Figura 6.3: Exemplo de implementação do padrão Singleton para uma otimização em particular.

```
#include "CommonSubExprOtm.h"
//------------------------------------------------------------------------------
CommonSubExprOtm::optimizer = 0;
//------------------------------------------------------------------------------
CommonSubExprOtm::CommonSubExprOtm() { }
//------------------------------------------------------------------------------
Optimizer* CommonSubExprOtm::getInstance() {
  if (optimizer == 0) optimizer = new CommonSubExprOtm();
  return optimizer; }
//------------------------------------------------------------------------------
CommonSubExprOtm::~CommonSubExprOtm() { }
//------------------------------------------------------------------------------
mir::Module* CommonSubExprOtm::optimize(mir::Module* input) {
  /* Does the optimization */
  ... }
//------------------------------------------------------------------------------
extern "C" {
  Optimizer *CreateOptimizer() { return new CommonSubExprOtm(); }
}
//------------------------------------------------------------------------------
```

Figura 6.4: Exemplo de implementação do padrão Singleton para uma otimização em particular.

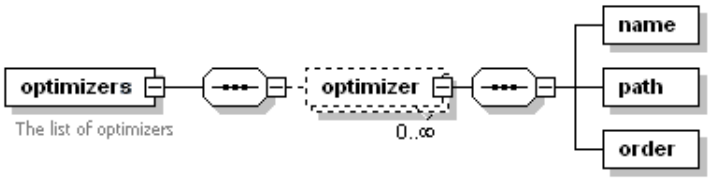

Figura 6.5: Sintaxe do arquivo de configurações `optimizers.cfg`.

6.3 Padrões de Projeto no kℓar

Conforme apresentado na Seção 3.3, padrões de projeto são úteis também no entendimento de um projeto de software, pois se constituem em soluções de conhecimento comum para a solução de problemas recorrentes. Uma vez identificada a presença de um padrão de projeto, o analista experiente já reconhece as implicações desta presença no funcionamento de um programa. Esta seção tem por objetivo explicitar os padrões de projeto presentes no kℓar, de modo a facilitar o entendimento de seu funcionamento.

6.3.1 Padrão *Estado* e a classe KlarLibrary

O padrão *Estado* se faz presente no kℓar por meio das seguintes classes:

- KlarLibrary - É o contexto do padrão.
- KlarLibrary_State - É o estado do padrão, e é implementada como uma classe abstrata em C++, de tal forma que todos os seus métodos são puramente virtuais.
- KlarLibrary_Idle - Representa o estado anterior a qualquer uso de KlarLibrary. Uma chamada ao método loadModule carrega o módulo a ser otimizado e faz com que um objeto da classe KlarLibrary_ModuleLoaded seja o estado atual. Outros métodos desta classe, se invocados, geram uma exceção.
- KlarLibrary_ModuleLoaded - Representa o estado posterior ao carregamento do módulo a ser otimizado. Neste estado, a lista de otimizações pode ser carregada por meio do método loadOptimizersList, de tal forma que o estado atual após a execução deste método é um objeto da classe KlarLibrary_OptimizersLoaded. Se o método loadModule for invocado, o estado atual continua sendo um objeto da classe KlarLibrary_ModuleLoaded, agora com um novo módulo. Outros métodos desta classe, se invocados, geram uma exceção.
- KlarLibrary_OptimizersLoaded - Representa o estado posterior ao carregamento da lista de otimizações. Neste estado, o módulo carregado pode ser otimizado por meio do método optimize, de tal forma que o estado atual após a execução deste método é um objeto da classe KlarLibrary_Optimized. Se o método loadOptimizersList for invocado, o estado atual continua sendo um objeto da classe KlarLibrary_OptimizersLoaded, agora com uma nova lista de otimizações. Se o método loadModule for invocado, o estado atual passa a ser um objeto da classe KlarLibrary_ModuleLoaded, agora com um novo módulo. O método getOptimizedModule, se invocado, gera uma exceção.

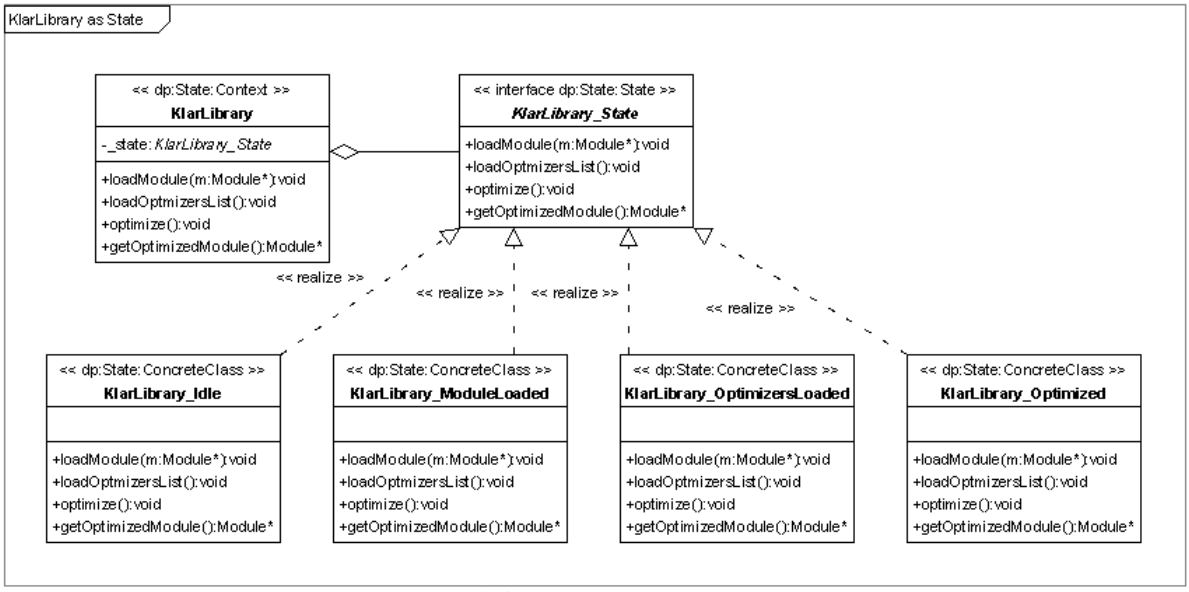

Figura 6.6: Diagrama de classes do padrão *Estado* manifesto no kℓar.

- KlarLibrary_Optimized - Representa o estado final da otimização de um módulo. A operação natural neste estado é a obtenção do módulo otimizado por meio do método getOptimizedModule. A invocação de outros métodos provoca o retrocesso ao estado apropriado.

O relacionamento entre estas classes é apresentado no diagrama de classes da Figura 6.6.

6.3.2 Padrão *Fábrica Abstrata* e a criação de classes derivadas de Optimizer

O padrão *Fábrica Abstrata* é utilizado no kℓar para permitir a modularidade desejada nas otimizações, de tal forma que estas sejam intercambiáveis e possam ser aplicadas em seqüência, em qualquer ordem. Além disso, as otimizações são carregadas dinamicamente, ou seja, quando o kℓar é compilado, não se sabe ao certo que otimizações serão utilizadas, fazendo-se necessária a utilização do padrão Fábrica Abstrata. Esta abordagem envolve outros detalhes que dizem respeito à compilação das otimizações, e estes detalhes são discutidos na Seção 6.2. Neste ponto, mantém-se a discussão apenas no aspecto do padrão Fábrica Abstrata.

- KlarLibrary - É o cliente das fábricas.
- CreateOptimizer - Seu papel é ser a fábrica abstrata de otimizadores. No kℓar, é implementado não como uma classe propriamente dita, mas sim como um typedef de uma assinatura de função, por motivos detalhados na Seção 6.2.
- MyCreateOptimizer1, MyCreateOptimizer2, etc. - Representam funções com a assinatura definida por CreateOptimizer que criam as diversas otimizações apropriadas.
- Optimizer - Classe abstrata que determina a interface de uma otimização.

- MyOptimizer1, MyOptimizer2, etc. - Estas classes representam as diversas otimizações que podem ser implementadas para uso no kℓar.

A Figura 6.7 apresenta as classes do padrão Fábrica Abstrata e as relações entre elas.

6.3.3 Padrão *Fachada* e a classe KlarLibrary

O kℓar foi concebido para ser acessado apenas por meio da classe KlarLibrary, de modo que os objetos desta classe constituem uma Fachada, e todas as demais classes implementam as funcionalidades do sistema.

O diagrama de classes apresentado na Figura 6.8 ilustra a presença do padrão no kℓar. A classe KlarSampleCmdLineApp é uma aplicação simples, de linha de comando, que utiliza o kℓar por meio de sua Fachada, a classe KlarLibrary.

Muito embora em geral classes de fachada sejam também únicas (padrão Singleton, detalhado na Seção 3.3.4), optou-se por permitir que vários objetos desta classe sejam instanciados. Desta forma, um ambiente integrado de desenvolvimento que permita que vários projetos sejam abertos simultaneamente pode atribuir um objeto da classe KlarLibrary para cada projeto, ou ainda para cada módulo presente em um projeto. Cabe ressaltar que as classes da linguagem MIR não se inclúem neste padrão, podendo ser utilizadas diretamente.

6.3.4 Padrão *Singleton* e as classes concretas de Optimizer

No kℓar, assume-se que nenhuma informação é armazenada nas classes concretas derivadas de Optimizer, ou seja, estas classes não fornecem a noção de um estado a ser preservado ou alterado entre sucessivas chamadas de seus métodos. Mais exatamente, estas classes possuem um caráter puramente algoritmico, e, dada um mesmo módulo a ser passado como parâmetro ao método Optimize, o resultado deste deve ser sempre o mesmo, não importa em que momento o método foi invocado. (Vide Seção 6.3.5.) Isto justifica a adoção do padrão Singleton para as classes concretas derivadas de Optimizer. O diagrama de classes apresentado na Figura 6.9 ilustra a presença do padrão no kℓar.

Note que toda classe derivada de Optimizer deve possuir um método de classe que implemente o padrão corretamente, e que os construtores devem ser todos privados, impedindo a instanciação direta de objetos da classe. Um pequeno código em C++ é apresentado como exemplo nas Figuras 6.3 e 6.4.

6.3.5 Padrão *Estratégia* e as classes concretas de Optimizer

Este padrão aparece no kℓar nas classes apresentadas na Figura 6.10, onde a classe KlarLibrary faz o papel de Contexto, a classe Optimizer faz o papel de Estratégia e as classes concretas derivadas de Optimizer são as implementações dos algoritmos, que neste caso são algoritmos de otimização de módulos em linguagem MIR. O módulo é a estrutura de dados a ser passada para estes algoritmos.

Ressalta-se que geralmente o padrão Estratégia é usado para *comutar* algoritmos de forma mutuamente exclusiva, mas no kℓar este padrão é usado para *aplicações sucessivas* de diferentes algoritmos sobre uma mesma estrutura de dados.

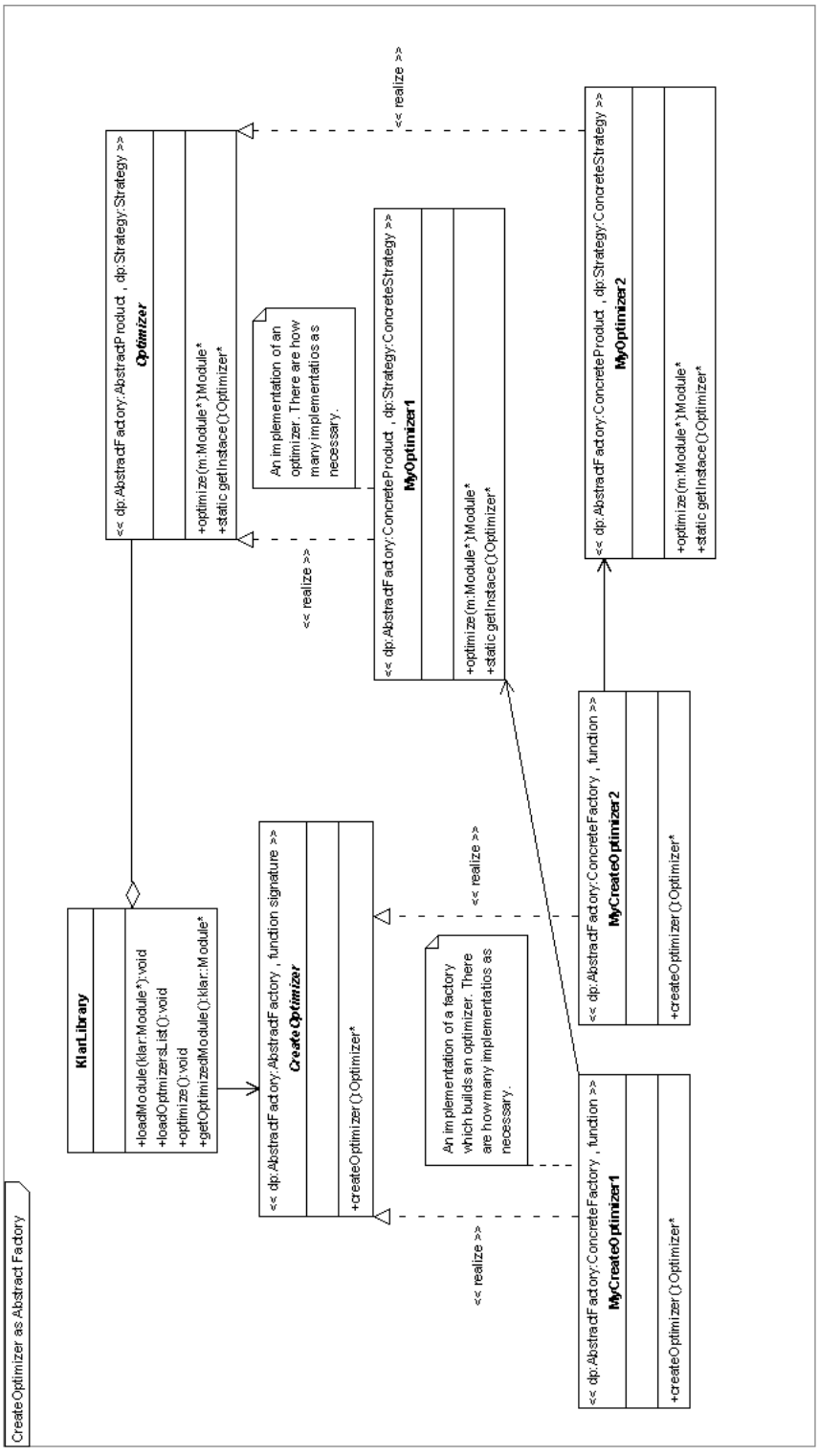

Figura 6.7: Diagrama de classes do padrão *Fábrica Abstrata* manifesto no klar.

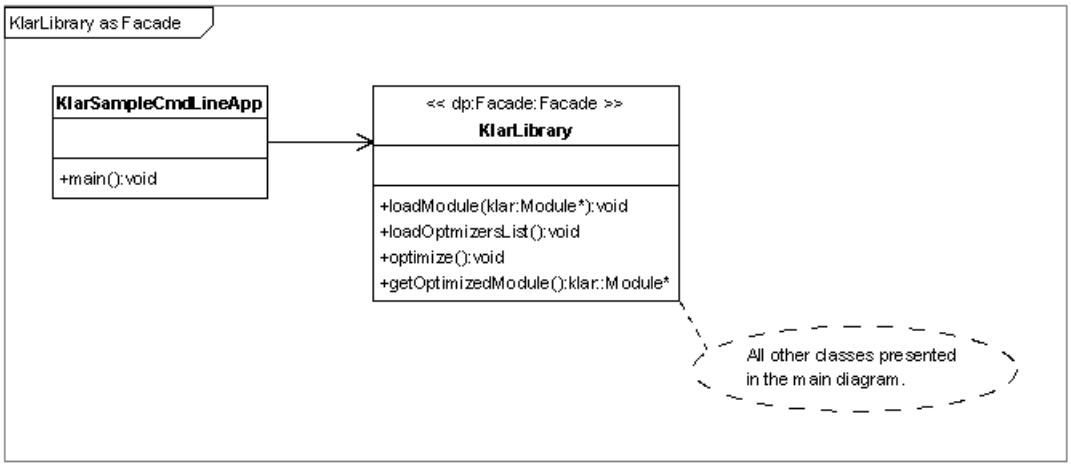

Figura 6.8: Diagrama de classes do padrão *Fachada* manifesto no kℓar.

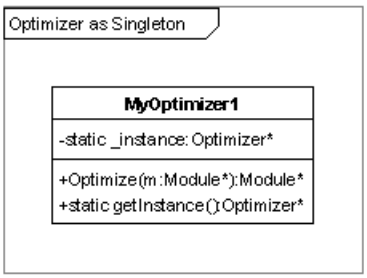

Figura 6.9: Diagrama de classes do padrão *Singleton* manifesto no kℓar.

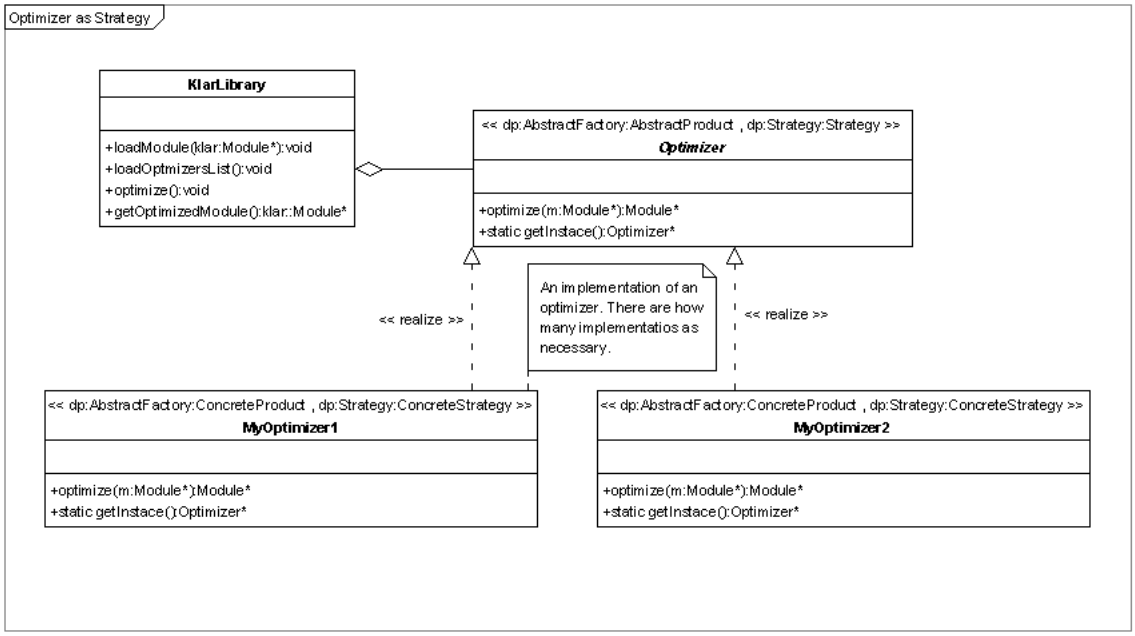

Figura 6.10: Diagrama de classes do padrão *Estratégia* manifesto no kℓar.

6.3.6 Padrões *Composite* e *Visitor* e as classes da linguagem MIR

Os padrões de projeto *Composite* e *Visitor* foram apresentados nas Seções 3.3.6 e 3.3.7, respectivamente, e são largamente utilizados na implementação das classes que representam elementos da linguagem MIR, como detalhado no Capítulo 5. Mais exatamente, o padrão Composite é utilizado na definição de hierarquias de objetos para a representação de três elementos, a saber: tipos, expressões e regras. Isto permite que tanto objetos individuais destes elementos quanto composições de objetos sejam tratados uniformemente. Já o padrão Visitor oferece a chance de percorrer as estruturas que compõem os módulos de uma forma estruturada, realizando alguma operação, e concentrando o código relativo a esta operação em um único lugar. Esta abordagem facilita a extensão do arcabouço por meio da adição de novas funcionalidades.

6.4 Uso do kℓar

O objetivo desta seção é apresentar o uso do kℓar como uma biblioteca por meio da introdução de uma pequena aplicação de linha de comando. Embora simples, a elaboração desta aplicação aborda os principais aspectos necessários para a utilização do kℓar. O programa resultante funciona da seguinte forma:

1. O programa recebe como parâmetro na linha de comando o nome de um arquivo de extensão `mas` ou `mod` contendo a definição de um arquivo MAS ou um módulo em linguagem MIR, respectivamente.

2. De acordo com a extensão do arquivo, o carregador apropriado é utilizado, obtendo-se uma estrutura em memória que representa o arquivo lido.

3. Se a estrutura é um módulo, este é otimizado e salvo como um novo arquivo `mod`, de forma que as transformações realizadas posaam ser avaliadas.

4. Código C++ é gerado para a estrutura carregada.

O código para o aplicativo proposto é apresentado na Figura 6.11. As linhas 1 e 2 fazem a inclusão de bibliotecas auxiliares. O cabeçalho de fachada do kℓar é incluído na linha 5. É este cabeçalho que define a classe Klar_Library, apresentada anteriormente. Além disso, os cabeçalhos das classes que representam um módulo são incluídos na linha 8.

Após as inclusões necessárias, pode-se iniciar o código que realmente constitui a aplicação. Conforme dito anteriormente, o primeiro passo da utilização do kℓar consiste na obtenção de uma instância da classe de fachada, o que é feito na linha 13. As linhas 16 a 20 consistem em código burocrático que determinam o nome e a extensão do arquivo. De acordo com a extensão, o programa toma a decisão de qual carregador utilizar. Se a extensão é `.mas`, então trata-se de um arquivo de disparador de agentes, que não é otimizado, mas apenas carregado e então é gerado código correspondente. Este processo acontece nas linhas 23 a 31. Caso a extensão seja `.mod`, então trata-se de um arquivo de módulo, que é carregado, otimizado e então gera-se código. Este processo é feito nas linhas 35 a 50. A linha 36 instancia o carregador apropriado, e o carregamento é feito na linha 39. As linhas 42 a 45 carregam os otimizadores e otimizam o módulo. A linha 48 gera a representação XML do módulo otimizado, enquanto a linha 49 gera o código executável. Para compilar a aplicação do Exemplo, pode-se utilizar o seguinte `makefile`

```
g++ KlarCmdLineApp.cc KlarLibrary.o -o klar -ldl
```

Note que o kℓar é disponibilizado como uma biblioteca de vínculo estático, e por isso deve estar acessível no momento da compilação.

6.5 Conclusão

Neste capítulo foi apresentada a arquitetura do kℓar. Esta apresentação incluiu a descrição de sua utilização, a organização das classes constituintes e a identificação dos padrões de projetos presentes no kℓar.

O kℓar oferece como principais características o ambiente adequado para a inserção de otimizações para ASM e também a infra-estrutura necessária para a geração de código C++ correspondente às especificações ASM válidas na linguagem MIR.

Conforme apresentado no aplicativo desenvolvido como exemplo, é relativamente fácil utilizar o kℓar. Além de estar disponível como uma classe de fachada simples, o kℓar oferece um carregador para montar em memória a estrutura que representa o módulo a ser otimizado a partir de sua descrição XML, poupando o usuário de ter que lidar com análise e interpretação de arquivos. Uma vez configuradas por meio de um arquivo especial, as otimizações podem ser aplicadas via a chamada de um simples método, de forma que o usuário do kℓar não precisa se preocupar com o carregamento dinâmico de classes.

Adicionalmente, a geração de código C++ executável correspondente à especificação otimizada pode ser feita simplesmente por meio da chamada de um método do visitor de geração de código. Este protocolo e a construção de um aplicativo de exemplo plenamente funcional com apenas poucas linhas de código atestam a simplicidade do uso do kℓar.

```
01: #include <iostream>
02: #include <string>
03:
04: // The header of the Klar facade
05: #include "KlarLibrary.h"
06:
07: // The header of the MIR language
08: #include "Mir.h"
09:
10: int main(int argc,char *argv[])
11: {
12:   // The klar facade
13:   klar::KlarLibrary* klar = new klar::KlarLibrary();
14:
15:   // The file name
16:   std::string filename(argv[1]);
17:
18:   if (argc > 0)
19:   {
20:     std::string ext = ...
21:     if (ext == "mas")
22:     {
23:       // The loader of MAS files
24:       mir::MASLoader* masloader = new mir::MASLoader(filename);
25:
26:       // Loads the MAS specification
27:       mir::MASSpecification* mas = masloader->doLoad();
28:
29:       // Generates C++ code
30:       mas->visit(new mir::CodeGenerator());
31:       return 0;
32:     }
33:     else if (ext == "mod")
34:     {
35:       // The loader of module files
36:       mir::ModuleLoader* mirloader = new mir::ModuleLoader(filename);
37:
38:       // Loads the MIR specification
39:       mir::Module* module = mirloader->doLoad();
40:
41:       // Optimizes the MIR specification
42:       klar->loadModule(module);
43:       klar->loadOptimizersList();
44:       klar->optimize();
45:       module = klar->getOptimizedModule();
46:
47:       // Generates C++ code and saves the modified module
48:       module->visit(new mir::CodeGenerator());
49:       module->visit(new mir::Serialize());
50:       return 0;
51:     }
52:   }
53:   cout << "Please enter with a file name with a valid extension.\n";
54:   return 1;
55: }
```

Figura 6.11: Exemplo de aplicação que utiliza o klar como uma biblioteca estática.

As otimizações foram classificadas segundo seus tipos, e o mecanismo para a implementação destas foi apresentado. Nas otimizações intrínsecas, que alteram como construções da linguagem são mapeadas em código C++ equivalente, a herança é o mecanismo que permite que melhorias na geração de código seja obtida atuando-se apenas nos pontos de interesse. Nas otimizações extrínsecas, os padrões de projeto Fábrica Abstrata e Estratégia tornam possível o carregamento dinâmico de otimizações com baixo acoplamento, de forma que otimizações diferentes podem ser desenvolvidas em separado e utilizadas em cascata. O próximo capítulo apresenta a implementação de uma destas otimizações e os resultados obtidos.

Capítulo 7

Validação do kℓar e Avaliação dos Resultados

7.1 kℓar Como Arcabouço para Otimizações ASM

Com vistas à validação do kℓar como um arcabouço para a otimização de especificações ASM, foi implementado o algoritmo proposto por Oliveira *et alii* [OBB04b], que trata do escalonamento de regras de modo a maximizar as atualizações realizadas diretamente. Esta otimização, apresentada suscintamente na Seção 2.5.1, ilustra o processo de criação, compilação e acoplamento de otimizações no contexto do kℓar. Esta seção documenta esta implementação. Como um dos objetivos do kℓar é fazer com que a implementação de tais otimizações seja facilitada pela infra-estrutura oferecida pelo arcabouço, no exemplo aqui apresentado, fez-se uso das facilidades o tanto quanto possível.

Segundo Oliveira *et alii* [OBB04b], a otimização para o escalonamento de regras pode ser modelada como um problema de grafos, conforme descrito a seguir. Seja R uma regra de transição de uma especificação ASM. Define-se $G_R = (V_R, E_R)$ como o *grafo de escalonamento ASM* associado à regra R. G_R é um grafo dirigido formado da seguinte maneira:

- V_R é o conjunto de vértices do grafo. Cada sub-regra constituinte da regra R corresponde a um vértice $v_r \in V_R$.
- Existe uma aresta $(v_i, v_j) \in E_R$ se e somente a sub-regra v_i faz a avaliação de uma função dinâmica que é atualizada na sub-regra v_j.

As arestas de G_R são rotuladas com um mesmo peso, enquanto que cada vértice $v_r \in V_R$ é associado um peso que indica o potencial benefício obtido pela remoção deste vértice durante o escalonamento. Este peso é chamado de *relação de benefício*.

O algoritmo consiste de três passos [OBB04b]:

Passo 1: Montagem da dependência entre as funções dinâmicas.

Passo 2: Construção do grafo de conflitos a partir das dependências detectadas.

Passo 3: Escalonamento das instruções.

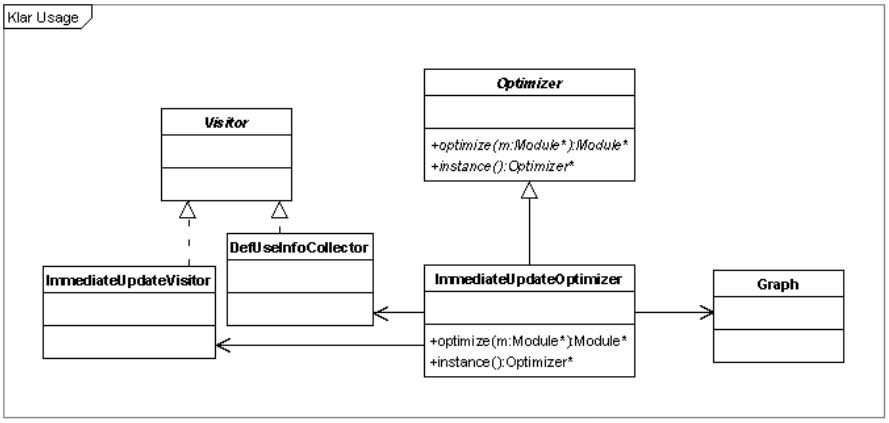

Figura 7.1: Diagrama de classes de uma otimização implementada no kℓar.

A seguir, apresenta-se em linhas gerais os detalhes da otimização implementada e mostra-se como esta implementação fez uso das facilidades oferecidas pelo kℓar.

A Figura 7.1 apresenta o diagrama UML da implementação. A classe ImmediateUpdateOptimizer representa a otimização propriamente dita, que, como definido pelo arcabouço, herda de Optimizer. Outras estruturas auxiliares são também utilizadas. Este diagrama é detalhado a seguir.

7.1.1 O Visitor DefUseInfoCollector

Um recurso recorrente em problemas de otimização de código imperativo consiste em determinar as cadeias de definição e uso, ou *DU-chains* [Muc97]. Em programas ASM, devido ao paralelismo intrínseco de uma regra de transição, este conceito não é apropriado. Entretanto, um conceito análogo pode ser utilizado: o das funções dinâmicas que são lidas e atualizadas por cada sub-regra de uma regra de transição.

O visitor DefUseInfoCollector tem como função percorrer regras de transição e associar dois conjuntos a cada sub-regra: o conjunto das funções dinâmicas *consultadas* pela sub-regra e o conjunto das funções dinâmicas *atualizadas* pela sub-regra. Este visitor faz parte da implementação do kℓar, e está disponível para uso por parte dos desenvolvedores de otimizações.

7.1.2 A Classe Graph

A classe Graph é uma classe auxiliar no processo de otimização implementado. Como o algoritmo proposto é baseado na construção de um grafo de conflitos, a utilização de uma classe para a representação de grafos se faz necessária. Desta forma, foi implementada a classe Graph, que fornece a funcionalidade necessária dentro do contexto da otimização. Internamente, o grafo é representado como uma matriz de adjacências, pois esta representação se mostrou mais adequada para a implementação do método que calcula os caminhos mais curtos, o qual é necessário no algoritmo de otimização.

O uso da classe Graph mostra como é possível estruturar a otimização como uma hierarquia de classes, de modo que a otimização não precisa ficar totalmente encapsulada apenas na classe

```
01:  mir::Module* ImmediateUpdateOptimizer::optimize(mir::Module* m)
02:  {
03:    // initialization
04:     du = new DefUseInfoCollector();
05:     iuv = new ImmediateUpdateVisitor(this);
06:     graph.clear();
07:     ruleListStack.clear();
08:     ruleLevel = -1;
09:     blockLevel = 0;
10:     vertexDegree = 0;
11:
12:     // Gets the information about definition and use (DU) of dynamic functions
13:     du->visitModule(m);
14:
15:     // get the optimized version of the rule
16:     mir::Rule *r = m->getTransitionRule();
17:     Rule* r_otm = generateCode(r);
18:     m->setTransitionRule(r_otm);
19:
20:     return m;
21:   }
```

Figura 7.2: O método optimize da otimização implementada.

derivada de Optimizer.

7.1.3 A Classe ImmediateUpdateOptimizer

A classe ImmediateUpdateOptimizer é o núcleo da otimização implementada. Conforme especificado pelo protocolo estabelecido pelo arcabouço, esta classe deriva de Optimizer e implementa os métodos virtuais optimize e instance, que têm por função implementar a otimização em si e retornar a instância única do otimizador, respectivamente.

Método optimize

A Figura 7.2 apresenta a implementação do método optimize.

As linhas 4 a 10 fazem a inicialização das estruturas de dados pertinentes ao processo de otimização conforme implementado. A linha 4 instancia o visitor responsável pela coleta de informações de uso e definição de funções dinâmicas. Este visitor é fornecido pelo kℓar, de modo que outras otimizações podem fazer uso deste instrumento de coleta de informação quando necessário. Na linha 5, um outro visitor é instanciado, a saber, ImmediateUpdateVisitor. É este visitor que percorrerá a árvore que representa a regra de transição do módulo, montando o grafo e removendo-se os vértices. A seguir, outras estruturas auxiliares são inicializadas.

Inicialmente, o método optimize coleta informações do uso de funções dinâmicas no módulo a ser otimizado. Isto é feito com uma simples chamada de método na linha 13, utilizando-se o visitor DefUseInfoCollector instanciado. O resultado desta operação são duas tabelas *hash*: uma associa cada sub-regra R da regra de transição a uma lista dos identificadores das funções dinâmicas consultadas pela regra R, enquanto a outra faz a associação das mesmas sub-regras a uma lista dos identificadores das funções dinâmicas atualizadas.

Em seguida, na linha 17, é gerado a versão otimizada da regra de transição do módulo por meio a uma chamada ao método generateCode. No nome deste método, o *Code* não se refere ao código C++, mas sim à regra de transição otimizada ainda em linguagem MIR. Este método dispara o visitor ImmediateUpdateVisitor, que iterativamente monta o grafo de conflitos e escolhe os vértices para a geração de código otimizado. Se o vértice escolhido não é uma regra simples, esta regra é recursivamente otimizada. O grafo de conflitos é uma instância de Graph, e é construído com base na informação coletada por DefUseInfoCollector. Quando todos os vértices forem removidos, o processo de otimização chegou ao fim, e o novo módulo contendo o novo programa MIR otimizado é retornado.

7.1.4 Impacto da Otimização Implementada

Testes foram realizados com o objetivo de verificar o impacto da otimização implementada sobre o desempenho de programas em MIR. Para este teste, foram utilizados três dos programas apresentados no Apêndice B, a saber, SelSort, Fibonacci e Counting. Os programas do Apêndice B são descritos na Seção 7.2.1. A metodologia do teste foi a seguinte:

1. Os programas em MIR utilizados para teste foram compilados para C++ antes da otimização.

2. Os programas em C++ resultantes foram compilados com as otimizações de código ativadas.

3. Os programas binários resultantes foram executados com diversos parâmetros que determinavam o tamanho do problema a ser resolvido, por exemplo, o tamanho da base de dados a ser ordenada ou a quantidade de números de Fibonacci a serem calculados, e este processo foi repetido várias vezes. Os tempos de execução foram registrados, e foram calculadas as médias para um mesmo programa e parâmetros.

4. Em seguida, os mesmos programas em MIR foram otimizados.

5. Repetiu-se os passos 1 a 3, desta vez com os programas otimizados.

6. Foram comparados os tempos de execução para um mesmo programa e mesmos parâmetros, incluindo as mesmas massas de dados.

A Tabela 7.1 apresenta os resultados de um pequeno *benchmark* realizado com a otimização implementada. Este *benchmark* está longe de ser completo, mas deve-se ressaltar que o objetivo aqui não é avaliar a otimização e sua eficiência. O objetivo da validação aqui realizada é avaliar a *facilidade* de se acoplar otimizações ao arcabouço kℓar.

De forma geral, o resultado apresentado pela otimização apresentou uma melhora entre 5% a 15% no tempo de execução dos programas testados. Esta variação se dá de acordo com a densidade de atualizações de funções dinâmicas em comparação com outras construções da linguagem. Como a otimização proposta melhora o tempo de execução de atualizações, o seu efeito será tanto mais percebido quanto mais atualizações de funções dinâmicas existirem em comparação com outras construções na regra de transição.

SelSort

Tamanho	1000	2000	4000	8000	16000
Versão Não-Otimizada	0.975 s	3.891 s	15.248 s	1m03.717 s	4m22.785 s
Versão Otimizada	0.872 s	3.363 s	13.406 s	56.873 s	3m53.961 s
Ganho Percentual	10.56	13.56	12.07	10.74	10.96

Fibonacci

Tamanho	10e3	20e3	40e3	80e3	16e3
Versão Não-Otimizada	0.04 s	0.076 s	0.154 s	0.308 s	0.616 s
Versão Otimizada	0.038 s	0.071 s	0.143 s	0.292 s	0.585 s
Ganho Percentual	5	6.57	7.14	5.19	5.03

Counting

Tamanho	2e6	4e6	6e6	8e6	10e6
Versão Não-Otimizada	4.107	8.214	12.313	16.427	20.506
Versão Otimizada	3.457	6.912	10.353	13.798	17.257
Ganho Percentual	15.82	15.85	15.91	16.00	15.84

Tabela 7.1: Um pequeno *benchmark* para a avaliação da otimização implementada.

7.2 kℓar Como Estrutura para Implementação de Compiladores ASM

Para a validação da implementação de cada construção da linguagem MIR, foram feitos exemplos que testassem as mesmas. Na Seção 7.2.1 são descritos estes exemplos, cujos códigos-fonte são apresentados no Apêndice B.

7.2.1 Descrição dos Exemplos

Ordenação por Seleção (SelSort)

Este exemplo se constitui na geração de uma seqüência de números inteiros aleatórios que são em seguida ordenados pelo método de ordenação por seleção. São utilizados quatro módulos, a saber:

- SelSort.mod: implementa o método de ordenação em si.
- Data.mod: gera os dados a serem ordenados.
- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.
- SelSortContext.mas

Números de Fibonacci (Fibonacci)

Este exemplo gera os n primeiros números de Fibonacci.

- Fibonacci.mod: gera os números de Fibonacci.
- FibonacciContext.mas

Contagem (Counting)

Este exemplo simples consiste apenas na incrementação de uma função dinâmica. O objetivo era utilizar este exemplo no emphbenchmark.

- Counting.mod: realiza as atualizações sucessivas que incrementam uma função dinâmica.
- CountingContext.mas

Sorteio (Raffle)

O objetivo deste exemplo é testar construções da linguagem intermediária que trabalham com conjuntos e tuplas. Para isto, é criado um conjunto com 3-uplas contendo o nome e o sobrenome de um indivíduo e um indicador se ele está vivo ou não. Um sorteio é realizado, obtendo-se o nome de um indivíduo que satisfaça à condição de estar vivo, e são impressos em tela os nomes completos dos participantes, e então o vencedor é apresentado. Os módulos utilizados são:

- Raffle.mod: forma o conjunto de participantes do sorteio e realiza o sorteio em si, apresentando ao final as informações pertinentes.
- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.
- RaffleContext.mas

Funções Matemáticas (Math)

As marcas da linguagem referentes às operações matemáticas são exploradas neste exemplo. Para tal, é implementada uma função que calcula o seno de um ângulo em radianos por meio de uma aproximação finita da seguinte série matemática:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \tag{7.1}$$

Para a implementação desta função, outras funções também foram implementadas, tais como fatorial, potenciação e paridade. O módulo principal, Math.mod, faz uso destas funções calculando o seno de um conjunto de ângulos. A função seno é encapsulada no módulo Functions.mod, que também provê as outras funções implementadas. Os módulos que constituem este exemplo são:

- Math.mod: módulo principal que calcula o seno de um conjunto de ângulos, imprimindo o resultado na tela.
- Functions.mod: agrupa as diversas funções matemáticas implementadas para este exemplo.
- Output.mod: provê funções de saída de dados na tela.

- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.
- MathContext.mas

União de Tipos e Tipo Any (TypeUnion)

Este exemplo tem por objetivo testar os elementos da linguagem relativos à união de tipos e suas projeções. Uma lista de elementos é percorrida e um procedimento diferente é tomado de acordo com o tipo real do elemento em questão.

- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.
- TypeUnion.mod Constrói uma lista de valores de diversos tipos e então os seleciona segundo seus tipos em uma regra *with*.
- TypeUnionContext.mas

Funções Lógicas (TrueTables)

As operações booleanas constituem o alvo deste exemplo. Tabelas-verdade são construídas e impressas em tela para as funções booleanas da linguagem.

- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.
- TrueTables.mod Testa as operações booleanas, montando e imprimindo as tabelas-verdade de cada uma.
- TrueTablesContext.mas

Multi-Agentes sem Sincronismo (NoSyncronism)

Este é o primeiro exemplo que faz uso de multi-agentes. Nenhum mecanismo de sincronismo é avaliado neste ponto, pois o interesse aqui é apenas testar a criação e destruição de novos agentes. O agente principal atua como um observador que aciona um alarme toda a vez que uma porta é aberta. Acionar o alarme consiste em criar um agente que imprime em tela um alarme repetidamente. Quando a porta é fechada, o agente-alarme é destruído.

- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.
- NoSync.mod Módulo principal que interage com o usuário, recebendo comandos para abertura e fechamento de um porta hipotética.

- Alarm.mod Alarme associado à abertura da porta, que initerruptamente imprime uma mensagem avisando que a porta está aberta.
- NoSyncContext.mas

Jantar dos Filósofos (Philosophers)

Uma implementação do problema do jantar dos filósofos (apud [Tan99]) é realizada neste exemplo. Esta implementação faz uso dos mecanismos de sincronização entre agentes de modo a evitar *deadlocks* e condições de corrida.

- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.
- PhilosophersDinning.mod : O contexto do problema do jantar dos filósofos. Monta a mesa e então dispara os agentes dos filósofos.
- Philosopher.mod: O módulo de um filósofo.
- Table.mod: A mesa onde estão os garfos dos filósofos.
- PhilosophersDinningContext.mas

Produtores e Consumidores (ProdCons)

Este exemplo é uma implementação do problema dos produtores e consumidores [Tan99]. Nesta implementação, um agente é o produtor, que preenche um buffer circular com o seu produto, enquanto outro agente é o consumidor, que retira produtos do buffer. O sincronismo está presente de modo a garantir a ausência de *overflow* e *underflow*.

- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.
- ProdCons.mod: O buffer circular onde os números primos são empilhados e consumidos.
- Producer.mod: O produtor de números primos.
- Consumer.mod: O consumidor de números primos.
- ProdConsContext.mas

Tipo de dados *node* (Node)

Este exemplo testa o tipo *node* por meio da construção de uma árvore binária com dois tipos de nós. Esta árvore é então percorrida e impressa em in-ordem.

- Node.mod: a árvore é definida e percorrida.
- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.

Apontadores para Abstrações (Abstractions)

Este exemplo ilustra como apontadores para abstrações são definidos, armazenados e utilizados.

- Abstractions.mod: define, armazena e utiliza apontadores para abstrações.
- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.

Passagem de Parâmetros de Entrada e Saída (ByRef)

Este exemplo ilustra como parâmetros de entrada e saída são utilizados.

- ByRef.mod: define, invoca e verifica o resultado da chamada de ação com parâmetros de entrada e saída.
- Output.mod: provê funções de saída de dados na tela.
- StringManipulation.mod: conjunto de funções para manipulação de strings, tais como conversão de valores em strings e concatenação das mesmas.

7.2.2 Avaliação dos Exemplos

Os exemplos da Seção 7.2.1 foram propostos de modo a testar cada elemento da linguagem MIR. As Tabelas 7.2, 7.3 e 7.4 apresentam a relação entre os elementos da linguagem e os exemplos que as utilizam.

Nesta tabela, a numeração ao alto corresponde aos exemplos apresentados anteriormente, segundo esta legenda:

1. Ordenação por Seleção (SelSort)
2. Números de Fibonacci (Fibonacci)
3. Contagem (Counting)
4. Sorteio (Raffle)
5. Funções Matemáticas (Math)
6. União de Tipos e Tipo Any (TypeUnion)
7. Funções Lógicas (TrueTables)
8. Multi-Agentes sem Sincronismo (NoSyncronism)
9. Jantar dos Filósofos (Philosophers)
10. Produtores e Consumidores (ProdCons)
11. Tipo de dados *node* (Node)
12. Apontadores para Abstrações (Abstractions)
13. Passagem de Parâmetros de Entrada e Saída (ByRef)

Construção	1	2	3	4	5	6	7	8	9	10	11	12	13
<abstraction>												★	
<action>	★										★	★	★
<actioncall>	★			★	★	★	★	★	★	★	★	★	★
<actions>	★	★	★	★	★	★	★	★	★	★	★	★	★
<actionsignature>	★					★	★		★	★	★	★	★
<add>	★	★	★		★				★	★		★	
<agent>	★	★	★	★	★	★	★	★	★	★	★	★	★
<agents>	★	★	★	★	★	★	★	★	★	★	★	★	★
<and>							★						
<any>						★							
<block>	★	★	★	★	★	★	★	★	★	★	★	★	★
<boolean>				★		★	★		★				
<case>						★		★	★	★			
<caseexp>							★						
<character>	★					★	★	★					
<choose>				★									
<choosealias>				★									
<conditional>	★	★	★	★					★	★			★
<cons>							★						
<create>								★	★	★			
<derivedfunction>					★	★			★	★			
<derivedfunctioncall>									★	★			
<destroy>								★					
<diff>					★								
<disjointunion>						★							
<dispatch>								★	★	★			
					★	★							
<dynamicfunction>	★	★	★	★	★	★			★	★	★	★	★
<dynamicfunctioncall>	★	★	★	★	★	★	★	★	★	★	★	★	★
<dynamicfunctions>	★	★	★	★	★	★	★	★	★	★	★	★	★
<dynamicfunctionsignature>	★								★	★			
<environment>	★	★	★	★	★	★	★	★	★	★	★	★	★
<equal>	★			★									★
<expression>	★	★	★	★	★	★	★	★	★	★	★	★	★
<externalfunction>	★				★			★		★			
<externalfunctioncall>	★			★	★	★	★	★	★	★	★	★	★
<externalfunctions>	★	★	★	★	★	★	★	★	★	★	★	★	★
<externalfunctionsignature>	★					★	★		★	★	★	★	★
<forall>				★	★	★	★		★				
<forallalias>				★	★	★	★		★				

Tabela 7.2: Avaliação dos exemplos frente às construções da linguagem intermediária. (Parte 1 de 3.)

Construção	1	2	3	4	5	6	7	8	9	10	11	12	13
<functional>	★	★		★	★	★	★	★	★	★	★	★	★
<ifexp>	★				★	★	★			★			
<immediateupdate>	★			★					★	★			
<import>	★					★	★		★	★	★	★	★
<imports>	★	★	★	★	★	★	★	★	★	★	★	★	★
<in>						★							
<init>	★	★	★	★	★	★	★	★	★	★	★	★	★
<integer>	★	★	★	★	★	★	★	★	★	★	★	★	★
<interval>						★							
<lambda>	★				★	★	★	★	★	★	★		
<lessequalthan>					★								
<lessthan>	★	★	★						★	★			
<let>					★								
<letalias>					★								
<letexp>					★								
<letexpalias>					★								
<list>						★	★						
<listagregate>						★	★						
<literalboolean>				★		★	★		★				
<literalcharacter>						★	★	★					
<literalinteger>	★	★	★	★	★	★			★	★		★	★
<literalreal>					★	★							★
<literalstring>	★			★	★	★	★	★	★	★	★	★	★
<mas>	★	★	★	★	★	★	★	★	★	★	★	★	★
<minus>	★	★			★					★			
<minusunary>					★								
<mod>	★				★				★				
<module>	★	★	★	★	★	★	★	★	★	★	★	★	★
<moduleref>	★			★	★	★	★	★	★	★	★	★	★
<moreequalthan>					★								
<morethan>					★								
<mult>					★	★						★	
<name>	★	★	★	★	★	★	★	★	★	★	★	★	★
<node>											★		
<not>					★				★				
<or>							★						
<parametercall>	★				★	★	★			★	★	★	★
<private>	★	★	★	★	★	★	★	★	★	★	★	★	★
<public>	★	★	★	★	★	★	★	★	★	★	★	★	★
<real>	★				★	★							★

Tabela 7.3: Avaliação dos exemplos frente às construções da linguagem intermediária. (Parte 2 de 3.)

Construção	1	2	3	4	5	6	7	8	9	10	11	12	13
<reference>								★	★	★			
<references>	★	★	★	★	★	★	★	★	★	★	★	★	★
<return>	★												
<rule>	★	★	★	★	★	★	★	★	★	★	★	★	★
<set>				★	★	★			★				
<setagregate>				★	★	★			★				
<signatures>	★					★	★		★	★	★	★	★
<staticandderivedfunctions>	★	★	★	★	★	★	★	★	★	★	★	★	★
<staticfunction>	★				★		★		★	★			
<staticfunctioncall>	★				★		★		★	★			
<staticfunctionsignature>	★				★					★			
<stop>	★	★	★	★	★	★	★	★		★	★	★	★
<string>	★			★	★	★	★	★	★	★	★	★	★
<submachine>	★												
<submachinecall>	★												
<submachines>	★	★	★	★	★	★	★	★	★	★	★	★	★
<submachinesignature>	★												
<transition>	★	★	★	★	★	★	★	★	★	★	★	★	★
<tuple>				★			★				★		
<tupleagregate>				★			★				★		
<tupleprojection>				★			★				★		
<type>	★	★	★	★	★	★	★	★	★	★	★	★	★
<typecast>						★					★		
<update>	★	★	★	★	★	★			★	★	★	★	★
<with>						★					★		
<withalias>						★					★		
<withexp>						★							
<withexpalias>						★							
<xor>							★						

Tabela 7.4: Avaliação dos exemplos frente às construções da linguagem intermediária. (Parte 3 de 3.)

7.2.3 Integração com ACOA

Como validação do kℓar como *back-end* para compiladores de linguagens orientadas pela metodologia ASM, pode-se citar o trabalho de Lobato [Lob05]. Este trabalho contempla a criação de um arcabouço, chamado ACOA, que tem por finalidade a geração de automática de compiladores para linguagens especificadas dentro deste arcabouço. Os compiladores gerados pelo ACOA compilam programas escritos nas linguagens do modelo ASM para a linguagem MIR, e utiliza o kℓar para compilar os programas em MIR para código C++.

Além da implementação do ACOA, Lobato também escreveu a especificação da linguagem Machĭna dentro deste arcabouço. Desta forma, é possível compilar, por meio do ACOA, programas escritos em Machĭna para MIR, e então faz-se uso do kℓar para a compilação de MIR para C++. A Figura 6.1 ilustrou a integração dos dois projetos.

7.3 Conclusões

O arcabouço kℓar foi validado segundo duas abordagens distintas. A primeira destas abordagens considera o aspecto da facilidade de desenvolvimento de otimizações ASM dentro do contexto do kℓar. Para tanto, uma otimização ASM foi implementada fazendo-se uso das facilidades providas pelo kℓar.

Em primeiro lugar, o trabalho de coleta de informações para a análise que precede a otimização e montagem do grafo de conflitos foi facilitado pela existência de um visitor que percorre o módulo obtendo as informações de uso e definição de funções dinâmicas em cada sub-regra da regra de transição.

Do ponto de vista da implementação do algoritmo de otimização em si, a organização da regra de transição como uma estrutura em forma de árvore permitiu que o cerne do algoritmo de otimização fosse implementado como um visitor também. Esta abordagem facilitou a implementação do caráter recursivo da otimização.

O desenvolvimento da otimização pode ser feito como um projeto isolado, não sendo necessário ao desenvolvedor da otimização conhecer os detalhes internos do arcabouço kℓar, bastando apenas conhecer o protocolo que as otimizações devem seguir. Este protocolo consiste em herdar de Optimizer e então implementar o método virtual optimize.

A segunda abordagem de validação considera o uso do kℓar como *back-end* para compiladores de linguagens ASM. Além dos exemplos apresentados no Apêndice B, que validam a implementação da linguagem MIR, foi apresentado o trabalho de Lobato [Lob05], que fez uso do kℓar como *back-end* para os compiladores gerados por seu arcabouço ACOA.

Capítulo 8

Conclusões

O kℓar foi desenvolvido com o objetivo principal de oferecer o ambiente apropriado para o desenvolvimento de otimizações específicas para o modelo de máquinas de estado abstratas. Com efeito, tal objetivo foi alcançado. Otimizações a serem desenvolvidas podem usar os recursos oferecidos pelo arcabouço, apresentados neste trabalho. Mais do que isto, a forma como as otimizações devem ser desenvolvidas garante o acoplamento dinâmico das mesmas, permitindo flexibilidade no uso de otimizações desenvolvidas. Uma otimização foi implementada com o objetivo de ilustrar estes aspectos do kℓar.

A infra-estrutura desenvolvida para o suporte da linguagem MIR permite a representação em memória de programas escritos nesta linguagem, bem como a geração de código C++ a partir desta representação. A sintaxe XML proposta, a geração de código XML a partir da representação em memória de programas em MIR e o carregador desenvolvido para a criação destas representações a partir do código XML permitem a persistência de programas MIR. Isto tudo torna o arcabouço kℓar um candidato natural para ser usado como alvo de compiladores de linguagens que se inspiram na metodologia de ASM.

O trabalho apresenta contribuições para a pesquisa na área de máquinas de estado abstratas. Além disso, este trabalho também abre caminho para que novas pesquisas sejam realizadas. Este capítulo apresenta estas contribuições e desdobramentos futuros. A principais contribuições do trabalho desenvolvido são:

Arcabouço para Otimizações A principal contribuição deste trabalho é fornecer um arcabouço para máquinas de estado abstratas onde otimizações possam ser facilmente adicionadas, configuradas e removidas. Este arcabouço serve também como *back-end* para compiladores de linguagens que seguem o paradigma de ASM. O intuito é permitir a pesquisa de otimizações específicas da metodologia de ASM e, com isto, ser capaz de gerar código eficiente a partir de especificações formais de alto nível.

Implementação da Linguagem MIR Para que especificações ASM sejam entendidas no contexto do kℓar, estas devem estar escritas em uma linguagem que seja entendida pelo arcabouço. Esta linguagem é a linguagem MIR [Tir00, Oli04, OBB04a]. Foi implementada uma infra-estrutura para suporte a esta linguagem, que consiste nos seguintes elementos:

Classes para Representação em Memória Para todas as construções da linguagem MIR existe uma classe correspondente que exibe a interface apropriada. Estas clas-

ses foram propostas e implementadas neste trabalho, e são utilizadas para a representação em memória de programas a serem otimizados.

Gerador de Código C++ Foi projetado e implementado um gerador de código que traduz uma representação em memória de um programa em linguagem MIR em código C++ equivalente, permitindo a execução de especificações ASM escritas na linguagem MIR. Este gerador foi implementado de tal forma que alterações no mapeamento de partes da linguagem MIR em código C++ podem ser facilmente realizadas.

Persistência em XML Para a persistência em memória não-volátil de programas em linguagem MIR, foi desenvolvido um mecanismo de tradução das representações em memória para arquivos XML que descrevessem estas representações. Adicionalmente, foi desenvolvido também um carregador de programas em MIR a partir destas descrições em XML. Neste trabalho, foi definida também a sintaxe XML para esta representação, dada por um arquivo XSD.

Modelo de Concorrência Fez-se uma modificação na linguagem MIR com a proposição e implementação de um modelo de concorrência geral o bastante para permitir que modelos mais avançados possam ser implementados por meio dos mecanismos de sincronização oferecidos. Este modelo é baseado no conceito de multi-agentes exposto anteriormente.

O arcabouço aqui apresentado abre caminho para novos trabalhos a serem desenvolvidos. A seguir, são enumeradas algumas destas possibilidades.

Desenvolvimento de uma Biblioteca Padrão Um dos componentes de sucesso de uma linguagem de programação é a presença de uma biblioteca padrão abrangente. A existência de tal biblioteca permite ao programador se concentrar no desenvolvimento daquilo que é realmente novo, aproveitando o código de uso geral que não diz respeito diretamente ao seu problema em particular. Um passo importante para a utilização da linguagem MIR, e conseqüentemente do arcabouço kℓar, é o desenvolvimento de uma biblioteca padrão que contemple elementos como processamento de strings, funções matemáticas, estruturas de dados mais elaboradas, protocolos de comunicação via rede, componentes de programação multi-agentes de mais alto nível, etc. Como o arcabouço foi desenvolvido em C++ e o código gerado a partir de especificações ASM é em C++, um ponto de partida interessante seria disponibilizar a STL de C++ com a roupagem da linguagem MIR.

Estudo de Padrões de Projeto Específicos da Programação ASM Padrões de projeto são soluções amplamente utilizadas para problemas de projeto em linguagens orientadas por objeto. Uma extensão de grande utilidade para a metodologia ASM seria o estudo de padrões de projeto específicos desta metodologia. Padrões específicos para máquinas de estado abstratas são necessários devido às diferenças entre estas e programas orientados por objetos. Particularmente, conceitos como herança e sobrecarga não estão presentes em ASM. Além disso, existe também o paralelismo intrínseco de uma regra de transição, diferente da programação imperativa tradicional.

Utilização do kℓar para Refabricação A refatoração consiste na transformação de código com o objetivo de melhorá-lo do ponto de vista do projeto [Fow99]. Como a atuação das refabricações se dá por meio da transformação de programas, estas se assemelham às

otimizações extrínsecas do kℓar. Desta forma, o arcabouço kℓar pode ser utilizado para a pesquisa e implementação de refatorações para ASM.

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 MIR.

Elaboração de uma Notação Semelhante à UML Uma das fases que precedem a codificação em um projeto de software é a modelagem. Nesta fase, uma notação gráfica para a representação de conceitos pode ser de grande valia. A notação mais utilizada para tanto, principalmente no contexto de programação orientada por objetos, é a UML. Entretanto, mais uma vez as características da metodologia ASM sugerem a existência de uma notação específica para esta metodologia, ou ao menos uma adaptação de uma notação existente. O resultado seria uma notação que representasse módulos, agentes e os diversos tipos de funções e abstrações de regra, bem como a relação entre estes elementos.

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 o kℓar seriam de grande valia para a utilização do arcabouço de forma mais produtiva. Uma ferramenta deste tipo poderia, inclusive, verificar o estado de uma ASM a cada passo de execução por meio do padrão *Observer*.

A pesquisa de otimizações específicas para o modelo ASM é um campo novo e ainda está longe de ser totalmente explorado. Com o desenvolvimento do kℓar, espera-se ter contribuído com esta pesquisa, provendo um ambiente onde estas otimizações possam ser implementadas e estudadas. Como resultado, espera-se que o poder da especificação formal de algoritmos oferecido pela metodologia de ASM esteja cada vez mais próximo da geração automática de código que atenda aos requisitos de eficiência necessários em ambientes profissionais.

Referências Bibliográficas

[Att04] Wesley Attrot. Xingó - compilação para uma representação intermediária executável. Master's thesis, Universidade Estadual de Campinas, 2004.

[BGR95] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998.

[BM97] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1997.

[BR94] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[BS97] D. Bèauquier and A. Slissenko. On Semantics of Algorithms with Continuous Time. Technical Report 97-15, Dept. of Informatics, Université Paris-12, October 1997.

[BS03] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[BTIB05] Roberto Da Silva Bigonha, Fábio Tirelo, Vladimir Oliveira Di Iorio, and Mariza Andrade Da Silva Bigonha. A linguagem de especificação formal machina 2.0. Technical Report LLP001/2005, Universidade Federal de Minas Gerais, 2005.

[BTM+99] Roberto Da Silva Bigonha, Fábio Tirelo, Marcelo De Almeida Maia, Marcelo Silva, Marco Túlio De Oliveira Valente, Mariza A. S. Bigonha, and Vladimir Oliveira Di Iorio. Projeto Machina. Technical Report LLP007/99, Universidade Federal de Minas Gerais, Julho 1999.

[But97] David R. Butenhof. *Programming with POSIX(R) Threads.* Addison-Wesley Professional, 1997.

[Car97] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.

[CLR90] T. H. Cormem, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw Hill, 1990.

[DDG96] G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra Abstract Machine. In H. Kleine Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 191–214. Springer, 1996.

[Del99] G. Del Castillo. Towards Comprehensive Tool Support for Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods — FM-Trends 98*, volume 1641 of *LNCS*, pages 311–325. Springer-Verlag, 1999.

[Del00] Giuseppe Del Castillo. *The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models.* PhD thesis, Universität Paderborn, 2000.

[Dox] Doxygen. The Doxygen Homepage: www.doxygen.org.

[EA01a] B. Eckel and C. Allison. *Thinking in C++*, volume 1. MindView, Inc, http://www.mindview.net/books, 2. edition, 2001.

[EA01b] B. Eckel and C. Allison. *Thinking in C++*, volume 2. MindView, Inc, http://www.mindview.net/books, 2. edition, 2001.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[Fow04] Martin Fowler. *UML Distilled.* Addison-Wesley, 3rd edition, 2004.

[GCC] GCC. The GCC Homepage: gcc.gnu.org.

[GH93] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GK97] U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[GM95] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 295–328. Oxford University Press, 1995.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[Gur91] Yuri Gurevich. Evolving algebras: An attempt to discover semantics, 1991.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur00] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[HW02] J. Huggins and C. Wallace. An Abstract State Machine Primer. Technical Report CS-TR-02-04, Computer Science Department, Michigan Technological University, 4 December 2002.

[Joh97a] Ralph E. Johnson. Components, frameworks, patterns. In *Proceedings of the 1997 symposium on Software reusability*, pages 10–17. ACM Press, 1997.

[Joh97b] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997.

[KP97] P. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[Lia99] Sheng Liang. *The Java Native Interface: Programmer's Gude and Specification.* Addison Wesley, 1999.

[Lob05] Mário Celso Candian Lobato. Proposta de Dissertação: Um Arcabouço para Compilação de Linguagens de Especificação ASM, 2005.

[Mat96] Michael Mattsson. Object-oriented frameworks - a survey of methodological issues. Licenciate Thesis, 1996. Lund University.

[MBB+05] Kristian Magnani, Mariza A. S. Bigonha, Roberto S. Bigonha, Vladimir O. Di Iorio, and Fabíola F. Oliveira. An Infrastructure for Implementing Compilers for Abstract State Machines. In *31th Latin American Conference on Informatics*, 2005.

[Mey96] Scott Meyers. *More Effective C++.* Addison-Wesley, 1996.

[Mey98] Scott Meyers. *Effective C++.* Addison-Wesley, 2nd edition, 1998.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, 1999.

[OBB04a] F. Oliveira, M. Bigonha, and R. Bigonha. MIR: Machĭna Intermediate Representation. Technical Report RT001/04, Laboratório de Linguagens de Programação - Departamento de Ciência da Computação - Universidade Federal de Minas Gerais, 2004.

[OBB04b] Fabíola F. Oliveira, Roberto S. Bigonha, and Mariza A. Silva Bigonha. Otimização de Código em Ambiente de Semântica Formal Executável Baseado em ASM. *Proceedings of 8th Brazilian Symposium on Programming Languages*, pages 172–185, May 2004.

[Oli04] Fabíola F. Oliveira. Otimização de Código em Ambiente de Semântica Formal Executável Baseado em ASM. 2004.

[Pil03] Dan Pilone. *UML Pocket Reference*. O'Reilly, 2003.

[Str97] K. Stroetmann. The Constrained Shortest Path Problem: A Case Study In Using ASMs. *Journal of Universal Computer Science*, 3(4):304–319, 1997.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[Tan99] Andrew S. Tanenbaum. *Structured Computer Organization* . Addison-Wesley Professional, 5th edition, 1999.

[TB00] Fábio Tirelo and Roberto S. Bigonha. Técnicas de Otimização de Programas Baseados em Máquinas de Estado Abstratas. *Proceedings of 4th Brazilian Symposium on Programming Languages*, pages 144–157, 2000.

[Tir00] Fábio Tirelo. Uma ferramenta para execução de um sistema dinâmico discreto baseado em Álgebras evolutivas. Master's thesis, Universidade Federal de Minas Gerais, 2000.

[TMIB99a] Fábio Tirelo, Marcelo De Almeida Maia, Vladimir Oliveira Di Iorio, and Roberto Da Silva Bigonha. Máquina de estado abstratas. Technical Report LLP015/99, Universidade Federal de Minas Gerais, Junho 1999.

[TMIB99b] Fábio Tirelo, Marcelo De Almeida Maia, Vladimir Oliveira Di Iorio, and Roberto Da Silva Bigonha. Projeto Machina: A linguagem de especificação ASM. Technical Report LLP008/99, Universidade Federal de Minas Gerais, Julho 1999.

[Ton04] Adriana Pinheiro Toni. Desenvolvimento de um framework para aplicações comerciais. Monografia, 2004. Universidade Federal de Minas Gerais.

[Wal97] C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, December 1997.

[ZJW03] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.

Apêndice A

O Arquivo de Serialização MIR

Definição do Arquivo de Serialização MIR - *XML-like*

```
<?xml version="1.0"?>
<!--
  Definition of the text representation of a module

  Developed by Kristian Magnani in 23rd March 2005.
  (kristian@ufmg.br, kristian.magnani@gmail.com)
-->
<xs:schema targetNamespace="http://www.dcc.ufmg.br/llp" elementFormDefault="qualified" xmlns:xs="h
  <!-- The name element -->
  <xs:element name="name">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:minLength value="1"/>
        <xs:maxLength value="31"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <!-- The moduleref element -->
  <xs:element name="moduleref">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The agentref element -->
  <xs:element name="agentref">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="expression"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The agent element -->
  <xs:element name="agent">
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element ref="moduleref"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The literalboolean expression element -->
  <xs:element name="literalboolean">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="true"/>
        <xs:enumeration value="false"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <!-- The literalcharacter expression element -->
  <xs:element name="literalcharacter">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:minLength value="1"/>
        <xs:maxLength value="1"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <!-- The literalinteger expression element -->
  <xs:element name="literalinteger" type="xs:integer"/>
  <!-- The literalreal expression element -->
  <xs:element name="literalreal" type="xs:double"/>
  <!-- The literalstring expression element -->
  <xs:element name="literalstring" type="xs:string"/>
  <!-- The self expression element -->
  <xs:element name="self" type="xs:string"/>
  <!-- The minusunary expression element -->
  <xs:element name="minusunary">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="expression"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The not expression element -->
  <xs:element name="not">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="expression"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The typecast expression element -->
  <xs:element name="typecast">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="type"/>
        <xs:element ref="expression"/>
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The mult expression element -->
<xs:element name="mult">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The div expression element -->
<xs:element name="div">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The mod expression element -->
<xs:element name="mod">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The add expression element -->
<xs:element name="add">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The minus expression element -->
<xs:element name="minus">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The intersection expression element -->
<xs:element name="intersection">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
```

```
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The equal expression element -->
<xs:element name="equal">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The diff expression element -->
<xs:element name="diff">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The lessthan expression element -->
<xs:element name="lessthan">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The morethan expression element -->
<xs:element name="morethan">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The lessequalthan expression element -->
<xs:element name="lessequalthan">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The moreequalthan expression element -->
<xs:element name="moreequalthan">
  <xs:complexType>
    <xs:sequence>
```

```
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The and expression element -->
<xs:element name="and">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The or expression element -->
<xs:element name="or">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The xor expression element -->
<xs:element name="xor">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The cons expression element -->
<xs:element name="cons">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The in expression element -->
<xs:element name="in">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The staticfunctioncall expression element -->
<xs:element name="staticfunctioncall">
  <xs:complexType>
```

```
    <xs:sequence>
      <xs:element name="qualification">
        <xs:complexType>
          <xs:sequence>
            <xs:sequence minOccurs="0">
              <xs:element ref="agentref" minOccurs="0"/>
              <xs:element ref="moduleref"/>
            </xs:sequence>
            <xs:element ref="name"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The derivedfunctioncall expression element -->
<xs:element name="derivedfunctioncall">
  <xs:complexType>
    <xs:sequence>
      <xs:sequence minOccurs="0">
        <xs:element ref="agentref" minOccurs="0"/>
        <xs:element ref="moduleref"/>
      </xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The dynamicfunctioncall expression element -->
<xs:element name="dynamicfunctioncall">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="qualification">
        <xs:complexType>
          <xs:sequence>
            <xs:sequence minOccurs="0">
              <xs:element ref="agentref" minOccurs="0"/>
              <xs:element ref="moduleref"/>
            </xs:sequence>
            <xs:element ref="name"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="index" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="expression"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
```

```
  </xs:complexType>
</xs:element>
<!-- The externalfunctioncall expression element -->
<xs:element name="externalfunctioncall">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="qualification">
        <xs:complexType>
          <xs:sequence>
            <xs:sequence minOccurs="0">
              <xs:element ref="agentref" minOccurs="0"/>
              <xs:element ref="moduleref"/>
            </xs:sequence>
            <xs:element ref="name"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The parametercall element -->
<xs:element name="parametercall">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The tuple agregate expression element -->
<xs:element name="tupleagregate">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The tuple projection expression element -->
<xs:element name="tupleprojection">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="literalinteger"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The set agregate expression element -->
<xs:element name="setagregate">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
```

```
</xs:element>
<!-- The list agregate expression element -->
<xs:element name="listagregate">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The tail expression element -->
<xs:element name="tail">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The head expression element -->
<xs:element name="head">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The initial expression element -->
<xs:element name="initial">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The last expression element -->
<xs:element name="last">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The ifexp expression element -->
<xs:element name="ifexp">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The letexp expression element -->
<xs:element name="letexp">
```

```
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The caseexp expression element -->
<xs:element name="caseexp">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="expression"/>
        <xs:element ref="expression"/>
      </xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The withexp expression element -->
<xs:element name="withexp">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="name"/>
        <xs:element ref="type"/>
        <xs:element ref="expression"/>
      </xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The letalias expression element -->
<xs:element name="letalias">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The forallalias expression element -->
<xs:element name="forallalias">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The choosealias expression element -->
<xs:element name="choosealias">
```

```
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The withalias expression element -->
<xs:element name="withalias">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The letexpalias expression element -->
<xs:element name="letexpalias">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The withexpalias expression element -->
<xs:element name="withexpalias">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The nodeagregate expression element -->
<xs:element name="nodeagregate">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The nodelabel expression element -->
<xs:element name="nodelabel">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The nodecontent expression element -->
<xs:element name="nodecontent">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
```

```
  </xs:complexType>
</xs:element>
<!-- The undef expression element -->
<xs:element name="undef" type="xs:string"/>
<!-- The any type element -->
<xs:element name="any" type="xs:string"/>
<!-- The boolean type element -->
<xs:element name="boolean" type="xs:string"/>
<!-- The character type element -->
<xs:element name="character" type="xs:string"/>
<!-- The integer type element -->
<xs:element name="integer" type="xs:string"/>
<!-- The real type element -->
<xs:element name="real" type="xs:string"/>
<!-- The string type element -->
<xs:element name="string" type="xs:string"/>
<!-- The disjointunion type element -->
<xs:element name="disjointunion">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="type" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The interval type element -->
<xs:element name="interval">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The tuple type element -->
<xs:element name="tuple">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="type" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The list type element -->
<xs:element name="list">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="type"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The set type element -->
<xs:element name="set">
  <xs:complexType>
    <xs:sequence>
```

```
      <xs:element ref="type"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The functional type element -->
<xs:element name="functional">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="type"/>
      <xs:element ref="type"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The node type element -->
<xs:element name="node">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="literalstring"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The label attribute, used for rule nodes -->
<xs:attribute name="label" type="xs:integer" use="required"/>
<!-- The next attribute, used (optionally) for rule nodes -->
<xs:attribute name="next" type="xs:integer" use="optional"/>
<!-- The update rule element -->
<xs:element name="update">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dynamicfunctioncall"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The immediate update rule element -->
<xs:element name="immediateupdate">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dynamicfunctioncall"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The create rule element -->
<xs:element name="create">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:element ref="moduleref"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<!-- The dispatch rule element -->
<xs:element name="dispatch">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The destroy rule element -->
<xs:element name="destroy">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The stop rule element -->
<xs:element name="stop" type="xs:string"/>
<!-- The return rule element -->
<xs:element name="return" type="xs:string"/>
<!-- The actioncall rule element -->
<xs:element name="actioncall">
  <xs:complexType>
    <xs:sequence>
      <xs:sequence minOccurs="0">
        <xs:element ref="agentref" minOccurs="0"/>
        <xs:element ref="moduleref"/>
      </xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The submachinecall rule element -->
<xs:element name="submachinecall">
  <xs:complexType>
    <xs:sequence>
      <xs:sequence minOccurs="0">
        <xs:element ref="agentref" minOccurs="0"/>
        <xs:element ref="moduleref"/>
      </xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The lambda rule element -->
<xs:element name="lambda" type="xs:string"/>
<!-- The conditional rule element -->
<xs:element name="conditional">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
```

```
      <xs:element ref="rule"/>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The forall rule element -->
<xs:element name="forall">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="expression"/>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The choose rule element -->
<xs:element name="choose">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="expression"/>
      <xs:element ref="expression"/>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The let rule element -->
<xs:element name="let">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="expression"/>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The case rule element -->
<xs:element name="case">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="expression"/>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="expression"/>
        <xs:element ref="rule"/>
      </xs:sequence>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The with rule element -->
<xs:element name="with">
  <xs:complexType>
    <xs:sequence>
```

```
      <xs:element ref="expression"/>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="name"/>
        <xs:element ref="type"/>
        <xs:element ref="rule"/>
      </xs:sequence>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The block rule element -->
<xs:element name="block">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="rule"/>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The reset rule element -->
<xs:element name="reset">
  <xs:complexType>
    <xs:sequence>
      <xs:sequence minOccurs="0">
        <xs:element ref="agentref" minOccurs="0"/>
        <xs:element ref="moduleref"/>
      </xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="literalinteger"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The signal rule element -->
<xs:element name="signal">
  <xs:complexType>
    <xs:sequence>
      <xs:sequence minOccurs="0">
        <xs:element ref="agentref" minOccurs="0"/>
        <xs:element ref="moduleref"/>
      </xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The wait rule element -->
<xs:element name="wait">
  <xs:complexType>
    <xs:sequence>
      <xs:sequence minOccurs="0">
        <xs:element ref="agentref" minOccurs="0"/>
        <xs:element ref="moduleref"/>
      </xs:sequence>
      <xs:element ref="name"/>
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The label element -->
<xs:element name="label">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="literalinteger"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The next element -->
<xs:element name="next">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="literalinteger"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The rule element -->
<xs:element name="rule">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="label" minOccurs="0"/>
      <xs:element ref="next" minOccurs="0"/>
      <xs:choice>
        <xs:element ref="update"/>
        <xs:element ref="immediateupdate"/>
        <xs:element ref="create"/>
        <xs:element ref="dispatch"/>
        <xs:element ref="destroy"/>
        <xs:element ref="stop"/>
        <xs:element ref="return"/>
        <xs:element ref="actioncall"/>
        <xs:element ref="submachinecall"/>
        <xs:element ref="lambda"/>
        <xs:element ref="conditional"/>
        <xs:element ref="forall"/>
        <xs:element ref="choose"/>
        <xs:element ref="let"/>
        <xs:element ref="case"/>
        <xs:element ref="with"/>
        <xs:element ref="block"/>
        <xs:element ref="reset"/>
        <xs:element ref="signal"/>
        <xs:element ref="wait"/>
        <xs:element name="abstractioncall">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="expression"/>
              <xs:sequence>
                <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
```

```
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The expression element -->
<xs:element name="expression">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="literalboolean"/>
      <xs:element ref="literalcharacter"/>
      <xs:element ref="literalinteger"/>
      <xs:element ref="literalreal"/>
      <xs:element ref="literalstring"/>
      <xs:element ref="self"/>
      <xs:element ref="minusunary"/>
      <xs:element ref="not"/>
      <xs:element ref="typecast"/>
      <xs:element ref="mult"/>
      <xs:element ref="div"/>
      <xs:element ref="mod"/>
      <xs:element ref="add"/>
      <xs:element ref="minus"/>
      <xs:element ref="intersection"/>
      <xs:element ref="equal"/>
      <xs:element ref="diff"/>
      <xs:element ref="lessthan"/>
      <xs:element ref="morethan"/>
      <xs:element ref="lessequalthan"/>
      <xs:element ref="moreequalthan"/>
      <xs:element ref="and"/>
      <xs:element ref="or"/>
      <xs:element ref="xor"/>
      <xs:element ref="cons"/>
      <xs:element ref="in"/>
      <xs:element ref="staticfunctioncall"/>
      <xs:element ref="derivedfunctioncall"/>
      <xs:element ref="dynamicfunctioncall"/>
      <xs:element ref="externalfunctioncall"/>
      <xs:element ref="parametercall"/>
      <xs:element ref="tupleagregate"/>
      <xs:element ref="tupleprojection"/>
      <xs:element ref="setagregate"/>
      <xs:element ref="listagregate"/>
      <xs:element ref="tail"/>
      <xs:element ref="head"/>
      <xs:element ref="initial"/>
      <xs:element ref="last"/>
      <xs:element ref="ifexp"/>
      <xs:element ref="letexp"/>
      <xs:element ref="caseexp"/>
```

```
      <xs:element ref="withexp"/>
      <xs:element ref="letalias"/>
      <xs:element ref="forallalias"/>
      <xs:element ref="choosealias"/>
      <xs:element ref="withalias"/>
      <xs:element ref="letexpalias"/>
      <xs:element ref="withexpalias"/>
      <xs:element ref="nodeagregate"/>
      <xs:element ref="nodelabel"/>
      <xs:element ref="nodecontent"/>
      <xs:element ref="undef"/>
      <xs:element name="abstractionreference">
        <xs:complexType>
          <xs:sequence>
            <xs:sequence minOccurs="0">
              <xs:element ref="agentref" minOccurs="0"/>
              <xs:element ref="moduleref"/>
            </xs:sequence>
            <xs:element ref="name"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="setconstructor">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="expression"/>
            <xs:element ref="expression"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="listconstructor">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="expression"/>
            <xs:element ref="expression"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
<!-- The type element -->
<xs:element name="type">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="any"/>
      <xs:element ref="boolean"/>
      <xs:element ref="character"/>
      <xs:element ref="integer"/>
      <xs:element ref="real"/>
      <xs:element ref="string"/>
      <xs:element ref="disjointunion"/>
      <xs:element ref="interval"/>
```

```
      <xs:element ref="tuple"/>
      <xs:element ref="list"/>
      <xs:element ref="set"/>
      <xs:element ref="functional"/>
      <xs:element ref="node"/>
      <xs:element name="abstraction">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="type" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
<!-- The init element -->
<xs:element name="init">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The transition element -->
<xs:element name="transition">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The static function element -->
<xs:element name="staticfunction">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="type"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The derived function element -->
<xs:element name="derivedfunction">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="type"/>
      <xs:element ref="expression"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The dynamic function element -->
<xs:element name="dynamicfunction">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="type"/>
        <xs:element ref="expression" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The external function element -->
  <xs:element name="externalfunction">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="type"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The parameter element -->
  <xs:element name="parameter">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="type"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The action element -->
  <xs:element name="action">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="name"/>
          <xs:element ref="type"/>
          <xs:element name="byref" minOccurs="0"/>
        </xs:sequence>
        <xs:element ref="rule"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- The submachine element -->
  <xs:element name="submachine">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="name"/>
          <xs:element ref="type"/>
          <xs:element name="byref" minOccurs="0"/>
        </xs:sequence>
        <xs:element ref="init" minOccurs="0"/>
        <xs:element ref="rule"/>
      </xs:sequence>
```

```
  </xs:complexType>
</xs:element>
<!-- The semphore element -->
<xs:element name="semaphore">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="literalinteger"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The table of static and derived functions -->
<xs:element name="staticandderivedfunctions">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="staticfunction"/>
      <xs:element ref="derivedfunction"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
<!-- The table of dynamic functions -->
<xs:element name="dynamicfunctions">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dynamicfunction" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The table of external functions -->
<xs:element name="externalfunctions">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="externalfunction" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The table of actions -->
<xs:element name="actions">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="action" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The table of submachines -->
<xs:element name="submachines">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="submachine" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The table of semaphores -->
```

```
<xs:element name="semaphores">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="semaphore" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The environment element -->
<xs:element name="environment">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="staticandderivedfunctions" minOccurs="0"/>
      <xs:element ref="dynamicfunctions" minOccurs="0"/>
      <xs:element ref="externalfunctions" minOccurs="0"/>
      <xs:element ref="actions" minOccurs="0"/>
      <xs:element ref="submachines" minOccurs="0"/>
      <xs:element ref="semaphores" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The local element -->
<xs:element name="local">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="environment"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The global element -->
<xs:element name="global">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="environment"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The actionsignature element -->
<xs:element name="actionsignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="name"/>
        <xs:element ref="type"/>
        <xs:element name="byref" minOccurs="0"/>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The submachinesignature element -->
<xs:element name="submachinesignature">
  <xs:complexType>
    <xs:sequence>
```

```
      <xs:element ref="name"/>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="name"/>
        <xs:element ref="type"/>
        <xs:element name="byref" minOccurs="0"/>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The staticfunctionsignature element -->
<xs:element name="staticfunctionsignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="type"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The derivedfunctionsignature element -->
<xs:element name="derivedfunctionsignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="type"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The dynamicfunctionsignature element -->
<xs:element name="dynamicfunctionsignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="type"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The externalfunctionsignature element -->
<xs:element name="externalfunctionsignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="type"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The semaphoresignature element -->
<xs:element name="semaphoresignature">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<!-- The signatures element -->
<xs:element name="signatures">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="actionsignature"/>
      <xs:element ref="submachinesignature"/>
      <xs:element ref="staticfunctionsignature"/>
      <xs:element ref="derivedfunctionsignature"/>
      <xs:element ref="dynamicfunctionsignature"/>
      <xs:element ref="externalfunctionsignature"/>
      <xs:element ref="semaphoresignature"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
<!-- The import element -->
<xs:element name="import">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="signatures" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The table of imports -->
<xs:element name="imports">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The reference element -->
<xs:element name="reference">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="signatures" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The table of references -->
<xs:element name="references">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="reference" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- The module element -->
<xs:element name="module">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
```

```
        <xs:element ref="references" minOccurs="0"/>
        <xs:element ref="imports" minOccurs="0"/>
        <xs:element ref="global"/>
        <xs:element ref="local"/>
        <xs:element ref="init" minOccurs="0"/>
        <xs:element ref="transition"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Apêndice B

Exemplos Utilizados na Validação do kℓar

B.1 Código Fonte dos Módulos

B.1.1 Counting.mod

```
<module>

  <name>Counting</name>

  <references>
  </references>

  <imports>
  </imports>

  <global>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>

      <dynamicfunctions>

      </dynamicfunctions>

      <externalfunctions>

      </externalfunctions>

      <actions>

      </actions>

      <submachines>
```

```
    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

      <dynamicfunction>
        <name>i</name>
        <type><integer/></type>
      </dynamicfunction>

      <dynamicfunction>
        <name>j</name>
        <type><integer/></type>
      </dynamicfunction>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>

<init>
  <rule>
    <block>
      <rule>
        <update>
          <dynamicfunctioncall>
            <name>i</name>
          </dynamicfunctioncall>
```

```
            <expression>
              <literalinteger>0</literalinteger>
            </expression>
          </update>
        </rule>
        <rule>
          <update>
            <dynamicfunctioncall>
              <name>j</name>
            </dynamicfunctioncall>
            <expression>
              <literalinteger>0</literalinteger>
            </expression>
          </update>
        </rule>
      </block>
    </rule>
  </init>

  <transition>
    <rule>
      <conditional>
        <expression>
          <lessthan>
            <expression>
              <dynamicfunctioncall>
                <name>i</name>
              </dynamicfunctioncall>
            </expression>
            <expression>
              <literalinteger>10000000</literalinteger>
            </expression>
          </lessthan>
        </expression>
        <rule>
          <block>
            <rule>
              <update>
                <dynamicfunctioncall>
                  <name>i</name>
                </dynamicfunctioncall>
                <expression>
                  <add>
                    <expression>
                      <dynamicfunctioncall>
                        <name>i</name>
                      </dynamicfunctioncall>
                    </expression>
                    <expression>
                      <literalinteger>1</literalinteger>
                    </expression>
                  </add>
                </expression>
```

```
            </update>
          </rule>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>j</name>
              </dynamicfunctioncall>
              <expression>
                <add>
                  <expression>
                    <dynamicfunctioncall>
                      <name>j</name>
                    </dynamicfunctioncall>
                  </expression>
                  <expression>
                    <literalinteger>1</literalinteger>
                  </expression>
                </add>
              </expression>
            </update>
          </rule>
        </block>
      </rule>
      <rule>
        <stop/>
      </rule>
    </conditional>
  </rule>
</transition>

</module>
```

B.1.2 CountingContext.mas

```
<mas>
  <name>CountingContext</name>
  <agents>
    <agent>
      <name>TheAgent</name>
      <moduleref><name>Counting</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>TheAgent</name>
  </dispatches>
</mas>
```

B.1.3 Fibonacci.mod

```
<module>

  <name>Fibonacci</name>
```

```
<references>
</references>

<imports>
</imports>

<global>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

      <dynamicfunction>
        <name>i</name>
        <type><integer/></type>
      </dynamicfunction>

      <dynamicfunction>
        <name>FibonacciNumbers</name>
        <type>
          <functional>
```

```
            <name>i</name>
            <type><integer/></type>
            <type><integer/></type>
          </functional>
        </type>
      </dynamicfunction>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>

<init>
  <rule>
    <block>
      <rule>
        <update>
          <dynamicfunctioncall>
            <name>i</name>
          </dynamicfunctioncall>
          <expression>
            <literalinteger>2</literalinteger>
          </expression>
        </update>
      </rule>
      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>FibonacciNumbers</name>
                <expression>
                  <literalinteger>0</literalinteger>
                </expression>
              </dynamicfunctioncall>
              <expression>
                <literalinteger>0</literalinteger>
              </expression>
            </update>
          </rule>
```

```
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>FibonacciNumbers</name>
                <expression>
                  <literalinteger>1</literalinteger>
                </expression>
              </dynamicfunctioncall>
              <expression>
                <literalinteger>1</literalinteger>
              </expression>
            </update>
          </rule>
        </block>
      </rule>
    </block>
  </rule>
</init>

<transition>

  <rule>

    <conditional>

      <expression>
        <lessthan>
          <expression>
            <dynamicfunctioncall>
              <name>i</name>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <literalinteger>10000</literalinteger>
          </expression>
        </lessthan>
      </expression>

      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>FibonacciNumbers</name>
                <expression>
                  <dynamicfunctioncall>
                    <name>i</name>
                  </dynamicfunctioncall>
                </expression>
              </dynamicfunctioncall>
              <expression>
                <add>
                  <expression>
```

```
            <dynamicfunctioncall>
              <name>FibonacciNumbers</name>
              <expression>
                <minus>
                  <expression>
                    <dynamicfunctioncall>
                      <name>i</name>
                    </dynamicfunctioncall>
                  </expression>
                  <expression>
                    <literalinteger>1</literalinteger>
                  </expression>
                </minus>
              </expression>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <dynamicfunctioncall>
              <name>FibonacciNumbers</name>
              <expression>
                <minus>
                  <expression>
                    <dynamicfunctioncall>
                      <name>i</name>
                    </dynamicfunctioncall>
                  </expression>
                  <expression>
                    <literalinteger>2</literalinteger>
                  </expression>
                </minus>
              </expression>
            </dynamicfunctioncall>
          </expression>
        </add>
      </expression>
    </update>
  </rule>
  <rule>
    <update>
      <dynamicfunctioncall>
        <name>i</name>
      </dynamicfunctioncall>
      <expression>
        <add>
          <expression>
            <dynamicfunctioncall>
              <name>i</name>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <literalinteger>1</literalinteger>
          </expression>
        </add>
```

```
                </expression>
              </update>
            </rule>
          </block>
        </rule>

        <rule>
          <stop/>
        </rule>

      </conditional>

    </rule>

  </transition>

</module>
```

B.1.4 FibonacciContext.mas

```
<mas>
  <name>FibonacciContext</name>
  <agents>
    <agent>
      <name>TheAgent</name>
      <moduleref><name>Fibonacci</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>TheAgent</name>
  </dispatches>
</mas>
```

B.1.5 SelSort.mod

```
<module>

  <name>SelSort</name>

  <references>
  </references>

  <imports>
    <import>
      <name>Data</name>
      <global>
        <signatures>
          <staticfunctionsignature>
            <name>ItemsSize</name>
            <type><integer/></type>
          </staticfunctionsignature>
          <dynamicfunctionsignature>
            <name>Items</name>
```

```
          <type>
            <functional>
              <name>i</name>
              <type><integer></integer></type>
              <type><integer></integer></type>
            </functional>
          </type>
        </dynamicfunctionsignature>
        <submachinesignature>
          <name>InitData</name>
        </submachinesignature>
        <submachinesignature>
          <name>PrintData</name>
        </submachinesignature>
      </signatures>
    </global>
  </import>
</imports>

<global>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>
```

```
    </staticandderivedfunctions>

    <dynamicfunctions>

      <dynamicfunction>
        <name>i</name>
        <type><integer/></type>
      </dynamicfunction>

      <dynamicfunction>
        <name>j</name>
        <type><integer/></type>
      </dynamicfunction>

      <dynamicfunction>
        <name>min</name>
        <type><integer/></type>
      </dynamicfunction>

      <dynamicfunction>
        <name>step</name>
        <type><integer/></type>
      </dynamicfunction>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>

<init>
  <rule>
    <block>
      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>i</name>
              </dynamicfunctioncall>
              <expression>
```

```
                <literalinteger>0</literalinteger>
              </expression>
            </update>
          </rule>
          <rule>
            <block>
              <rule>
                <update>
                  <dynamicfunctioncall>
                    <name>j</name>
                  </dynamicfunctioncall>
                  <expression>
                    <literalinteger>1</literalinteger>
                  </expression>
                </update>
              </rule>
              <rule>
                <update>
                  <dynamicfunctioncall>
                    <name>step</name>
                  </dynamicfunctioncall>
                  <expression>
                    <literalinteger>0</literalinteger>
                  </expression>
                </update>
              </rule>
            </block>
          </rule>
        </block>
      </rule>

      <rule>
        <block>
          <rule>
            <submachinecall>
              <moduleref><name>Data</name></moduleref>
              <name>InitData</name>
            </submachinecall>
          </rule>
          <rule>
            <submachinecall>
              <moduleref><name>Data</name></moduleref>
              <name>PrintData</name>
            </submachinecall>
          </rule>
        </block>
      </rule>

    </block>
  </rule>
</init>

<transition>
```

```
<rule>

  <conditional>

    <expression>
      <lessthan>
        <expression>
          <dynamicfunctioncall>
            <name>i</name>
          </dynamicfunctioncall>
        </expression>
        <expression>
          <minus>
            <expression>
              <staticfunctioncall>
                <moduleref><name>Data</name></moduleref>
                <name>ItemsSize</name>
              </staticfunctioncall>
            </expression>
            <expression>
              <literalinteger>1</literalinteger>
            </expression>
          </minus>
        </expression>
      </lessthan>
    </expression>

    <rule>

      <conditional>

        <expression>
          <equal>
            <expression>
              <dynamicfunctioncall>
                <name>step</name>
              </dynamicfunctioncall>
            </expression>
            <expression>
              <literalinteger>0</literalinteger>
            </expression>
          </equal>
        </expression>

        <rule>
          <block>
            <rule>
              <update>
                <dynamicfunctioncall>
                  <name>min</name>
                </dynamicfunctioncall>
                <expression>
```

```
            <dynamicfunctioncall>
              <name>i</name>
            </dynamicfunctioncall>
          </expression>
        </update>
      </rule>
      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>j</name>
              </dynamicfunctioncall>
              <expression>
                <add>
                  <expression>
                    <dynamicfunctioncall>
                      <name>i</name>
                    </dynamicfunctioncall>
                  </expression>
                  <expression>
                    <literalinteger>1</literalinteger>
                  </expression>
                </add>
              </expression>
            </update>
          </rule>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>step</name>
              </dynamicfunctioncall>
              <expression>
                <literalinteger>1</literalinteger>
              </expression>
            </update>
          </rule>
        </block>
      </rule>
    </block>
  </rule>

  <rule>

    <conditional>

      <expression>
        <equal>
          <expression>
            <dynamicfunctioncall>
              <name>step</name>
            </dynamicfunctioncall>
          </expression>
```

```
    <expression>
      <literalinteger>1</literalinteger>
    </expression>
  </equal>
</expression>

<rule>

  <conditional>

    <expression>
      <lessthan>
        <expression>
          <dynamicfunctioncall>
            <moduleref><name>Data</name></moduleref>
            <name>Items</name>
            <expression>
              <dynamicfunctioncall>
                <name>j</name>
              </dynamicfunctioncall>
            </expression>
          </dynamicfunctioncall>
        </expression>
        <expression>
          <dynamicfunctioncall>
            <moduleref><name>Data</name></moduleref>
            <name>Items</name>
            <expression>
              <dynamicfunctioncall>
                <name>min</name>
              </dynamicfunctioncall>
            </expression>
          </dynamicfunctioncall>
        </expression>
      </lessthan>
    </expression>

    <rule>
      <block>

        <rule>
          <update>
            <dynamicfunctioncall>
              <name>min</name>
            </dynamicfunctioncall>
            <expression>
              <dynamicfunctioncall>
                <name>j</name>
              </dynamicfunctioncall>
            </expression>
          </update>
        </rule>
```

```
<rule>
  <conditional>

    <expression>
      <equal>
        <expression>
          <dynamicfunctioncall>
            <name>j</name>
          </dynamicfunctioncall>
        </expression>
        <expression>
          <minus>
            <expression>
              <staticfunctioncall>
                <moduleref><name>Data</name></moduleref>
                <name>ItemsSize</name>
              </staticfunctioncall>
            </expression>
            <expression>
              <literalinteger>1</literalinteger>
            </expression>
          </minus>
        </expression>
      </equal>
    </expression>

    <rule>
      <update>
        <dynamicfunctioncall>
          <name>step</name>
        </dynamicfunctioncall>
        <expression>
          <literalinteger>2</literalinteger>
        </expression>
      </update>
    </rule>

    <rule>
      <update>
        <dynamicfunctioncall>
          <name>j</name>
        </dynamicfunctioncall>
        <expression>
          <add>
            <expression>
              <dynamicfunctioncall>
                <name>j</name>
              </dynamicfunctioncall>
            </expression>
            <expression>
              <literalinteger>1</literalinteger>
            </expression>
          </add>
```

```
            </expression>
          </update>
        </rule>

      </conditional>
    </rule>

  </block>
</rule>

<rule>
  <conditional>

    <expression>
      <equal>
        <expression>
          <dynamicfunctioncall>
            <name>j</name>
          </dynamicfunctioncall>
        </expression>
        <expression>
          <minus>
            <expression>
              <staticfunctioncall>
                <moduleref><name>Data</name></moduleref>
                <name>ItemsSize</name>
              </staticfunctioncall>
            </expression>
            <expression>
              <literalinteger>1</literalinteger>
            </expression>
          </minus>
        </expression>
      </equal>
    </expression>

    <rule>
      <update>
        <dynamicfunctioncall>
          <name>step</name>
        </dynamicfunctioncall>
        <expression>
          <literalinteger>2</literalinteger>
        </expression>
      </update>
    </rule>

    <rule>
      <update>
        <dynamicfunctioncall>
          <name>j</name>
        </dynamicfunctioncall>
        <expression>
```

```
              <add>
                <expression>
                  <dynamicfunctioncall>
                    <name>j</name>
                  </dynamicfunctioncall>
                </expression>
                <expression>
                  <literalinteger>1</literalinteger>
                </expression>
              </add>
            </expression>
          </update>
        </rule>

      </conditional>
    </rule>

  </conditional>

</rule>

<rule>

  <block>

    <rule>
      <block>

        <rule>
          <update>
            <dynamicfunctioncall>
              <moduleref><name>Data</name></moduleref>
              <name>Items</name>
              <expression>
                <dynamicfunctioncall>
                  <name>i</name>
                </dynamicfunctioncall>
              </expression>
            </dynamicfunctioncall>
            <expression>
              <dynamicfunctioncall>
                <moduleref><name>Data</name></moduleref>
                <name>Items</name>
                <expression>
                  <dynamicfunctioncall>
                    <name>min</name>
                  </dynamicfunctioncall>
                </expression>
              </dynamicfunctioncall>
            </expression>
          </update>
        </rule>
```

```
      <rule>
        <update>
          <dynamicfunctioncall>
            <moduleref><name>Data</name></moduleref>
            <name>Items</name>
            <expression>
              <dynamicfunctioncall>
                <name>min</name>
              </dynamicfunctioncall>
            </expression>
          </dynamicfunctioncall>
          <expression>
            <dynamicfunctioncall>
              <moduleref><name>Data</name></moduleref>
              <name>Items</name>
              <expression>
                <dynamicfunctioncall>
                  <name>i</name>
                </dynamicfunctioncall>
              </expression>
            </dynamicfunctioncall>
          </expression>
        </update>
      </rule>

    </block>
  </rule>

  <rule>
    <block>

      <rule>
        <update>
          <dynamicfunctioncall>
            <name>i</name>
          </dynamicfunctioncall>
          <expression>
            <add>
              <expression>
                <dynamicfunctioncall>
                  <name>i</name>
                </dynamicfunctioncall>
              </expression>
              <expression>
                <literalinteger>1</literalinteger>
              </expression>
            </add>
          </expression>
        </update>
      </rule>

      <rule>
        <update>
```

```
                        <dynamicfunctioncall>
                          <name>step</name>
                        </dynamicfunctioncall>
                        <expression>
                          <literalinteger>0</literalinteger>
                        </expression>
                      </update>
                    </rule>

                  </block>
                </rule>
              </block>

            </rule>

          </conditional>

        </rule>

      </conditional>

    </rule>

    <rule>
      <block>
        <rule>
          <submachinecall>
            <moduleref><name>Data</name></moduleref>
            <name>PrintData</name>
          </submachinecall>
        </rule>
        <rule>
          <stop/>
        </rule>
      </block>
    </rule>

  </conditional>

</rule>

</transition>

</module>
```

B.1.6 Data.mod

```
<module>

  <name>Data</name>

  <references>
  </references>
```

```
<imports>
  <import>
    <name>Output</name>
    <global>
      <signatures>
        <actionsignature>
          <name>Write</name>
          <name>s</name>
          <type><string/></type>
        </actionsignature>
      </signatures>
    </global>
  </import>
  <import>
    <name>StringManipulation</name>
    <global>
      <signatures>
        <externalfunctionsignature>
          <name>IntegerToString</name>
          <type>
            <functional>
              <name>i</name>
              <type><integer/></type>
              <type><string/></type>
            </functional>
          </type>
        </externalfunctionsignature>
        <externalfunctionsignature>
          <name>Concat</name>
          <type>
            <functional>
              <name>s</name>
              <type><string/></type>
              <type>
                <functional>
                  <name>r</name>
                  <type><string/></type>
                  <type><string/></type>
                </functional>
              </type>
            </functional>
            </type>
        </externalfunctionsignature>
      </signatures>
    </global>
  </import>
</imports>

<global>

  <environment>
```

```
<staticandderivedfunctions>

  <staticfunction>
    <name>ItemsSize</name>
    <type><integer/></type>
    <expression>
      <literalinteger>1000</literalinteger>
    </expression>
  </staticfunction>

</staticandderivedfunctions>

<dynamicfunctions>

  <dynamicfunction>
    <name>Items</name>
    <type>
      <functional>
        <name>i</name>
        <type><integer></integer></type>
        <type><integer></integer></type>
      </functional>
    </type>
  </dynamicfunction>

  <dynamicfunction>
    <name>i</name>
    <type><integer></integer></type>
  </dynamicfunction>

  <dynamicfunction>
    <name>ret</name>
    <type><integer></integer></type>
  </dynamicfunction>

</dynamicfunctions>

<externalfunctions>

  <externalfunction>
    <name>RandomInteger</name>
    <type><integer></integer></type>
  </externalfunction>

</externalfunctions>

<actions>

</actions>

<submachines>

  <submachine>
```

```
<name>PrintData</name>

<init>
  <rule>
    <update>
      <dynamicfunctioncall>
        <name>i</name>
      </dynamicfunctioncall>
      <expression>
        <literalinteger>0</literalinteger>
      </expression>
    </update>
  </rule>
</init>

<rule>
  <conditional>

      <expression>
        <lessthan>
          <expression>
            <dynamicfunctioncall>
              <name>i</name>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <staticfunctioncall>
              <name>ItemsSize</name>
            </staticfunctioncall>
          </expression>
        </lessthan>
      </expression>

      <rule>
        <block>

          <rule>
            <update>
              <dynamicfunctioncall>
                <name>i</name>
              </dynamicfunctioncall>
              <expression>
                <add>
                  <expression>
                    <dynamicfunctioncall>
                      <name>i</name>
                    </dynamicfunctioncall>
                  </expression>
                  <expression>
                    <literalinteger>1</literalinteger>
                  </expression>
                </add>
              </expression>
```

```
    </update>
  </rule>

  <rule>
    <actioncall>
      <moduleref><name>Output</name></moduleref>
      <name>Write</name>
      <expression>
        <externalfunctioncall>
          <moduleref><name>StringManipulation</name></moduleref>
          <name>Concat</name>
          <expression>
            <externalfunctioncall>
              <moduleref><name>StringManipulation</name></moduleref>
              <name>IntegerToString</name>
              <expression>
                <dynamicfunctioncall>
                  <name>Items</name>
                  <expression>
                    <dynamicfunctioncall>
                      <name>i</name>
                    </dynamicfunctioncall>
                  </expression>
                </dynamicfunctioncall>
              </expression>
            </externalfunctioncall>
          </expression>
          <expression>
            <ifexp>
              <expression>
                <equal>
                  <expression>
                    <mod>
                      <expression>
                        <dynamicfunctioncall>
                          <name>i</name>
                        </dynamicfunctioncall>
                      </expression>
                      <expression>
                        <literalinteger>10</literalinteger>
                      </expression>
                    </mod>
                  </expression>
                  <expression>
                    <literalinteger>9</literalinteger>
                  </expression>
                </equal>
              </expression>
              <expression>
                <literalstring>"\n"</literalstring>
              </expression>
              <expression>
                <literalstring>"\t"</literalstring>
```

```
                        </expression>
                      </ifexp>
                    </expression>
                  </externalfunctioncall>
                </expression>
              </actioncall>
            </rule>

          </block>
        </rule>

        <rule>
          <block>
            <rule>
              <return/>
            </rule>
            <rule>
              <actioncall>
                <moduleref><name>Output</name></moduleref>
                <name>Write</name>
                <expression>
                  <literalstring>"\n\n"</literalstring>
                </expression>
              </actioncall>
            </rule>
          </block>
        </rule>

    </conditional>

  </rule>
</submachine>

<submachine>
  <name>InitData</name>
  <rule>

    <conditional>

        <expression>
          <lessthan>
            <expression>
              <dynamicfunctioncall>
                <name>i</name>
              </dynamicfunctioncall>
            </expression>
            <expression>
              <staticfunctioncall>
                <name>ItemsSize</name>
              </staticfunctioncall>
            </expression>
          </lessthan>
        </expression>
```

```
            <rule>
              <block>

                <rule>
                  <update>
                    <dynamicfunctioncall>
                      <name>i</name>
                    </dynamicfunctioncall>
                    <expression>
                      <add>
                        <expression>
                          <dynamicfunctioncall>
                            <name>i</name>
                          </dynamicfunctioncall>
                        </expression>
                        <expression>
                          <literalinteger>1</literalinteger>
                        </expression>
                      </add>
                    </expression>
                  </update>
                </rule>

                <rule>
                  <update>
                    <dynamicfunctioncall>
                      <name>Items</name>
                      <expression>
                        <dynamicfunctioncall>
                          <name>i</name>
                        </dynamicfunctioncall>
                      </expression>
                    </dynamicfunctioncall>
                    <expression>
                      <externalfunctioncall>
                        <name>RandomInteger</name>
                      </externalfunctioncall>
                    </expression>
                  </update>
                </rule>

              </block>
            </rule>

            <rule><return/></rule>

          </conditional>

        </rule>
      </submachine>

    </submachines>
```

```
    </environment>

  </global>

  <local>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>

      <dynamicfunctions>

      </dynamicfunctions>

      <externalfunctions>

      </externalfunctions>

      <actions>

      </actions>

      <submachines>

      </submachines>

    </environment>

  </local>

  <init>
    <rule>
      <update>
        <dynamicfunctioncall>
          <name>i</name>
        </dynamicfunctioncall>
        <expression>
          <literalinteger>0</literalinteger>
        </expression>
      </update>
    </rule>
  </init>

  <transition>
    <rule><lambda/></rule>
  </transition>

</module>
```

B.1.7 Output.mod

```
<module>

  <name>Output</name>

  <references>
  </references>

  <imports>
  </imports>

  <global>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>

      <dynamicfunctions>

        <dynamicfunction>
          <name>success</name>
          <type><integer></integer></type>
        </dynamicfunction>

      </dynamicfunctions>

      <externalfunctions>

        <externalfunction>
          <name>WriteExtFunc</name>
          <type>
            <functional>
              <name>s</name>
              <type><string/></type>
              <type><integer></integer></type>
            </functional>
          </type>
        </externalfunction>

      </externalfunctions>

      <actions>

        <action>
          <name>Write</name>
          <name>s</name>
          <type><string/></type>
          <rule>
            <immediateupdate>
              <dynamicfunctioncall>
```

```
              <name>success</name>
            </dynamicfunctioncall>
            <expression>
              <externalfunctioncall>
                <name>WriteExtFunc</name>
                <expression>
                  <parametercall>
                    <name>s</name>
                  </parametercall>
                </expression>
              </externalfunctioncall>
            </expression>
          </immediateupdate>
        </rule>
      </action>

    </actions>

    <submachines>

    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>
```

```
  <init>
    <rule>
      <lambda>
      </lambda>
    </rule>
  </init>
  <transition>
    <rule>
      <lambda>
      </lambda>
    </rule>
  </transition>
</module>
```

B.1.8 StringManipulation.mod

```
<module>
  <name>StringManipulation</name>
  <references>
  </references>
  <imports>
  </imports>
  <global>
    <environment>
      <staticandderivedfunctions>
      </staticandderivedfunctions>
      <dynamicfunctions>
      </dynamicfunctions>
      <externalfunctions>
```

```
  <externalfunction>
    <name>IntegerToString</name>
    <type>
      <functional>
        <name>i</name>
        <type><integer/></type>
        <type><string/></type>
      </functional>
    </type>
  </externalfunction>

  <externalfunction>
    <name>CharToString</name>
    <type>
      <functional>
        <name>i</name>
        <type><character/></type>
        <type><string/></type>
      </functional>
    </type>
  </externalfunction>

  <externalfunction>
    <name>RealToString</name>
    <type>
      <functional>
        <name>i</name>
        <type><real/></type>
        <type><string/></type>
      </functional>
    </type>
  </externalfunction>

  <externalfunction>
    <name>Concat</name>
    <type>
      <functional>
        <name>s</name>
        <type><string/></type>
        <type>
          <functional>
            <name>r</name>
            <type><string/></type>
            <type><string/></type>
          </functional>
        </type>
      </functional>
    </type>
  </externalfunction>

</externalfunctions>

<actions>
```

```
      </actions>
      <submachines>
      </submachines>
    </environment>
  </global>
  <local>
    <environment>
      <staticandderivedfunctions>
      </staticandderivedfunctions>
      <dynamicfunctions>
      </dynamicfunctions>
      <externalfunctions>
      </externalfunctions>
      <actions>
      </actions>
      <submachines>
      </submachines>
    </environment>
  </local>
  <init>
    <rule>
      <lambda>
      </lambda>
    </rule>
  </init>
  <transition>
```

```
    <rule>

      <lambda>

      </lambda>

    </rule>

  </transition>

</module>
```

B.1.9 SelSortContext.mas

```
<mas>
  <name>SelSortContext</name>
  <agents>
    <agent>
      <name>TheAgent</name>
      <moduleref><name>SelSort</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>TheAgent</name>
  </dispatches>
</mas>
```

B.1.10 Raffle.mod

```
<module>

  <name>Raffle</name>

  <references>
  </references>

  <imports>
    <import>
      <name>Output</name>
      <global>
        <signatures>
          <actionsignature>
            <name>Write</name>
            <name>s</name>
            <type><string/></type>
          </actionsignature>
        </signatures>
      </global>
    </import>
    <import>
      <name>StringManipulation</name>
      <global>
        <signatures>
```

```
        <externalfunctionsignature>
          <name>IntegerToString</name>
          <type>
            <functional>
              <name>i</name>
              <type><integer/></type>
              <type><string/></type>
            </functional>
          </type>
        </externalfunctionsignature>
        <externalfunctionsignature>
          <name>Concat</name>
          <type>
            <functional>
              <name>s</name>
              <type><string/></type>
              <type>
                <functional>
                  <name>r</name>
                  <type><string/></type>
                  <type><string/></type>
                </functional>
              </type>
            </functional>
          </type>
        </externalfunctionsignature>
      </signatures>
    </global>
  </import>
</imports>

<global>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>
```

```
    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

      <dynamicfunction>
        <name>Beatles</name>
        <type>
          <set>
            <type>
              <tuple>
                <name>surname</name><type><string/></type>
                <name>name</name><type><string/></type>
                <name>alive</name><type><boolean/></type>
              </tuple>
            </type>
          </set>
        </type>
      </dynamicfunction>

      <dynamicfunction>
        <name>step</name>
        <type><integer/></type>
      </dynamicfunction>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>
```

```
<init>

  <rule>

    <block>

      <rule>
        <update>
          <dynamicfunctioncall>
            <name>step</name>
          </dynamicfunctioncall>
          <expression>
            <literalinteger>0</literalinteger>
          </expression>
        </update>
      </rule>


      <rule>
        <update>
          <dynamicfunctioncall>
            <name>Beatles</name>
          </dynamicfunctioncall>
          <expression>
            <setagregate>
              <expression>
                <tupleagregate>
                  <expression><literalstring>"McCartney"</literalstring></expression>
                  <expression><literalstring>"Paul"</literalstring></expression>
                  <expression><literalboolean>true</literalboolean></expression>
                </tupleagregate>
              </expression>
              <expression>
                <tupleagregate>
                  <expression><literalstring>"Lennon"</literalstring></expression>
                  <expression><literalstring>"John"</literalstring></expression>
                  <expression><literalboolean>false</literalboolean></expression>
                </tupleagregate>
              </expression>
              <expression>
                <tupleagregate>
                  <expression><literalstring>"Harrison"</literalstring></expression>
                  <expression><literalstring>"George"</literalstring></expression>
                  <expression><literalboolean>false</literalboolean></expression>
                </tupleagregate>
              </expression>
              <expression>
                <tupleagregate>
                  <expression><literalstring>"Star"</literalstring></expression>
                  <expression><literalstring>"Ringo"</literalstring></expression>
                  <expression><literalboolean>true</literalboolean></expression>
                </tupleagregate>
```

```
              </expression>
            </setagregate>
          </expression>
        </update>
      </rule>

    </block>

  </rule>

</init>

<transition>

  <rule>

    <conditional>
      <expression>
        <equal>
          <expression>
            <dynamicfunctioncall>
              <name>step</name>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <literalinteger>0</literalinteger>
          </expression>
        </equal>
      </expression>
      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>step</name>
              </dynamicfunctioncall>
              <expression>
                <literalinteger>1</literalinteger>
              </expression>
            </update>
          </rule>
          <rule>
            <actioncall>
              <moduleref><name>Output</name></moduleref>
              <name>Write</name>
              <expression>
                <literalstring>"The Beatles are:\n"</literalstring>
              </expression>
            </actioncall>
          </rule>
        </block>
      </rule>
      <rule>
```

```
<conditional>
  <expression>
    <equal>
      <expression>
        <dynamicfunctioncall>
          <name>step</name>
        </dynamicfunctioncall>
      </expression>
      <expression>
        <literalinteger>1</literalinteger>
      </expression>
    </equal>
  </expression>
  <rule>
    <block>
      <rule>
        <forall>
          <name>beatle</name>
          <expression>
            <dynamicfunctioncall>
              <name>Beatles</name>
            </dynamicfunctioncall>
          </expression>
          <rule>
            <actioncall>
              <moduleref><name>Output</name></moduleref>
              <name>Write</name>
              <expression>
                <externalfunctioncall>
                  <moduleref><name>StringManipulation</name></moduleref>
                  <name>Concat</name>
                  <expression>
                    <tupleprojection>
                      <expression>
                        <forallalias><name>beatle</name></forallalias>
                      </expression>
                      <literalinteger>1</literalinteger>
                    </tupleprojection>
                  </expression>
                  <expression>
                    <externalfunctioncall>
                      <moduleref><name>StringManipulation</name></moduleref>
                      <name>Concat</name>
                      <expression>
                        <literalstring>" "</literalstring>
                      </expression>
                      <expression>
                        <externalfunctioncall>
                          <moduleref><name>StringManipulation</name></moduleref>
                          <name>Concat</name>
                          <expression>
                            <tupleprojection>
                              <expression>
```

```
                                    <forallalias><name>beatle</name></forallalias>
                                  </expression>
                                  <literalinteger>0</literalinteger>
                                </tupleprojection>
                              </expression>
                              <expression>
                                <literalstring>"\n"</literalstring>
                              </expression>
                            </externalfunctioncall>
                          </expression>
                        </externalfunctioncall>
                      </expression>
                    </externalfunctioncall>
                  </expression>
                </actioncall>
              </rule>
            </forall>
          </rule>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>step</name>
              </dynamicfunctioncall>
              <expression>
                <literalinteger>2</literalinteger>
              </expression>
            </update>
          </rule>
        </block>
      </rule>
      <rule>
        <conditional>
          <expression>
            <equal>
              <expression>
                <dynamicfunctioncall>
                  <name>step</name>
                </dynamicfunctioncall>
              </expression>
              <expression>
                <literalinteger>2</literalinteger>
              </expression>
            </equal>
          </expression>
          <rule>
            <block>
              <rule>
                <update>
                  <dynamicfunctioncall>
                    <name>step</name>
                  </dynamicfunctioncall>
                  <expression>
                    <literalinteger>3</literalinteger>
```

```
        </expression>
      </update>
    </rule>
    <rule>
      <actioncall>
        <moduleref><name>Output</name></moduleref>
        <name>Write</name>
        <expression>
          <literalstring>"And the choosen (alive) Beatle is:\n"</literalstring>
        </expression>
      </actioncall>
    </rule>
  </block>
</rule>
<rule>
  <conditional>
    <expression>
      <equal>
        <expression>
          <dynamicfunctioncall>
            <name>step</name>
          </dynamicfunctioncall>
        </expression>
        <expression>
          <literalinteger>3</literalinteger>
        </expression>
      </equal>
    </expression>
    <rule>
      <block>
        <rule>
          <update>
            <dynamicfunctioncall>
              <name>step</name>
            </dynamicfunctioncall>
            <expression>
              <literalinteger>4</literalinteger>
            </expression>
          </update>
        </rule>
        <rule>
          <choose>
            <name>beatle</name>
            <expression>
              <dynamicfunctioncall>
                <name>Beatles</name>
              </dynamicfunctioncall>
            </expression>
            <expression>
              <tupleprojection>
                <expression>
                  <choosealias>
                    <name>beatle</name>
```

```
            </choosealias>
          </expression>
          <literalinteger>2</literalinteger>
        </tupleprojection>
      </expression>
      <rule>
        <actioncall>
          <moduleref><name>Output</name></moduleref>
          <name>Write</name>
          <expression>
            <externalfunctioncall>
              <moduleref><name>StringManipulation</name></moduleref>
              <name>Concat</name>
              <expression>
                <tupleprojection>
                  <expression>
                    <choosealias><name>beatle</name></choosealias>
                  </expression>
                  <literalinteger>1</literalinteger>
                </tupleprojection>
              </expression>
              <expression>
                <externalfunctioncall>
                  <moduleref><name>StringManipulation</name></moduleref>
                  <name>Concat</name>
                  <expression>
                    <literalstring>" "</literalstring>
                  </expression>
                  <expression>
                    <externalfunctioncall>
                      <moduleref><name>StringManipulation</name></moduleref>
                      <name>Concat</name>
                      <expression>
                        <tupleprojection>
                          <expression>
                            <choosealias><name>beatle</name></choosealias>
                          </expression>
                          <literalinteger>0</literalinteger>
                        </tupleprojection>
                      </expression>
                      <expression>
                        <literalstring>"\n"</literalstring>
                      </expression>
                    </externalfunctioncall>
                  </expression>
                </externalfunctioncall>
              </expression>
            </externalfunctioncall>
          </expression>
        </actioncall>
      </rule>
    </choose>
  </rule>
```

```
                    </block>
                  </rule>
                  <rule>
                    <stop/>




                  </rule>
                </conditional>
              </rule>
            </conditional>

          </rule>
        </conditional>
      </rule>



    </conditional>

  </rule>

</transition>

</module>
```

B.1.11 RaffleContext.mas

```
<mas>
  <name>RaffleContext</name>
  <agents>
    <agent>
      <name>MyAgent</name>
      <moduleref><name>Raffle</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>MyAgent</name>
  </dispatches>
</mas>
```

B.1.12 Math.mod

```
<module>

  <name>Math</name>

  <references>
  </references>

  <imports>
    <import>
```

```
  <name>Functions</name>
  <global>
    <signatures>
      <staticfunctionsignature>
        <name>Pi</name>
        <type><real/></type>
      </staticfunctionsignature>
      <staticfunctionsignature>
        <name>Sin</name>
        <type>
          <functional>
            <name>d</name>
            <type><real/></type>
            <type><real/></type>
          </functional>
        </type>
      </staticfunctionsignature>
    </signatures>
  </global>
</import>
<import>
  <name>Output</name>
  <global>
    <signatures>
      <actionsignature>
        <name>Write</name>
        <name>s</name>
        <type><string/></type>
      </actionsignature>
    </signatures>
  </global>
</import>
<import>
  <name>StringManipulation</name>
  <global>
    <signatures>
      <externalfunctionsignature>
        <name>RealToString</name>
        <type>
          <functional>
            <name>i</name>
            <type><real/></type>
            <type><string/></type>
          </functional>
        </type>
      </externalfunctionsignature>
      <externalfunctionsignature>
        <name>Concat</name>
        <type>
          <functional>
            <name>s</name>
            <type><string/></type>
            <type>
```

```
                <functional>
                  <name>r</name>
                  <type><string/></type>
                  <type><string/></type>
                </functional>
              </type>
            </functional>
          </type>
        </externalfunctionsignature>
      </signatures>
    </global>
  </import>

</imports>

<global>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>
```

```
        <dynamicfunction>
          <name>Grads</name>
          <type>
            <set>
              <type>
                <real/>
              </type>
            </set>
          </type>
        </dynamicfunction>

      </dynamicfunctions>

      <externalfunctions>

      </externalfunctions>

      <actions>

      </actions>

      <submachines>

      </submachines>

    </environment>

  </local>

  <init>

    <rule>

      <let>
        <name>pi</name>
        <expression>
          <staticfunctioncall>
            <moduleref><name>Functions</name></moduleref>
            <name>Pi</name>
          </staticfunctioncall>
        </expression>
        <rule>

      <update>
        <dynamicfunctioncall>
          <name>Grads</name>
        </dynamicfunctioncall>
        <expression>
          <setagregate>
            <expression>
              <div>
                <expression>
                  <letalias>
```

```
        <name>pi</name>
      </letalias>
    </expression>
    <expression>
      <literalreal>5.0</literalreal>
    </expression>
  </div>
</expression>
<expression>
  <div>
    <expression>
      <letalias>
        <name>pi</name>
      </letalias>
    </expression>
    <expression>
      <literalreal>4.0</literalreal>
    </expression>
  </div>
</expression>
<expression>
  <div>
    <expression>
      <letalias>
        <name>pi</name>
      </letalias>
    </expression>
    <expression>
      <literalreal>3.0</literalreal>
    </expression>
  </div>
</expression>
<expression>
  <div>
    <expression>
      <letalias>
        <name>pi</name>
      </letalias>
    </expression>
    <expression>
      <literalreal>2.0</literalreal>
    </expression>
  </div>
</expression>
<expression>
  <div>
    <expression>
      <letalias>
        <name>pi</name>
      </letalias>
    </expression>
    <expression>
      <literalreal>1.0</literalreal>
```

```
              </expression>
            </div>
          </expression>
        </setagregate>
      </expression>
    </update>

      </rule>
    </let>

  </rule>

</init>

<transition>

  <rule>

    <block>
      <rule>
        <stop/>
      </rule>
      <rule>
        <forall>
          <name>d</name>
          <expression>
            <dynamicfunctioncall>
              <name>Grads</name>
            </dynamicfunctioncall>
          </expression>
          <rule>
            <actioncall>
              <moduleref><name>Output</name></moduleref>
              <name>Write</name>
              <expression>
                <staticfunctioncall>
                  <moduleref><name>StringManipulation</name></moduleref>
                  <name>Concat</name>
                  <expression>
                    <staticfunctioncall>
                      <moduleref><name>StringManipulation</name></moduleref>
                      <name>Concat</name>
                      <expression>
                        <literalstring>"Sin("</literalstring>
                      </expression>
                      <expression>
                        <staticfunctioncall>
                          <moduleref><name>StringManipulation</name></moduleref>
                          <name>Concat</name>
                          <expression>
                            <staticfunctioncall>
                              <moduleref><name>StringManipulation</name></moduleref>
                              <name>RealToString</name>
```

```
                        <expression>
                          <forallalias>
                            <name>d</name>
                          </forallalias>
                        </expression>
                      </staticfunctioncall>
                    </expression>
                    <expression>
                      <literalstring>") \t= \t"</literalstring>
                    </expression>
                  </staticfunctioncall>
                </expression>
              </staticfunctioncall>
            </expression>
            <expression>
              <staticfunctioncall>
                <moduleref><name>StringManipulation</name></moduleref>
                <name>Concat</name>
                <expression>
                  <externalfunctioncall>
                    <moduleref><name>StringManipulation</name></moduleref>
                    <name>RealToString</name>
                    <expression>
                      <staticfunctioncall>
                        <moduleref><name>Functions</name></moduleref>
                        <name>Sin</name>
                        <expression>
                          <forallalias>
                            <name>d</name>
                          </forallalias>
                        </expression>
                      </staticfunctioncall>
                    </expression>
                  </externalfunctioncall>
                </expression>
                <expression>
                  <literalstring>"\n"</literalstring>
                </expression>
              </staticfunctioncall>
            </expression>
          </staticfunctioncall>
        </expression>
      </actioncall>
    </rule>
  </forall>
 </rule>
</block>

</rule>

</transition>

</module>
```

B.1.13 Functions.mod

```
<module>

  <name>Functions</name>

  <references>
  </references>

  <imports>
  </imports>

  <global>

    <environment>

      <staticandderivedfunctions>

        <staticfunction>
          <name>Pi</name>
          <type><real/></type>
          <expression><literalreal>3.1415926535898</literalreal></expression>
        </staticfunction>

        <staticfunction>
          <name>Fatorial</name>
          <type>
            <functional>
              <name>d</name>
              <type>
                <real/>
              </type>
              <type>
                <real/>
              </type>
            </functional>
          </type>
          <expression>
            <ifexp>
              <expression>
                <not>
                  <expression>
                    <morethan>
                      <expression>
                        <parametercall>
                          <name>d</name>
                        </parametercall>
                      </expression>
                      <expression>
                        <literalreal>0.0</literalreal>
                      </expression>
                    </morethan>
                  </expression>
```

```
          </not>
        </expression>
        <expression>
          <literalreal>1.0</literalreal>
        </expression>
        <expression>
          <mult>
            <expression>
              <staticfunctioncall>
                <name>Fatorial</name>
                <expression>
                  <minus>
                    <expression>
                      <parametercall>
                        <name>d</name>
                      </parametercall>
                    </expression>
                    <expression>
                      <literalreal>1.0</literalreal>
                    </expression>
                  </minus>
                </expression>
              </staticfunctioncall>
            </expression>
            <expression>
              <parametercall>
                <name>d</name>
              </parametercall>
            </expression>
          </mult>
        </expression>
      </ifexp>
    </expression>
  </staticfunction>

  <staticfunction>
    <name>Sin</name>
    <type>
      <functional>
        <name>d</name>
        <type>
          <real/>
        </type>
        <type>
          <real/>
        </type>
      </functional>
    </type>
    <expression>
      <staticfunctioncall>
        <name>SinAux</name>
        <expression>
          <parametercall>
```

```
            <name>d</name>
          </parametercall>
        </expression>
        <expression>
          <literalinteger>100</literalinteger>
        </expression>
      </staticfunctioncall>
    </expression>
  </staticfunction>

  <derivedfunction>
    <name>SinAux</name>
    <type>
      <functional>
        <name>d</name>
        <type>
          <real/>
        </type>
        <type>
          <functional>
            <name>i</name>
            <type>
              <integer/>
            </type>
            <type>
              <real/>
            </type>
          </functional>
        </type>
      </functional>
    </type>
    <expression>
      <ifexp>
        <expression>
          <lessequalthan>
            <expression>
              <parametercall>
                <name>i</name>
              </parametercall>
            </expression>
            <expression>
              <literalinteger>-1</literalinteger>
            </expression>
          </lessequalthan>
        </expression>
        <expression>
          <literalreal>0.0</literalreal>
        </expression>
        <expression>
          <add>
            <expression>
              <staticfunctioncall>
                <name>SinAux</name>
```

```
    <expression>
      <parametercall>
        <name>d</name>
      </parametercall>
    </expression>
    <expression>
      <minus>
        <expression>
          <parametercall>
            <name>i</name>
          </parametercall>
        </expression>
        <expression>
          <literalinteger>1</literalinteger>
        </expression>
      </minus>
    </expression>
   </staticfunctioncall>
</expression>
<expression>
  <mult>
    <expression>
      <externalfunctioncall>
        <name>IntegerToReal</name>
        <expression>
          <staticfunctioncall>
            <name>Parity</name>
            <expression>
              <parametercall>
                <name>i</name>
              </parametercall>
            </expression>
          </staticfunctioncall>
        </expression>
      </externalfunctioncall>
    </expression>
    <expression>
      <letexp>
        <name>n</name>
        <expression>
          <externalfunctioncall>
            <name>IntegerToReal</name>
            <expression>
              <add>
                <expression>
                  <mult>
                    <expression>
                      <literalinteger>2</literalinteger>
                    </expression>
                    <expression>
                      <parametercall>
                        <name>i</name>
                      </parametercall>
```

```
                                    </expression>
                                  </mult>
                                </expression>
                                <expression>
                                  <literalinteger>1</literalinteger>
                                </expression>
                              </add>
                            </expression>
                          </externalfunctioncall>
                        </expression>
                        <expression>
                          <div>
                            <expression>
                              <staticfunctioncall>
                                <name>Power</name>
                                <expression>
                                  <parametercall>
                                    <name>d</name>
                                  </parametercall>
                                </expression>
                                <expression>
                                  <letexpalias>
                                    <name>n</name>
                                  </letexpalias>
                                </expression>
                              </staticfunctioncall>
                            </expression>
                            <expression>
                              <staticfunctioncall>
                                <name>Fatorial</name>
                                <expression>
                                  <letexpalias>
                                    <name>n</name>
                                  </letexpalias>
                                </expression>
                              </staticfunctioncall>
                            </expression>
                          </div>
                        </expression>
                      </letexp>
                    </expression>
                  </mult>
                </expression>
              </add>
            </expression>
          </ifexp>
        </expression>
      </derivedfunction>

      <staticfunction>
        <name>Power</name>
        <type>
          <functional>
```

```
    <name>a</name>
    <type>
      <real/>
    </type>
    <type>
      <functional>
        <name>b</name>
        <type>
          <real/>
        </type>
        <type>
          <real/>
        </type>
      </functional>
    </type>
  </functional>
</type>
<expression>
  <ifexp>
    <expression>
      <not>
        <expression>
          <moreequalthan>
            <expression>
              <parametercall>
                <name>b</name>
              </parametercall>
            </expression>
            <expression>
              <literalreal>1.0</literalreal>
            </expression>
          </moreequalthan>
        </expression>
      </not>
    </expression>
    <expression>
      <literalreal>1.0</literalreal>
    </expression>
    <expression>
      <mult>
        <expression>
          <parametercall>
            <name>a</name>
          </parametercall>
        </expression>
        <expression>
          <staticfunctioncall>
            <name>Power</name>
            <expression>
              <parametercall>
                <name>a</name>
              </parametercall>
            </expression>
```

```
                <expression>
                  <minus>
                    <expression>
                      <parametercall>
                        <name>b</name>
                      </parametercall>
                    </expression>
                    <expression>
                      <literalreal>1.0</literalreal>
                    </expression>
                  </minus>
                </expression>
              </staticfunctioncall>
            </expression>
          </mult>
        </expression>
      </ifexp>
    </expression>
  </staticfunction>

  <staticfunction>
    <name>Parity</name>
    <type>
      <functional>
        <name>i</name>
        <type>
          <integer/>
        </type>
        <type>
          <integer/>
        </type>
      </functional>
    </type>
    <expression>
      <ifexp>
        <expression>
          <diff>
            <expression>
              <mod>
                <expression>
                  <parametercall>
                    <name>i</name>
                  </parametercall>
                </expression>
                <expression>
                  <literalinteger>2</literalinteger>
                </expression>
              </mod>
            </expression>
            <expression>
              <literalinteger>0</literalinteger>
            </expression>
          </diff>
```

```
            </expression>
            <expression>
              <literalinteger>-1</literalinteger>
            </expression>
            <expression>
              <literalinteger>1</literalinteger>
            </expression>
          </ifexp>
        </expression>
      </staticfunction>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>
      <externalfunction>
        <name>IntegerToReal</name>
        <type>
          <functional>
            <name>i</name>
            <type><integer/></type>
            <type><real/></type>
          </functional>
        </type>
      </externalfunction>
    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>
```

```
      <externalfunctions>
      </externalfunctions>
      <actions>
      </actions>
      <submachines>
      </submachines>
    </environment>
  </local>
  <init>
    <rule>
      <lambda>
      </lambda>
    </rule>
  </init>
  <transition>
    <rule>
      <lambda>
      </lambda>
    </rule>
  </transition>
</module>
```

B.1.14 MathContext.mas

```
<mas>
  <name>MathContext</name>
  <agents>
    <agent>
      <name>MyAgent</name>
      <moduleref><name>Math</name></moduleref>
    </agent>
  </agents>
```

```
  <dispatches>
    <name>MyAgent</name>
  </dispatches>
</mas>
```

B.1.15 TypeUnion.mod

```
<module>

  <name>TypeUnion</name>

  <references>
  </references>

  <imports>
    <import>
      <name>Data</name>
      <global>
        <signatures>
          <externalfunctionsignature>
            <name>RandomInteger</name>
            <type><integer></integer></type>
          </externalfunctionsignature>
        </signatures>
      </global>
    </import>
    <import>
      <name>Output</name>
      <global>
        <signatures>
          <actionsignature>
            <name>Write</name>
            <name>s</name>
            <type><string/></type>
          </actionsignature>
        </signatures>
      </global>
    </import>
    <import>
      <name>StringManipulation</name>
      <global>
        <signatures>
          <externalfunctionsignature>
            <name>IntegerToString</name>
            <type>
              <functional>
                <name>i</name>
                <type><integer/></type>
                <type><string/></type>
              </functional>
            </type>
          </externalfunctionsignature>
          <externalfunctionsignature>
```

```
          <name>CharToString</name>
          <type>
            <functional>
              <name>i</name>
              <type><character/></type>
              <type><string/></type>
            </functional>
          </type>
        </externalfunctionsignature>
        <externalfunctionsignature>
          <name>RealToString</name>
          <type>
            <functional>
              <name>r</name>
              <type><real/></type>
              <type><string/></type>
            </functional>
          </type>
        </externalfunctionsignature>
        <externalfunctionsignature>
          <name>Concat</name>
          <type>
            <functional>
              <name>s</name>
              <type><string/></type>
              <type>
                <functional>
                  <name>r</name>
                  <type><string/></type>
                  <type><string/></type>
                </functional>
              </type>
            </functional>
          </type>
        </externalfunctionsignature>
      </signatures>
    </global>
  </import>
</imports>

<global>

  <environment>

    <staticandderivedfunctions>

      <derivedfunction>
        <name>RandomMax</name>
        <type>
          <functional>
            <name>max</name>
            <type><integer/></type>
            <type><integer/></type>
```

```
      </functional>
    </type>
    <expression>
      <div>
        <expression>
          <mult>
            <expression>
              <externalfunctioncall>
                <moduleref><name>Data</name></moduleref>
                <name>RandomInteger</name>
              </externalfunctioncall>
            </expression>
            <expression>
              <parametercall>
                <name>max</name>
              </parametercall>
            </expression>
          </mult>
        </expression>
        <expression>
          <literalinteger>100</literalinteger>
        </expression>
      </div>
    </expression>
  </derivedfunction>

</staticandderivedfunctions>

<dynamicfunctions>

  <dynamicfunction>
    <name>Interval</name>
    <type>
      <interval>
        <expression>
          <literalinteger>0</literalinteger>
        </expression>
        <expression>
          <literalinteger>1000</literalinteger>
        </expression>
      </interval>
    </type>
  </dynamicfunction>

  <dynamicfunction>
    <name>SetWhereToSearch</name>
    <type>
      <set>
        <type>
          <integer/>
        </type>
      </set>
    </type>
```

```
      </dynamicfunction>

      <dynamicfunction>
        <name>OfAnyType</name>
        <type>
          <any/>
        </type>
      </dynamicfunction>

      <dynamicfunction>
        <name>Items</name>
        <type>
          <list>
            <type>
              <disjointunion>
                <type>
                  <boolean/>
                </type>
                <type>
                  <character/>
                </type>
                <type>
                  <integer/>
                </type>
                <type>
                  <real/>
                </type>
                <type>
                  <string/>
                </type>
              </disjointunion>
            </type>
          </list>
        </type>
      </dynamicfunction>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</global>
```

```
<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>

<init>
  <rule>
    <block>
      <rule>
        <lambda/>
      </rule>
      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>Items</name>
              </dynamicfunctioncall>
              <expression>
                <listagregate>
                  <expression>
                    <literalboolean>true</literalboolean>
                  </expression>
                  <expression>
                    <literalcharacter>'a'</literalcharacter>
                  </expression>
                  <expression>
                    <literalinteger>1234567</literalinteger>
                  </expression>
```

```
            <expression>
              <literalreal>1.687</literalreal>
            </expression>
            <expression>
              <literalstring>"to be or not to be..."</literalstring>
            </expression>
            <expression>
              <literalboolean>false</literalboolean>
            </expression>
            <expression>
              <literalcharacter>'%'</literalcharacter>
            </expression>
            <expression>
              <literalinteger>7654321</literalinteger>
            </expression>
            <expression>
              <literalreal>3.14159</literalreal>
            </expression>
            <expression>
              <literalstring>"e = mc^2"</literalstring>
            </expression>
            <expression>
              <literalstring>"just another one"</literalstring>
            </expression>
          </listagregate>
        </expression>
      </update>
    </rule>
    <rule>
      <block>
        <rule>
          <update>
            <dynamicfunctioncall>
              <name>SetWhereToSearch</name>
            </dynamicfunctioncall>
            <expression>
              <setagregate>
                <expression>
                  <literalinteger>42</literalinteger>
                </expression>
                <expression>
                  <literalinteger>44</literalinteger>
                </expression>
                <expression>
                  <literalinteger>46</literalinteger>
                </expression>
              </setagregate>
            </expression>
          </update>
        </rule>
        <rule>
          <update>
            <dynamicfunctioncall>
```

```
                    <name>OfAnyType</name>
                  </dynamicfunctioncall>
                  <expression>
                    <literalstring>"I'm a string!\n"</literalstring>
                  </expression>
                </update>
              </rule>
            </block>
          </rule>
        </block>
      </rule>
    </block>
  </rule>
</init>

<transition>
  <rule>
    <block>
      <rule>
        <forall>
          <name>item</name>
          <expression>
            <dynamicfunctioncall>
              <name>Items</name>
            </dynamicfunctioncall>
          </expression>
          <rule>
            <with>
              <expression>
                <forallalias>
                  <name>item</name>
                </forallalias>
              </expression>
              <name>b</name>
              <type>
                <boolean/>
              </type>
              <rule>
                <actioncall>
                  <moduleref><name>Output</name></moduleref>
                  <name>Write</name>
                  <expression>
                    <ifexp>
                      <expression>
                        <withalias>
                          <name>b</name>
                        </withalias>
                      </expression>
                      <expression>
                        <literalstring>"BOOLEAN (true)\n"</literalstring>
                      </expression>
                      <expression>
                        <literalstring>"BOOLEAN (false)\n"</literalstring>
```

```
        </expression>
      </ifexp>
    </expression>
  </actioncall>
</rule>
<name>c</name>
<type>
  <character/>
</type>
<rule>
  <actioncall>
    <moduleref><name>Output</name></moduleref>
    <name>Write</name>
    <expression>
      <externalfunctioncall>
        <moduleref><name>StringManipulation</name></moduleref>
        <name>Concat</name>
        <expression>
          <literalstring>"CHARACTER ('"</literalstring>
        </expression>
        <expression>
          <externalfunctioncall>
            <moduleref><name>StringManipulation</name></moduleref>
            <name>Concat</name>
            <expression>
              <externalfunctioncall>
                <moduleref><name>StringManipulation</name></moduleref>
                <name>CharToString</name>
                <expression>
                  <withalias>
                    <name>c</name>
                  </withalias>
                </expression>
              </externalfunctioncall>
            </expression>
            <expression>
              <literalstring>"')\n"</literalstring>
            </expression>
          </externalfunctioncall>
        </expression>
      </externalfunctioncall>
    </expression>
  </actioncall>
</rule>
<name>i</name>
<type>
  <integer/>
</type>
<rule>
  <actioncall>
    <moduleref><name>Output</name></moduleref>
    <name>Write</name>
    <expression>
```

```
      <externalfunctioncall>
        <moduleref><name>StringManipulation</name></moduleref>
        <name>Concat</name>
        <expression>
          <literalstring>"INTEGER ("</literalstring>
        </expression>
        <expression>
          <externalfunctioncall>
            <moduleref><name>StringManipulation</name></moduleref>
            <name>Concat</name>
            <expression>
              <externalfunctioncall>
                <moduleref><name>StringManipulation</name></moduleref>
                <name>IntegerToString</name>
                <expression>
                  <withalias>
                    <name>i</name>
                  </withalias>
                </expression>
              </externalfunctioncall>
            </expression>
            <expression>
              <literalstring>")\n"</literalstring>
            </expression>
          </externalfunctioncall>
        </expression>
      </externalfunctioncall>
    </expression>
  </actioncall>
</rule>
<name>r</name>
<type>
  <real/>
</type>
<rule>
  <actioncall>
    <moduleref><name>Output</name></moduleref>
    <name>Write</name>
    <expression>
      <externalfunctioncall>
        <moduleref><name>StringManipulation</name></moduleref>
        <name>Concat</name>
        <expression>
          <literalstring>"REAL ("</literalstring>
        </expression>
        <expression>
          <externalfunctioncall>
            <moduleref><name>StringManipulation</name></moduleref>
            <name>Concat</name>
            <expression>
              <externalfunctioncall>
                <moduleref><name>StringManipulation</name></moduleref>
                <name>RealToString</name>
```

```
                    <expression>
                      <withalias>
                        <name>r</name>
                      </withalias>
                    </expression>
                  </externalfunctioncall>
                </expression>
                <expression>
                  <literalstring>")\n"</literalstring>
                </expression>
              </externalfunctioncall>
            </expression>
          </externalfunctioncall>
        </expression>
      </actioncall>
    </rule>
    <name>s</name>
    <type>
      <string/>
    </type>
    <rule>
      <case>
        <expression>
          <withalias>
            <name>s</name>
          </withalias>
        </expression>
        <expression>
          <literalstring>"to be or not to be..."</literalstring>
        </expression>
        <rule>
          <actioncall>
            <moduleref><name>Output</name></moduleref>
            <name>Write</name>
            <expression>
              <externalfunctioncall>
                <moduleref><name>StringManipulation</name></moduleref>
                <name>Concat</name>
                <expression>
                  <literalstring>"STRING ("</literalstring>
                </expression>
                <expression>
                  <externalfunctioncall>
                    <moduleref><name>StringManipulation</name></moduleref>
                    <name>Concat</name>
                    <expression>
                      <withalias>
                        <name>s</name>
                      </withalias>
                    </expression>
                    <expression>
                      <literalstring>") from Shakespeare\n"</literalstring>
                    </expression>
```

```
          </externalfunctioncall>
        </expression>
      </externalfunctioncall>
    </expression>
  </actioncall>
</rule>
<expression>
  <literalstring>"e = mc^2"</literalstring>
</expression>
<rule>
  <actioncall>
    <moduleref><name>Output</name></moduleref>
    <name>Write</name>
    <expression>
      <externalfunctioncall>
        <moduleref><name>StringManipulation</name></moduleref>
        <name>Concat</name>
        <expression>
          <literalstring>"STRING ("</literalstring>
        </expression>
        <expression>
          <externalfunctioncall>
            <moduleref><name>StringManipulation</name></moduleref>
            <name>Concat</name>
            <expression>
              <withalias>
                <name>s</name>
              </withalias>
            </expression>
            <expression>
              <literalstring>") from Einstein\n"</literalstring>
            </expression>
          </externalfunctioncall>
        </expression>
      </externalfunctioncall>
    </expression>
  </actioncall>
</rule>
<rule>
  <actioncall>
    <moduleref><name>Output</name></moduleref>
    <name>Write</name>
    <expression>
      <externalfunctioncall>
        <moduleref><name>StringManipulation</name></moduleref>
        <name>Concat</name>
        <expression>
          <literalstring>"STRING ("</literalstring>
        </expression>
        <expression>
          <externalfunctioncall>
            <moduleref><name>StringManipulation</name></moduleref>
            <name>Concat</name>
```

```
                        <expression>
                          <withalias>
                            <name>s</name>
                          </withalias>
                        </expression>
                        <expression>
                          <literalstring>")\n"</literalstring>
                        </expression>
                      </externalfunctioncall>
                    </expression>
                  </externalfunctioncall>
                </expression>
              </actioncall>
            </rule>
          </case>
        </rule>
        <rule>
          <lambda/>
        </rule>
      </with>
    </rule>
  </forall>
</rule>
<rule>
  <block>
    <rule>
      <block>
        <rule>
          <actioncall>
            <moduleref><name>Output</name></moduleref>
            <name>Write</name>
            <expression>
              <withexp>
                <expression>
                  <dynamicfunctioncall>
                    <name>Interval</name>
                  </dynamicfunctioncall>
                </expression>
                <name>i</name>
                <type>
                  <interval>
                    <expression>
                      <literalinteger>0</literalinteger>
                    </expression>
                    <expression>
                      <literalinteger>10000</literalinteger>
                    </expression>
                  </interval>
                </type>
                <expression>
                  <externalfunctioncall>
                    <moduleref><name>StringManipulation</name></moduleref>
                    <name>Concat</name>
```

```
            <expression>
              <externalfunctioncall>
                <moduleref><name>StringManipulation</name></moduleref>
                <name>IntegerToString</name>
                <expression>
                  <typecast>
                    <expression>
                      <withexpalias>
                        <name>i</name>
                      </withexpalias>
                    </expression>
                    <type>
                      <integer/>
                    </type>
                  </typecast>
                </expression>
              </externalfunctioncall>
            </expression>
            <expression>
              <literalstring>" -> Interval found.\n"</literalstring>
            </expression>
          </externalfunctioncall>
        </expression>
        <expression>
          <literalstring>"Not an interval, error!!!\n"</literalstring>
        </expression>
      </withexp>
    </expression>
  </actioncall>
</rule>
<rule>
  <block>
    <rule>
      <actioncall>
        <moduleref><name>Output</name></moduleref>
        <name>Write</name>
        <expression>
          <ifexp>
            <expression>
              <in>
                <expression>
                  <literalinteger>42</literalinteger>
                </expression>
                <expression>
                  <dynamicfunctioncall>
                    <name>SetWhereToSearch</name>
                  </dynamicfunctioncall>
                </expression>
              </in>
            </expression>
            <expression>
              <literalstring>"42 is in set.\n"</literalstring>
            </expression>
```

```
                    <expression>
                      <literalstring>"42 is not in set.\n"</literalstring>
                    </expression>
                  </ifexp>
                </expression>
              </actioncall>
            </rule>
            <rule>
              <actioncall>
                <moduleref><name>Output</name></moduleref>
                <name>Write</name>
                <expression>
                  <ifexp>
                    <expression>
                      <in>
                        <expression>
                          <literalinteger>43</literalinteger>
                        </expression>
                        <expression>
                          <dynamicfunctioncall>
                            <name>SetWhereToSearch</name>
                          </dynamicfunctioncall>
                        </expression>
                      </in>
                    </expression>
                    <expression>
                      <literalstring>"43 is in set.\n"</literalstring>
                    </expression>
                    <expression>
                      <literalstring>"43 is not in set.\n"</literalstring>
                    </expression>
                  </ifexp>
                </expression>
              </actioncall>
            </rule>
          </block>
        </rule>
      </block>
    </rule>
    <rule>
      <block>
        <rule>
          <stop/>
        </rule>
        <rule>
          <with>
            <expression>
              <dynamicfunctioncall>
                <name>OfAnyType</name>
              </dynamicfunctioncall>
            </expression>
            <name>s</name>
            <type>
```

```
                    <string/>
                  </type>
                  <rule>
                    <actioncall>
                      <moduleref><name>Output</name></moduleref>
                      <name>Write</name>
                      <expression>
                        <withalias><name>s</name></withalias>
                      </expression>
                    </actioncall>
                  </rule>
                  <rule>
                    <actioncall>
                      <moduleref><name>Output</name></moduleref>
                      <name>Write</name>
                      <expression>
                        <literalstring>"I'm not a string...\n"</literalstring>
                      </expression>
                    </actioncall>
                  </rule>
                </with>
              </rule>
            </block>
          </rule>
        </block>
      </rule>
    </block>
  </rule>
</transition>

</module>
```

B.1.16 TypeUnionContext.mas

```
<mas>
  <name>TypeUnionContext</name>
  <agents>
    <agent>
      <name>MyAgent</name>
      <moduleref><name>TypeUnion</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>MyAgent</name>
  </dispatches>
</mas>
```

B.1.17 TrueTables.mod

```
<module>

  <name>TrueTables</name>
```

```
<references>
</references>

<imports>
  <import>
    <name>Output</name>
    <global>
      <signatures>
        <actionsignature>
          <name>Write</name>
          <name>s</name>
          <type><string/></type>
        </actionsignature>
      </signatures>
    </global>
  </import>
  <import>
    <name>StringManipulation</name>
    <global>
      <signatures>
        <externalfunctionsignature>
          <name>CharToString</name>
          <type>
            <functional>
              <name>c</name>
              <type><character/></type>
              <type><string/></type>
            </functional>
          </type>
        </externalfunctionsignature>
        <externalfunctionsignature>
          <name>Concat</name>
          <type>
            <functional>
              <name>s</name>
              <type><string/></type>
              <type>
                <functional>
                  <name>r</name>
                  <type><string/></type>
                  <type><string/></type>
                </functional>
              </type>
            </functional>
          </type>
        </externalfunctionsignature>
      </signatures>
    </global>
  </import>

</imports>

<global>
```

```
    <environment>
      <staticandderivedfunctions>
      </staticandderivedfunctions>
      <dynamicfunctions>
      </dynamicfunctions>
      <externalfunctions>
      </externalfunctions>
      <actions>
      </actions>
      <submachines>
      </submachines>
    </environment>
  </global>
  <local>
    <environment>
      <staticandderivedfunctions>
        <staticfunction>
          <name>ToChar</name>
          <type>
            <functional>
              <name>b</name>
              <type><boolean/></type>
              <type><character/></type>
            </functional>
          </type>
          <expression>
            <ifexp>
              <expression>
                <parametercall><name>b</name></parametercall>
              </expression>
              <expression>
                <literalcharacter>'T'</literalcharacter>
              </expression>
              <expression>
                <literalcharacter>'F'</literalcharacter>
              </expression>
```

```
    </ifexp>
  </expression>
</staticfunction>

<staticfunction>
  <name>tablenames</name>
  <type>
    <list>
      <type>
        <string/>
      </type>
    </list>
  </type>
  <expression>
    <listagregate>
      <expression>
        <literalstring>" AND "</literalstring>
      </expression>
      <expression>
        <literalstring>" OR "</literalstring>
      </expression>
      <expression>
        <literalstring>" XOR "</literalstring>
      </expression>
    </listagregate>
  </expression>
</staticfunction>

<staticfunction>
  <name>prototable</name>
  <type>
    <list>
      <type>
        <tuple>
          <name>a</name><type><boolean/></type>
          <name>b</name><type><boolean/></type>
        </tuple>
      </type>
    </list>
  </type>
  <expression>
    <cons>
      <expression>
        <tupleagregate>
          <expression>
            <literalboolean>false</literalboolean>
          </expression>
          <expression>
            <literalboolean>false</literalboolean>
          </expression>
        </tupleagregate>
      </expression>
      <expression>
```

```
            <listagregate>
              <expression>
                <tupleagregate>
                  <expression>
                    <literalboolean>false</literalboolean>
                  </expression>
                  <expression>
                    <literalboolean>true</literalboolean>
                  </expression>
                </tupleagregate>
              </expression>
              <expression>
                <tupleagregate>
                  <expression>
                    <literalboolean>true</literalboolean>
                  </expression>
                  <expression>
                    <literalboolean>false</literalboolean>
                  </expression>
                </tupleagregate>
              </expression>
              <expression>
                <tupleagregate>
                  <expression>
                    <literalboolean>true</literalboolean>
                  </expression>
                  <expression>
                    <literalboolean>true</literalboolean>
                  </expression>
                </tupleagregate>
              </expression>
            </listagregate>
          </expression>
        </cons>
      </expression>
    </staticfunction>

  </staticandderivedfunctions>

  <dynamicfunctions>

  </dynamicfunctions>

  <externalfunctions>

  </externalfunctions>

  <actions>

  </actions>

  <submachines>
```

```
    </submachines>

  </environment>

</local>

<init>

  <rule>
    <lambda/>
  </rule>

</init>

<transition>

  <rule>

    <block>

      <rule>
        <stop/>
      </rule>

      <rule>
        <forall>
          <name>function</name>
          <expression>
            <staticfunctioncall>
              <name>tablenames</name>
            </staticfunctioncall>
          </expression>
          <rule>
            <forall>
              <name>domain</name>
              <expression>
                <staticfunctioncall>
                  <name>prototable</name>
                </staticfunctioncall>
              </expression>
              <rule>
                <actioncall>
                  <moduleref><name>Output</name></moduleref>
                  <name>Write</name>
                  <expression>
                    <externalfunctioncall>
                      <moduleref><name>StringManipulation</name></moduleref>
                      <name>Concat</name>
                      <expression>
                        <externalfunctioncall>
                          <moduleref><name>StringManipulation</name></moduleref>
                          <name>CharToString</name>
                          <expression>
```

```
            <staticfunctioncall>
              <name>ToChar</name>
              <expression>
                <tupleprojection>
                  <expression>
                    <forallalias>
                      <name>domain</name>
                    </forallalias>
                  </expression>
                  <literalinteger>0</literalinteger>
                </tupleprojection>
              </expression>
            </staticfunctioncall>
          </expression>
        </externalfunctioncall>
      </expression>
      <expression>
        <externalfunctioncall>
          <moduleref><name>StringManipulation</name></moduleref>
          <name>Concat</name>
          <expression>
            <forallalias>
              <name>function</name>
            </forallalias>
          </expression>
          <expression>
            <externalfunctioncall>
              <moduleref><name>StringManipulation</name></moduleref>
              <name>Concat</name>
              <expression>
                <externalfunctioncall>
                  <moduleref><name>StringManipulation</name></moduleref>
                  <name>CharToString</name>
                  <expression>
                    <staticfunctioncall>
                      <name>ToChar</name>
                      <expression>
                        <tupleprojection>
                          <expression>
                            <forallalias>
                              <name>domain</name>
                            </forallalias>
                          </expression>
                          <literalinteger>1</literalinteger>
                        </tupleprojection>
                      </expression>
                    </staticfunctioncall>
                  </expression>
                </externalfunctioncall>
              </expression>
              <expression>
                <externalfunctioncall>
                  <moduleref><name>StringManipulation</name></moduleref>
```

```
<name>Concat</name>
<expression>
  <literalstring>" = "</literalstring>
</expression>
<expression>
  <externalfunctioncall>
    <moduleref><name>StringManipulation</name></moduleref>
    <name>Concat</name>
    <expression>
      <caseexp>
        <expression>
          <forallalias>
            <name>function</name>
          </forallalias>
        </expression>
        <expression>
          <literalstring>" AND "</literalstring>
        </expression>
        <expression>
          <externalfunctioncall>
            <moduleref><name>StringManipulation</name></module
            <name>CharToString</name>
            <expression>
              <staticfunctioncall>
                <name>ToChar</name>
                <expression>
                  <and>
                    <expression>
                      <tupleprojection>
                        <expression>
                          <forallalias>
                            <name>domain</name>
                          </forallalias>
                        </expression>
                        <literalinteger>0</literalinteger>
                      </tupleprojection>
                    </expression>
                    <expression>
                      <tupleprojection>
                        <expression>
                          <forallalias>
                            <name>domain</name>
                          </forallalias>
                        </expression>
                        <literalinteger>1</literalinteger>
                      </tupleprojection>
                    </expression>
                  </and>
                </expression>
              </staticfunctioncall>
            </expression>
          </externalfunctioncall>
        </expression>
```

```
<expression>
  <literalstring>" OR "</literalstring>
</expression>
<expression>
  <externalfunctioncall>
    <moduleref><name>StringManipulation</name></module
    <name>CharToString</name>
    <expression>
      <staticfunctioncall>
        <name>ToChar</name>
        <expression>
          <or>
            <expression>
              <tupleprojection>
                <expression>
                  <forallalias>
                    <name>domain</name>
                  </forallalias>
                </expression>
                <literalinteger>0</literalinteger>
              </tupleprojection>
            </expression>
            <expression>
              <tupleprojection>
                <expression>
                  <forallalias>
                    <name>domain</name>
                  </forallalias>
                </expression>
                <literalinteger>1</literalinteger>
              </tupleprojection>
            </expression>
          </or>
        </expression>
      </staticfunctioncall>
    </expression>
  </externalfunctioncall>
</expression>
<expression>
  <literalstring>" XOR "</literalstring>
</expression>
<expression>
  <externalfunctioncall>
    <moduleref><name>StringManipulation</name></module
    <name>CharToString</name>
    <expression>
      <staticfunctioncall>
        <name>ToChar</name>
        <expression>
          <xor>
            <expression>
              <tupleprojection>
                <expression>
```

```
                                                    <forallalias>
                                                      <name>domain</name>
                                                    </forallalias>
                                                  </expression>
                                                  <literalinteger>0</literalinteger>
                                                </tupleprojection>
                                              </expression>
                                              <expression>
                                                <tupleprojection>
                                                  <expression>
                                                    <forallalias>
                                                      <name>domain</name>
                                                    </forallalias>
                                                  </expression>
                                                  <literalinteger>1</literalinteger>
                                                </tupleprojection>
                                              </expression>
                                            </xor>
                                          </expression>
                                        </staticfunctioncall>
                                      </expression>
                                    </externalfunctioncall>
                                  </expression>
                                  <expression>
                                    <literalstring>"undef"</literalstring>
                                  </expression>
                                </caseexp>
                              </expression>
                              <expression>
                                <literalstring>" \n"</literalstring>
                              </expression>
                            </externalfunctioncall>
                          </expression>
                        </externalfunctioncall>
                      </expression>
                    </externalfunctioncall>
                  </expression>
                </externalfunctioncall>
              </expression>
            </externalfunctioncall>
          </expression>
        </actioncall>
      </rule>
    </forall>
  </rule>
</forall>

  </rule>
 </block>

</rule>

</transition>
```

```
</module>
```

B.1.18 TrueTablesContext.mas

```
<mas>
  <name>TrueTablesContext</name>
  <agents>
    <agent>
      <name>MyAgent</name>
      <moduleref><name>TrueTables</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>MyAgent</name>
  </dispatches>
</mas>
```

B.1.19 NoSync.mod

```
<module>

  <name>NoSync</name>

  <references>
    <reference>
      <name>Alarm</name>
    </reference>
  </references>

  <imports>
    <import>
      <name>Output</name>
      <global>
        <signatures>
          <actionsignature>
            <name>Write</name>
            <name>s</name>
            <type><string/></type>
          </actionsignature>
        </signatures>
      </global>
    </import>
  </imports>

  <global>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>
```

```
    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>
      <externalfunction>
        <name>AskForInstruction</name>
        <type>
          <character/>
        </type>
      </externalfunction>
    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>
```

```
<init>

  <rule>

    <lambda>

    </lambda>

  </rule>

</init>

<transition>

  <rule>

    <case>
      <expression>
        <externalfunctioncall>
          <name>AskForInstruction</name>
        </externalfunctioncall>
      </expression>
      <expression>
        <literalcharacter>'o'</literalcharacter>
      </expression>
      <rule>
        <block>
          <rule>
            <actioncall>
              <moduleref><name>Output</name></moduleref>
              <name>Write</name>
              <expression>
                <literalstring>"The door is open.\n"</literalstring>
              </expression>
            </actioncall>
          </rule>
          <rule>
            <block>
              <rule>
                <create>
                  <expression><literalstring>"alarm"</literalstring></expression>
                  <moduleref><name>Alarm</name></moduleref>
                </create>
              </rule>
              <rule>
                <dispatch>
                  <expression><literalstring>"alarm"</literalstring></expression>
                </dispatch>
              </rule>
            </block>
          </rule>
        </block>
```

```
        </rule>
        <expression>
          <literalcharacter>'c'</literalcharacter>
        </expression>
        <rule>
          <block>
            <rule>
              <actioncall>
                <moduleref><name>Output</name></moduleref>
                <name>Write</name>
                <expression>
                  <literalstring>"The door is closed.\n"</literalstring>
                </expression>
              </actioncall>
            </rule>
            <rule>
              <destroy>
                <expression><literalstring>"alarm"</literalstring></expression>
              </destroy>
            </rule>
          </block>
        </rule>
        <rule>
          <stop/>
        </rule>
      </case>

    </rule>

  </transition>

</module>
```

B.1.20 Alarm.mod

```
<module>

  <name>Alarm</name>

  <references>
  </references>

  <imports>
    <import>
      <name>Output</name>
      <global>
        <signatures>
          <actionsignature>
            <name>Write</name>
            <name>s</name>
            <type><string/></type>
          </actionsignature>
        </signatures>
```

```
    </global>
  </import>
</imports>

<global>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>
```

```
      </submachines>

    </environment>

  </local>

  <init>

    <rule>

      <lambda>

      </lambda>

    </rule>

  </init>

  <transition>

    <rule>

      <actioncall>
        <moduleref><name>Output</name></moduleref>
        <name>Write</name>
        <expression>
          <literalstring>"WARNING: THE DOOR IS OPEN!!!\n"</literalstring>
        </expression>
      </actioncall>

    </rule>

  </transition>

</module>
```

B.1.21 NoSyncContext.mas

```
<mas>
  <name>NoSyncContext</name>
  <agents>
    <agent>
      <name>MyAgent</name>
      <moduleref><name>NoSync</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>MyAgent</name>
  </dispatches>
</mas>
```

B.1.22 PhilosophersDinning.mod

```
<module>

  <name>PhilosophersDinning</name>

  <references>
    <reference>
      <name>Table</name>
    </reference>
    <reference>
      <name>Philosopher</name>
    </reference>
  </references>

  <imports>
    <import>
      <name>Output</name>
      <global>
        <signatures>
          <actionsignature>
            <name>Write</name>
            <name>s</name>
            <type><string/></type>
          </actionsignature>
        </signatures>
      </global>
    </import>
    <import>
      <name>StringManipulation</name>
      <global>
        <signatures>
          <externalfunctionsignature>
            <name>IntegerToString</name>
            <type>
              <functional>
                <name>i</name>
                <type><integer/></type>
                <type><string/></type>
              </functional>
            </type>
          </externalfunctionsignature>
          <externalfunctionsignature>
            <name>Concat</name>
            <type>
              <functional>
                <name>s</name>
                <type><string/></type>
                <type>
                  <functional>
                    <name>r</name>
                    <type><string/></type>
                    <type><string/></type>
```

```
                </functional>
              </type>
            </functional>
          </type>
        </externalfunctionsignature>
      </signatures>
    </global>
  </import>
</imports>

<global>

  <environment>

    <staticandderivedfunctions>
      <staticfunction>
        <name>PhilosophersNames</name>
        <type>
          <set>
            <type>
              <string/>
            </type>
          </set>
        </type>
        <expression>
          <setagregate>
            <expression><literalstring>"Kant"</literalstring></expression>
            <expression><literalstring>"Nietsche"</literalstring></expression>
            <expression><literalstring>"Socrates"</literalstring></expression>
            <expression><literalstring>"Platao"</literalstring></expression>
            <expression><literalstring>"Aristoteles"</literalstring></expression>
          </setagregate>
        </expression>
      </staticfunction>
    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>
```

```
</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>
      <dynamicfunction>
        <name>step</name>
        <type><integer/></type>
      </dynamicfunction>
    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>

<init>
  <rule>
    <update>
      <dynamicfunctioncall>
        <name>step</name>
      </dynamicfunctioncall>
      <expression>
        <literalinteger>0</literalinteger>
      </expression>
    </update>
  </rule>
</init>

<transition>
  <rule>
    <case>
      <expression>
        <dynamicfunctioncall>
          <name>step</name>
```

```
  </dynamicfunctioncall>
</expression>
<expression>
  <literalinteger>0</literalinteger>
</expression>
<rule>
  <block>
    <rule>
      <block>
        <rule>
          <create>
            <expression><literalstring>"table"</literalstring></expression>
            <moduleref><name>Table</name></moduleref>
          </create>
        </rule>
        <rule>
          <dispatch>
            <expression><literalstring>"table"</literalstring></expression>
          </dispatch>
        </rule>
      </block>
    </rule>
    <rule>
      <update>
        <dynamicfunctioncall>
          <name>step</name>
        </dynamicfunctioncall>
        <expression>
          <add>
            <expression>
              <dynamicfunctioncall>
                <name>step</name>
              </dynamicfunctioncall>
            </expression>
            <expression>
              <literalinteger>1</literalinteger>
            </expression>
          </add>
        </expression>
      </update>
    </rule>
  </block>
</rule>
<expression>
  <literalinteger>1</literalinteger>
</expression>
<rule>
  <block>
    <rule>
      <forall>
        <name>phi</name>
        <expression>
          <staticfunctioncall>
```

```
            <name>PhilosophersNames</name>
          </staticfunctioncall>
        </expression>
        <rule>
          <create>
            <expression><forallalias><name>phi</name></forallalias></expression>
            <moduleref><name>Philosopher</name></moduleref>
          </create>
        </rule>
      </forall>
    </rule>
    <rule>
      <update>
        <dynamicfunctioncall>
          <name>step</name>
        </dynamicfunctioncall>
        <expression>
          <add>
            <expression>
              <dynamicfunctioncall>
                <name>step</name>
              </dynamicfunctioncall>
            </expression>
            <expression>
              <literalinteger>1</literalinteger>
            </expression>
          </add>
        </expression>
      </update>
    </rule>
  </block>
</rule>
<expression>
  <literalinteger>2</literalinteger>
</expression>
<rule>
  <block>
    <rule>
      <forall>
        <name>phi</name>
        <expression>
          <staticfunctioncall>
            <name>PhilosophersNames</name>
          </staticfunctioncall>
        </expression>
        <rule>
          <dispatch>
            <expression><forallalias><name>phi</name></forallalias></expression>
          </dispatch>
        </rule>
      </forall>
    </rule>
    <rule>
```

```
              <update>
                <dynamicfunctioncall>
                  <name>step</name>
                </dynamicfunctioncall>
                <expression>
                  <add>
                    <expression>
                      <dynamicfunctioncall>
                        <name>step</name>
                      </dynamicfunctioncall>
                    </expression>
                    <expression>
                      <literalinteger>1</literalinteger>
                    </expression>
                  </add>
                </expression>
              </update>
            </rule>
          </block>
        </rule>
        <rule>
          <lambda/>
        </rule>
      </case>
    </rule>
  </transition>

</module>
```

B.1.23 Philosopher.mod

```
<module>

  <name>Philosopher</name>

  <references>
    <reference>
      <name>Table</name>
      <global>
        <signatures>
          <semaphoresignature>
            <name>ForksAccessRight</name>
          </semaphoresignature>
          <dynamicfunctionsignature>
            <name>PhilosophersCounter</name>
            <type><integer/></type>
          </dynamicfunctionsignature>
          <dynamicfunctionsignature>
            <name>Forks</name>
            <type>
              <functional>
                <name>i</name>
                <type>
```

```
                <integer/>
              </type>
              <type>
                <boolean/>
              </type>
            </functional>
          </type>
        </dynamicfunctionsignature>
      </signatures>
    </global>
  </reference>
</references>

<imports>
  <import>
    <name>Output</name>
    <global>
      <signatures>
        <actionsignature>
          <name>Write</name>
          <name>s</name>
          <type><string/></type>
        </actionsignature>
      </signatures>
    </global>
  </import>
  <import>
    <name>StringManipulation</name>
    <global>
      <signatures>
        <externalfunctionsignature>
          <name>IntegerToString</name>
          <type>
            <functional>
              <name>i</name>
              <type><integer/></type>
              <type><string/></type>
            </functional>
          </type>
        </externalfunctionsignature>
        <externalfunctionsignature>
          <name>Concat</name>
          <type>
            <functional>
              <name>s</name>
              <type><string/></type>
              <type>
                <functional>
                  <name>r</name>
                  <type><string/></type>
                  <type><string/></type>
                </functional>
              </type>
```

```
            </functional>
          </type>
        </externalfunctionsignature>
      </signatures>
    </global>
  </import>
</imports>

<global>

  <environment>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

      <derivedfunction>
        <name>LeftFork</name>
        <type><integer/></type>
        <expression>
          <dynamicfunctioncall>
            <name>id</name>
          </dynamicfunctioncall>
        </expression>
      </derivedfunction>

      <derivedfunction>
        <name>RightFork</name>
        <type><integer/></type>
        <expression>
          <mod>
            <expression>
              <add>
                <expression>
                  <dynamicfunctioncall>
                    <name>id</name>
                  </dynamicfunctioncall>
                </expression>
                <expression>
                  <literalinteger>1</literalinteger>
                </expression>
              </add>
            </expression>
            <expression>
              <literalinteger>5</literalinteger>
            </expression>
          </mod>
```

```
          </expression>
        </derivedfunction>

      </staticandderivedfunctions>

      <dynamicfunctions>

        <dynamicfunction>
          <name>step</name>
          <type>
            <integer/>
          </type>
        </dynamicfunction>

        <dynamicfunction>
          <name>id</name>
          <type>
            <integer/>
          </type>
        </dynamicfunction>

        <dynamicfunction>
          <name>state</name>
          <type>
            <string/>
          </type>
        </dynamicfunction>

      </dynamicfunctions>

    </environment>

  </local>

  <init>
    <rule>
      <block>
        <rule>
          <update>
            <dynamicfunctioncall>
              <name>step</name>
            </dynamicfunctioncall>
            <expression>
              <literalinteger>0</literalinteger>
            </expression>
          </update>
        </rule>
        <rule>
          <block>
            <rule>
              <update>
                <dynamicfunctioncall>
                  <name>state</name>
```

```
              </dynamicfunctioncall>
              <expression>
                <literalstring>"thinking"</literalstring>
              </expression>
            </update>
          </rule>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>id</name>
              </dynamicfunctioncall>
              <expression>
                <literalinteger>-1</literalinteger>
              </expression>
            </update>
          </rule>
        </block>
      </rule>
    </block>
  </rule>
</init>

<transition>

  <rule>
    <case>
      <expression>
        <dynamicfunctioncall>
          <name>step</name>
        </dynamicfunctioncall>
      </expression>
      <expression>
        <literalinteger>0</literalinteger>
      </expression>
      <rule>
        <block>
          <rule>
            <wait>
              <moduleref><name>Table</name></moduleref>
              <name>ForksAccessRight</name>
            </wait>
          </rule>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>step</name>
              </dynamicfunctioncall>
              <expression>
                <literalinteger>1</literalinteger>
              </expression>
            </update>
          </rule>
        </block>
```

```
</rule>
<expression>
  <literalinteger>1</literalinteger>
</expression>
<rule>
  <block>
    <rule>
      <case>
        <expression>
          <dynamicfunctioncall>
            <name>state</name>
          </dynamicfunctioncall>
        </expression>
        <expression>
          <literalstring>"hungry"</literalstring>
        </expression>
        <rule>
          <block>
            <rule>
              <actioncall>
                <moduleref><name>Output</name></moduleref>
                <name>Write</name>
                <expression>
                  <externalfunctioncall>
                    <moduleref><name>StringManipulation</name></moduleref>
                    <name>Concat</name>
                    <expression>
                      <self/>
                    </expression>
                    <expression>
                      <literalstring>" is hungry.\n"</literalstring>
                    </expression>
                  </externalfunctioncall>
                </expression>
              </actioncall>
            </rule>
            <rule>
              <conditional>
                <expression>
                  <and>
                    <expression>
                      <not>
                        <expression>
                          <dynamicfunctioncall>
                            <moduleref><name>Table</name></moduleref>
                            <name>Forks</name>
                            <expression>
                              <derivedfunctioncall>
                                <name>LeftFork</name>
                              </derivedfunctioncall>
                            </expression>
                          </dynamicfunctioncall>
                        </expression>
```

```
          </not>
        </expression>
        <expression>
          <not>
            <expression>
              <dynamicfunctioncall>
                <moduleref><name>Table</name></moduleref>
                <name>Forks</name>
                <expression>
                  <derivedfunctioncall>
                    <name>RightFork</name>
                  </derivedfunctioncall>
                </expression>
              </dynamicfunctioncall>
            </expression>
          </not>
        </expression>
      </and>
    </expression>
    <rule>
      <block>
        <rule>
          <update>
            <dynamicfunctioncall>
              <name>state</name>
            </dynamicfunctioncall>
            <expression>
              <literalstring>"eating"</literalstring>
            </expression>
          </update>
        </rule>
        <rule>
          <block>
            <rule>
              <update>
                <dynamicfunctioncall>
                  <moduleref><name>Table</name></moduleref>
                  <name>Forks</name>
                  <expression>
                    <derivedfunctioncall>
                      <name>RightFork</name>
                    </derivedfunctioncall>
                  </expression>
                </dynamicfunctioncall>
                <expression>
                  <literalboolean>true</literalboolean>
                </expression>
              </update>
            </rule>
            <rule>
              <update>
                <dynamicfunctioncall>
                  <moduleref><name>Table</name></moduleref>
```

```
                        <name>Forks</name>
                        <expression>
                          <derivedfunctioncall>
                            <name>LeftFork</name>
                          </derivedfunctioncall>
                        </expression>
                      </dynamicfunctioncall>
                      <expression>
                        <literalboolean>true</literalboolean>
                      </expression>
                    </update>
                  </rule>
                </block>
              </rule>
            </block>
          </rule>
          <rule>
            <lambda/>
          </rule>
        </conditional>
      </rule>
    </block>
  </rule>
  <expression>
    <literalstring>"eating"</literalstring>
  </expression>
  <rule>
    <block>
      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>state</name>
              </dynamicfunctioncall>
              <expression>
               <literalstring>"thinking"</literalstring>
              </expression>
            </update>
          </rule>
          <rule>
            <actioncall>
              <moduleref><name>Output</name></moduleref>
              <name>Write</name>
              <expression>
                <externalfunctioncall>
                  <moduleref><name>StringManipulation</name></moduleref>
                  <name>Concat</name>
                  <expression>
                    <self/>
                  </expression>
                  <expression>
                    <literalstring>" is eating.\n"</literalstring>
```

```
                    </expression>
                  </externalfunctioncall>
                </expression>
              </actioncall>
            </rule>
          </block>
        </rule>
        <rule>
          <block>
            <rule>
              <update>
                <dynamicfunctioncall>
                  <moduleref><name>Table</name></moduleref>
                  <name>Forks</name>
                  <expression>
                    <derivedfunctioncall>
                      <name>RightFork</name>
                    </derivedfunctioncall>
                  </expression>
                </dynamicfunctioncall>
                <expression>
                  <literalboolean>false</literalboolean>
                </expression>
              </update>
            </rule>
            <rule>
              <update>
                <dynamicfunctioncall>
                  <moduleref><name>Table</name></moduleref>
                  <name>Forks</name>
                  <expression>
                    <derivedfunctioncall>
                      <name>LeftFork</name>
                    </derivedfunctioncall>
                  </expression>
                </dynamicfunctioncall>
                <expression>
                  <literalboolean>false</literalboolean>
                </expression>
              </update>
            </rule>
          </block>
        </rule>
      </block>
    </rule>
    <expression>
      <literalstring>"thinking"</literalstring>
    </expression>
    <rule>
      <block>
        <rule>
          <block>
            <rule>
```

```
        <update>
          <dynamicfunctioncall>
            <name>state</name>
          </dynamicfunctioncall>
          <expression>
            <literalstring>"hungry"</literalstring>
          </expression>
        </update>
      </rule>
      <rule>
        <actioncall>
          <moduleref><name>Output</name></moduleref>
          <name>Write</name>
          <expression>
            <externalfunctioncall>
              <moduleref><name>StringManipulation</name></moduleref>
              <name>Concat</name>
              <expression>
                <self/>
              </expression>
              <expression>
                <literalstring>" is thinking.\n"</literalstring>
              </expression>
            </externalfunctioncall>
          </expression>
        </actioncall>
      </rule>
    </block>
  </rule>
  <rule>
    <conditional>
      <expression>
        <lessthan>
          <expression>
            <dynamicfunctioncall>
              <name>id</name>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <literalinteger>0</literalinteger>
          </expression>
        </lessthan>
      </expression>
      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>id</name>
              </dynamicfunctioncall>
              <expression>
                <dynamicfunctioncall>
                  <moduleref><name>Table</name></moduleref>
```

```
                                <name>PhilosophersCounter</name>
                              </dynamicfunctioncall>
                            </expression>
                          </update>
                        </rule>
                        <rule>
                          <update>
                            <dynamicfunctioncall>
                              <moduleref><name>Table</name></moduleref>
                              <name>PhilosophersCounter</name>
                            </dynamicfunctioncall>
                            <expression>
                              <add>
                                <expression>
                                  <dynamicfunctioncall>
                                    <moduleref><name>Table</name></moduleref>
                                    <name>PhilosophersCounter</name>
                                  </dynamicfunctioncall>
                                </expression>
                                <expression>
                                  <literalinteger>1</literalinteger>
                                </expression>
                              </add>
                            </expression>
                          </update>
                        </rule>
                      </block>
                    </rule>
                    <rule>
                      <lambda/>
                    </rule>
                  </conditional>
                </rule>
              </block>
            </rule>
            <rule>
              <lambda/>
            </rule>
          </case>
        </rule>
        <rule>
          <update>
            <dynamicfunctioncall>
              <name>step</name>
            </dynamicfunctioncall>
            <expression>
              <literalinteger>2</literalinteger>
            </expression>
          </update>
        </rule>
      </block>
    </rule>
    <expression>
```

```
          <literalinteger>2</literalinteger>
        </expression>
        <rule>
          <block>
            <rule>
              <signal>
                <moduleref><name>Table</name></moduleref>
                <name>ForksAccessRight</name>
              </signal>
            </rule>
            <rule>
              <update>
                <dynamicfunctioncall>
                  <name>step</name>
                </dynamicfunctioncall>
                <expression>
                  <literalinteger>0</literalinteger>
                </expression>
              </update>
            </rule>
          </block>
        </rule>
        <rule>
          <lambda/>
        </rule>
      </case>
    </rule>

  </transition>

</module>
```

B.1.24 Table.mod

```
<module>

  <name>Table</name>

  <references>
  </references>

  <imports>
  </imports>

  <global>
    <environment>
      <staticandderivedfunctions>
        <staticfunction>
          <name>MaxForks</name>
          <type><integer/></type>
          <expression><literalinteger>5</literalinteger></expression>
        </staticfunction>
      </staticandderivedfunctions>
```

```
    <dynamicfunctions>
      <dynamicfunction>
        <name>Forks</name>
        <type>
          <functional>
            <name>i</name>
            <type>
              <integer/>
            </type>
            <type>
              <boolean/>
            </type>
          </functional>
        </type>
      </dynamicfunction>
      <dynamicfunction>
        <name>i</name>
        <type><integer/></type>
      </dynamicfunction>
      <dynamicfunction>
        <name>PhilosophersCounter</name>
        <type><integer/></type>
      </dynamicfunction>
    </dynamicfunctions>
    <semaphores>
      <semaphore>
        <name>ForksAccessRight</name>
        <literalinteger>1</literalinteger>
      </semaphore>
      <semaphore>
        <name>Sleep</name>
        <literalinteger>0</literalinteger>
      </semaphore>
    </semaphores>
  </environment>
</global>

<local>
  <environment>
  </environment>
</local>

<init>

  <rule>
    <block>
      <rule>
        <update>
          <dynamicfunctioncall>
            <name>i</name>
          </dynamicfunctioncall>
          <expression>
            <literalinteger>0</literalinteger>
```

```
          </expression>
        </update>
      </rule>
      <rule>
        <update>
          <dynamicfunctioncall>
            <name>PhilosophersCounter</name>
          </dynamicfunctioncall>
          <expression>
            <literalinteger>0</literalinteger>
          </expression>
        </update>
      </rule>
    </block>
  </rule>

</init>

<transition>

  <rule>
    <conditional>
      <expression>
        <lessthan>
          <expression>
            <dynamicfunctioncall>
              <name>i</name>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <staticfunctioncall>
              <name>MaxForks</name>
            </staticfunctioncall>
          </expression>
        </lessthan>
      </expression>
      <rule>
        <block>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>Forks</name>
                <expression>
                  <dynamicfunctioncall>
                    <name>i</name>
                  </dynamicfunctioncall>
                </expression>
              </dynamicfunctioncall>
              <expression>
                <literalboolean>false</literalboolean>
              </expression>
            </update>
          </rule>
```

```
            <rule>
              <update>
                <dynamicfunctioncall>
                  <name>i</name>
                </dynamicfunctioncall>
                <expression>
                  <add>
                    <expression>
                      <dynamicfunctioncall>
                        <name>i</name>
                      </dynamicfunctioncall>
                    </expression>
                    <expression>
                      <literalinteger>1</literalinteger>
                    </expression>
                  </add>
                </expression>
              </update>
            </rule>
          </block>
        </rule>
        <rule>
          <wait>
            <name>Sleep</name>
          </wait>
        </rule>
      </conditional>
    </rule>

  </transition>

</module>
```

B.1.25 PhilosophersDinningContext.mas

```
<mas>
  <name>PhilosophersDinningContext</name>
  <agents>
    <agent>
      <name>MyAgent</name>
      <moduleref><name>PhilosophersDinning</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>MyAgent</name>
  </dispatches>
</mas>
```

B.1.26 ProdCons.mod

```
<module>

  <name>ProdCons</name>
```

```
<references>
  <reference>
    <name>Producer</name>
  </reference>
  <reference>
    <name>Consumer</name>
  </reference>
</references>

<imports>
  <import>
    <name>Output</name>
    <global>
      <signatures>
        <actionsignature>
          <name>Write</name>
          <name>s</name>
          <type><string/></type>
        </actionsignature>
      </signatures>
    </global>
  </import>
  <import>
    <name>StringManipulation</name>
    <global>
      <signatures>
        <externalfunctionsignature>
          <name>IntegerToString</name>
          <type>
            <functional>
              <name>i</name>
              <type><integer/></type>
              <type><string/></type>
            </functional>
          </type>
        </externalfunctionsignature>
        <externalfunctionsignature>
          <name>Concat</name>
          <type>
            <functional>
              <name>s</name>
              <type><string/></type>
              <type>
                <functional>
                  <name>r</name>
                  <type><string/></type>
                  <type><string/></type>
                </functional>
              </type>
            </functional>
          </type>
        </externalfunctionsignature>
```

```
      </signatures>
    </global>
  </import>
</imports>

<global>

  <environment>

    <staticandderivedfunctions>

      <staticfunction>
        <name>MaxPrime</name>
        <type>
          <integer/>
        </type>
        <expression>
          <literalinteger>2000000000</literalinteger>
        </expression>
      </staticfunction>

      <staticfunction>
        <name>BufferSize</name>
        <type>
          <integer/>
        </type>
        <expression>
          <literalinteger>100</literalinteger>
        </expression>
      </staticfunction>

      <derivedfunction>
        <name>next</name>
        <type>
          <functional>
            <name>k</name>
            <type><integer/></type>
            <type><integer/></type>
          </functional>
        </type>
        <expression>
          <ifexp>
            <expression>
              <lessthan>
                <expression>
                  <parametercall>
                    <name>k</name>
                  </parametercall>
                </expression>
                <expression>
                  <minus>
                    <expression>
                      <staticfunctioncall>
```

```
                    <name>BufferSize</name>
                  </staticfunctioncall>
                </expression>
                <expression>
                  <literalinteger>1</literalinteger>
                </expression>
              </minus>
            </expression>
          </lessthan>
        </expression>
        <expression>
          <add>
            <expression>
              <parametercall>
                <name>k</name>
              </parametercall>
            </expression>
            <expression>
              <literalinteger>1</literalinteger>
            </expression>
          </add>
        </expression>
        <expression>
          <literalinteger>0</literalinteger>
        </expression>
      </ifexp>
    </expression>
  </derivedfunction>

</staticandderivedfunctions>

<dynamicfunctions>

  <dynamicfunction>
    <name>in</name>
    <type>
      <integer/>
    </type>
  </dynamicfunction>

  <dynamicfunction>
    <name>out</name>
    <type>
      <integer/>
    </type>
  </dynamicfunction>

  <dynamicfunction>
    <name>Buffer</name>
    <type>
      <functional>
        <name>i</name>
        <type><integer/></type>
```

```
            <type><integer/></type>
          </functional>
        </type>
      </dynamicfunction>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>

    <actions>

    </actions>

    <submachines>

    </submachines>

    <semaphores>

      <semaphore>
        <name>filled</name>
        <literalinteger>0</literalinteger>
      </semaphore>

      <semaphore>
        <name>available</name>
        <literalinteger>100</literalinteger>
      </semaphore>

    </semaphores>

  </environment>

</global>

<local>

  <environment>

    <staticandderivedfunctions>

    </staticandderivedfunctions>

    <dynamicfunctions>

    </dynamicfunctions>

    <externalfunctions>

    </externalfunctions>
```

```
      <actions>

      </actions>

      <submachines>

      </submachines>

      <semaphores>

      </semaphores>

    </environment>

  </local>

  <init>
    <rule>
      <block>
        <rule>
          <block>
            <rule>
              <immediateupdate>
                <dynamicfunctioncall>
                  <name>in</name>
                </dynamicfunctioncall>
                <expression>
                  <literalinteger>0</literalinteger>
                </expression>
              </immediateupdate>
            </rule>
            <rule>
              <immediateupdate>
                <dynamicfunctioncall>
                  <name>out</name>
                </dynamicfunctioncall>
                <expression>
                  <literalinteger>0</literalinteger>
                </expression>
              </immediateupdate>
            </rule>
          </block>
        </rule>
        <rule>
          <block>
            <rule>
              <block>
                <rule>
                  <create>
                    <expression><literalstring>"producer"</literalstring></expression>
                    <moduleref><name>Producer</name></moduleref>
                  </create>
                </rule>
```

```
              <rule>
                <create>
                  <expression><literalstring>"consumer"</literalstring></expression>
                  <moduleref><name>Consumer</name></moduleref>
                </create>
              </rule>
            </block>
          </rule>
          <rule>
            <block>
              <rule>
                <dispatch>
                  <expression><literalstring>"producer"</literalstring></expression>
                </dispatch>
              </rule>
              <rule>
                <dispatch>
                  <expression><literalstring>"consumer"</literalstring></expression>
                </dispatch>
              </rule>
            </block>
          </rule>
        </block>
      </rule>
    </block>
  </rule>
</init>

<transition>
  <rule>
    <lambda/>
  </rule>
</transition>

</module>
```

B.1.27 Consumer.mod

```
<module>

  <name>Consumer</name>

  <references>
    <reference>
      <name>ProdCons</name>
      <global>
        <signatures>
          <staticfunctionsignature>
            <name>MaxPrime</name>
            <type><integer/></type>
          </staticfunctionsignature>
          <semaphoresignature>
            <name>filled</name>
```

```
          </semaphoresignature>
          <semaphoresignature>
            <name>available</name>
          </semaphoresignature>
          <dynamicfunctionsignature>
            <name>out</name>
            <type>
              <integer/>
            </type>
            </dynamicfunctionsignature>
          <dynamicfunctionsignature>
            <name>Buffer</name>
            <type>
              <functional>
                <name>i</name>
                <type><integer/></type>
                <type><integer/></type>
              </functional>
            </type>
          </dynamicfunctionsignature>
          <derivedfunctionsignature>
            <name>next</name>
            <type>
              <functional>
                <name>k</name>
                <type><integer/></type>
                <type><integer/></type>
              </functional>
            </type>
          </derivedfunctionsignature>
        </signatures>
      </global>
    </reference>
  </references>

  <imports>
    <import>
      <name>Output</name>
      <global>
        <signatures>
          <actionsignature>
            <name>Write</name>
            <name>s</name>
            <type><string/></type>
          </actionsignature>
        </signatures>
      </global>
    </import>
    <import>
      <name>StringManipulation</name>
      <global>
        <signatures>
          <externalfunctionsignature>
```

```
            <name>IntegerToString</name>
            <type>
              <functional>
                <name>i</name>
                <type><integer/></type>
                <type><string/></type>
              </functional>
            </type>
          </externalfunctionsignature>
          <externalfunctionsignature>
            <name>Concat</name>
            <type>
              <functional>
                <name>s</name>
                <type><string/></type>
                <type>
                  <functional>
                    <name>r</name>
                    <type><string/></type>
                    <type><string/></type>
                  </functional>
                </type>
              </functional>
            </type>
          </externalfunctionsignature>
        </signatures>
      </global>
    </import>
  </imports>

  <global>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>

      <dynamicfunctions>

      </dynamicfunctions>

      <externalfunctions>

      </externalfunctions>

      <actions>

      </actions>

      <submachines>

      </submachines>
```

```
    </environment>

  </global>

  <local>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>

      <dynamicfunctions>

        <dynamicfunction>
          <name>prime</name>
          <type><integer/></type>
        </dynamicfunction>

        <dynamicfunction>
          <name>step</name>
          <type><integer/></type>
        </dynamicfunction>

      </dynamicfunctions>

      <externalfunctions>

      </externalfunctions>

      <actions>

      </actions>

      <submachines>

      </submachines>

    </environment>

  </local>

  <init>
    <rule>
      <block>
        <rule>
          <update>
            <dynamicfunctioncall>
              <name>prime</name>
            </dynamicfunctioncall>
            <expression>
              <literalinteger>2</literalinteger>
```

```
          </expression>
        </update>
      </rule>
      <rule>
        <update>
          <dynamicfunctioncall>
            <name>step</name>
          </dynamicfunctioncall>
          <expression>
            <literalinteger>0</literalinteger>
          </expression>
        </update>
      </rule>
    </block>
  </rule>
</init>

<transition>
  <rule>
    <conditional>
      <expression>
        <lessthan>
          <expression>
            <dynamicfunctioncall>
              <name>prime</name>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <staticfunctioncall>
              <moduleref><name>ProdCons</name></moduleref>
              <name>MaxPrime</name>
            </staticfunctioncall>
          </expression>
        </lessthan>
      </expression>
      <rule>
        <case>
          <expression>
            <dynamicfunctioncall>
              <name>step</name>
            </dynamicfunctioncall>
          </expression>
          <expression>
            <literalinteger>0</literalinteger>
          </expression>
          <rule>
            <block>
              <rule>
                <wait>
                  <moduleref><name>ProdCons</name></moduleref>
                  <name>filled</name>
                </wait>
              </rule>
```

```
    <rule>
      <update>
        <dynamicfunctioncall>
          <name>step</name>
        </dynamicfunctioncall>
        <expression>
          <literalinteger>1</literalinteger>
        </expression>
      </update>
    </rule>
  </block>
</rule>
<expression>
  <literalinteger>1</literalinteger>
</expression>
<rule>
  <block>
    <rule>
      <block>
        <rule>
          <update>
            <dynamicfunctioncall>
              <name>prime</name>
            </dynamicfunctioncall>
            <expression>
              <dynamicfunctioncall>
                <moduleref><name>ProdCons</name></moduleref>
                <name>Buffer</name>
                <expression>
                  <dynamicfunctioncall>
                    <moduleref><name>ProdCons</name></moduleref>
                    <name>out</name>
                  </dynamicfunctioncall>
                </expression>
              </dynamicfunctioncall>
            </expression>
          </update>
        </rule>
        <rule>
          <block>
            <rule>
              <actioncall>
                <moduleref><name>Output</name></moduleref>
                <name>Write</name>
                <expression>
                  <externalfunctioncall>
                    <moduleref><name>StringManipulation</name></moduleref>
                    <name>Concat</name>
                    <expression>
                      <externalfunctioncall>
                        <moduleref><name>StringManipulation</name></moduleref>
                        <name>IntegerToString</name>
                        <expression>
```

```
                                <dynamicfunctioncall>
                                  <moduleref><name>ProdCons</name></moduleref>
                                  <name>Buffer</name>
                                  <expression>
                                    <dynamicfunctioncall>
                                      <moduleref><name>ProdCons</name></moduleref>
                                      <name>out</name>
                                    </dynamicfunctioncall>
                                  </expression>
                                </dynamicfunctioncall>
                              </expression>
                            </externalfunctioncall>
                          </expression>
                          <expression>
                            <literalstring>"\n"</literalstring>
                          </expression>
                        </externalfunctioncall>
                      </expression>
                    </actioncall>
                  </rule>
                  <rule>
                    <update>
                      <dynamicfunctioncall>
                        <moduleref><name>ProdCons</name></moduleref>
                        <name>out</name>
                      </dynamicfunctioncall>
                      <expression>
                        <derivedfunctioncall>
                          <moduleref><name>ProdCons</name></moduleref>
                          <name>next</name>
                          <expression>
                            <dynamicfunctioncall>
                              <moduleref><name>ProdCons</name></moduleref>
                              <name>out</name>
                            </dynamicfunctioncall>
                          </expression>
                        </derivedfunctioncall>
                      </expression>
                    </update>
                  </rule>
                </block>
              </rule>
            </block>
          </rule>
          <rule>
            <update>
              <dynamicfunctioncall>
                <name>step</name>
              </dynamicfunctioncall>
              <expression>
                <literalinteger>2</literalinteger>
              </expression>
            </update>
```

```
              </rule>
            </block>
          </rule>
          <expression>
            <literalinteger>2</literalinteger>
          </expression>
          <rule>
            <block>
              <rule>
                <signal>
                  <moduleref><name>ProdCons</name></moduleref>
                  <name>available</name>
                </signal>
              </rule>
              <rule>
                <update>
                  <dynamicfunctioncall>
                    <name>step</name>
                  </dynamicfunctioncall>
                  <expression>
                    <literalinteger>0</literalinteger>
                  </expression>
                </update>
              </rule>
            </block>
          </rule>
          <rule>
            <stop/>
          </rule>
        </case>
      </rule>
      <rule>
        <stop/>
      </rule>
    </conditional>
  </rule>
</transition>

</module>
```

B.1.28 ProdConsContext.mas

```
<mas>
  <name>ProdConsContext</name>
  <agents>
    <agent>
      <name>MyAgent</name>
      <moduleref><name>ProdCons</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>MyAgent</name>
  </dispatches>
```

```
</mas>
```

B.1.29 Node.mod

```
<module>

  <name>Node</name>

  <references>
  </references>

  <imports>
    <import>
      <name>Output</name>
      <global>
        <signatures>
          <actionsignature>
            <name>Write</name>
            <name>s</name>
            <type><string/></type>
          </actionsignature>
        </signatures>
      </global>
    </import>
    <import>
      <name>StringManipulation</name>
      <global>
        <signatures>
          <externalfunctionsignature>
            <name>IntegerToString</name>
            <type>
              <functional>
                <name>i</name>
                <type><integer/></type>
                <type><string/></type>
              </functional>
            </type>
          </externalfunctionsignature>
          <externalfunctionsignature>
            <name>Concat</name>
            <type>
              <functional>
                <name>s</name>
                <type><string/></type>
                <type>
                  <functional>
                    <name>r</name>
                    <type><string/></type>
                    <type><string/></type>
                  </functional>
                </type>
              </functional>
              </type>
```

```
          </externalfunctionsignature>
        </signatures>
      </global>
    </import>
  </imports>

  <global>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>

      <dynamicfunctions>

      </dynamicfunctions>

      <externalfunctions>

      </externalfunctions>

      <actions>

      </actions>

      <submachines>

      </submachines>

    </environment>

  </global>

  <local>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>

      <dynamicfunctions>

        <dynamicfunction>
          <name>f</name>
          <type>
            <node/>
          </type>
        </dynamicfunction>

      </dynamicfunctions>
```

```
<externalfunctions>

</externalfunctions>

<actions>
  <action>
    <name>PrintTree</name>
    <name>node</name>
    <type><node/></type>
    <rule>
      <with>
        <expression>
          <parametercall><name>node</name></parametercall>
        </expression>
        <name>inNode</name>
        <type><node><name>inNode</name></node></type>
        <rule>
          <block>
            <rule>
              <actioncall>
                <moduleref><name>Output</name></moduleref>
                <name>Write</name>
                <expression>
                  <externalfunctioncall>
                    <moduleref><name>StringManipulation</name></moduleref>
                    <name>Concat</name>
                    <expression>
                      <literalstring>"inNode"</literalstring>
                    </expression>
                    <expression>
                      <literalstring>"\n"</literalstring>
                    </expression>
                  </externalfunctioncall>
                </expression>
              </actioncall>
            </rule>
            <rule>
              <block>
                <rule>
                  <actioncall>
                    <name>PrintTree</name>
                    <expression>
                      <tupleprojection>
                        <expression>
                          <typecast>
                            <expression>
                              <nodecontent>
                                <expression>
                                  <withalias><name>inNode</name></withalias>
                                </expression>
                              </nodecontent>
                            </expression>
                            <type>
```

```
                    <tuple>
                      <name>left</name><type><node/></type>
                      <name>right</name><type><node/></type>
                    </tuple>
                  </type>
                </typecast>
              </expression>
              <literalinteger>0</literalinteger>
            </tupleprojection>
          </expression>
        </actioncall>
      </rule>
      <rule>
        <actioncall>
          <name>PrintTree</name>
          <expression>
            <tupleprojection>
              <expression>
                <typecast>
                  <expression>
                    <nodecontent>
                      <expression>
                        <withalias><name>inNode</name></withalias>
                      </expression>
                    </nodecontent>
                  </expression>
                  <type>
                    <tuple>
                      <name>left</name><type><node/></type>
                      <name>right</name><type><node/></type>
                    </tuple>
                  </type>
                </typecast>
              </expression>
              <literalinteger>1</literalinteger>
            </tupleprojection>
          </expression>
        </actioncall>
      </rule>
    </block>
  </rule>
</block>
</rule>
<name>leaf</name>
<type><node><name>leaf</name></node></type>
<rule>
  <actioncall>
    <moduleref><name>Output</name></moduleref>
    <name>Write</name>
    <expression>
      <externalfunctioncall>
        <moduleref><name>StringManipulation</name></moduleref>
        <name>Concat</name>
```

```
                    <expression>
                      <literalstring>"leaf"</literalstring>
                    </expression>
                    <expression>
                      <literalstring>"\n"</literalstring>
                    </expression>
                  </externalfunctioncall>
                </expression>
              </actioncall>
            </rule>
            <rule>
              <lambda/>
            </rule>
          </with>
        </rule>
      </action>
    </actions>

    <submachines>

    </submachines>

  </environment>

</local>

<init>
  <rule>
    <update>
      <dynamicfunctioncall>
        <name>f</name>
      </dynamicfunctioncall>
      <expression>
        <nodeagregate>
          <name>inNode</name>
          <expression>
            <tupleagregate>
              <expression>
                <nodeagregate>
                  <name>leaf</name>
                  <expression><undef/></expression>
                </nodeagregate>
              </expression>
              <expression>
                <nodeagregate>
                  <name>leaf</name>
                  <expression><undef/></expression>
                </nodeagregate>
              </expression>
            </tupleagregate>
          </expression>
        </nodeagregate>
      </expression>
```

```
      </update>
    </rule>
  </init>

  <transition>

    <rule>
      <block>
        <rule>
          <actioncall>
            <name>PrintTree</name>
            <expression>
              <dynamicfunctioncall>
                <name>f</name>
              </dynamicfunctioncall>
            </expression>
          </actioncall>
        </rule>
        <rule>
          <stop/>
        </rule>
      </block>
    </rule>

  </transition>

</module>
```

B.1.30 NodeContext.mas

```
<mas>
  <name>NodeContext</name>
  <agents>
    <agent>
      <name>TheAgent</name>
      <moduleref><name>Node</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>TheAgent</name>
  </dispatches>
</mas>
```

B.1.31 Abstractions.mod

```
<module>

  <name>Abstractions</name>

  <references>
  </references>

  <imports>
```

```
    <import>
      <name>Output</name>
      <global>
        <signatures>
          <actionsignature>
            <name>Write</name>
            <name>s</name>
            <type><string/></type>
          </actionsignature>
        </signatures>
      </global>
    </import>
    <import>
      <name>StringManipulation</name>
      <global>
        <signatures>
          <externalfunctionsignature>
            <name>IntegerToString</name>
            <type>
              <functional>
                <name>i</name>
                <type><integer/></type>
                <type><string/></type>
              </functional>
            </type>
          </externalfunctionsignature>
          <externalfunctionsignature>
            <name>Concat</name>
            <type>
              <functional>
                <name>s</name>
                <type><string/></type>
                <type>
                  <functional>
                    <name>r</name>
                    <type><string/></type>
                    <type><string/></type>
                  </functional>
                </type>
              </functional>
              </type>
          </externalfunctionsignature>
        </signatures>
      </global>
    </import>
  </imports>

  <global>

    <environment>

      <staticandderivedfunctions>
```

```
      </staticandderivedfunctions>

      <dynamicfunctions>

      </dynamicfunctions>

      <externalfunctions>

      </externalfunctions>

      <actions>

      </actions>

      <submachines>

      </submachines>

    </environment>

  </global>

  <local>

    <environment>

      <staticandderivedfunctions>

      </staticandderivedfunctions>

      <dynamicfunctions>

        <dynamicfunction>
          <name>f</name>
          <type>
            <abstraction>
              <type><integer/></type>
            </abstraction>
          </type>
        </dynamicfunction>

        <dynamicfunction>
          <name>g</name>
          <type>
            <abstraction>
              <type><integer/></type>
            </abstraction>
          </type>
        </dynamicfunction>

      </dynamicfunctions>

      <externalfunctions>
```

```
</externalfunctions>

<actions>

  <action>
    <name>PrintSquare</name>
    <name>i</name>
    <type><integer/></type>
    <rule>
      <actioncall>
        <moduleref><name>Output</name></moduleref>
        <name>Write</name>
        <expression>
          <externalfunctioncall>
            <moduleref><name>StringManipulation</name></moduleref>
            <name>Concat</name>
            <expression>
              <externalfunctioncall>
                <moduleref><name>StringManipulation</name></moduleref>
                <name>IntegerToString</name>
                <expression>
                  <mult>
                    <expression>
                      <parametercall><name>i</name></parametercall>
                    </expression>
                    <expression>
                      <parametercall><name>i</name></parametercall>
                    </expression>
                  </mult>
                </expression>
              </externalfunctioncall>
            </expression>
            <expression>
              <literalstring>"\n"</literalstring>
            </expression>
          </externalfunctioncall>
        </expression>
      </actioncall>
    </rule>
  </action>

  <action>
    <name>PrintDouble</name>
    <name>i</name>
    <type><integer/></type>
    <rule>
      <actioncall>
        <moduleref><name>Output</name></moduleref>
        <name>Write</name>
        <expression>
          <externalfunctioncall>
            <moduleref><name>StringManipulation</name></moduleref>
```

```
                <name>Concat</name>
                <expression>
                  <externalfunctioncall>
                    <moduleref><name>StringManipulation</name></moduleref>
                    <name>IntegerToString</name>
                    <expression>
                      <add>
                        <expression>
                          <parametercall><name>i</name></parametercall>
                        </expression>
                        <expression>
                          <parametercall><name>i</name></parametercall>
                        </expression>
                      </add>
                    </expression>
                  </externalfunctioncall>
                </expression>
                <expression>
                  <literalstring>"\n"</literalstring>
                </expression>
              </externalfunctioncall>
            </expression>
          </actioncall>
        </rule>
      </action>

    </actions>

    <submachines>

    </submachines>

  </environment>

</local>

<init>
  <rule>
    <block>
      <rule>
        <update>
          <dynamicfunctioncall>
            <name>f</name>
          </dynamicfunctioncall>
          <expression>
            <abstractionreference>
              <name>Abstractions</name>
              <expression><self/></expression>
              <name>PrintSquare</name>
            </abstractionreference>
          </expression>
        </update>
      </rule>
```

```
      <rule>
        <update>
          <dynamicfunctioncall>
            <name>g</name>
          </dynamicfunctioncall>
          <expression>
            <abstractionreference>
              <name>Abstractions</name>
              <expression><self/></expression>
              <name>PrintDouble</name>
            </abstractionreference>
          </expression>
        </update>
      </rule>
    </block>
  </rule>
</init>

<transition>

  <rule>
    <block>
      <rule>
        <stop/>
      </rule>
      <rule>
        <block>
          <rule>
            <abstractioncall>
              <expression>
                <dynamicfunctioncall>
                  <name>f</name>
                </dynamicfunctioncall>
              </expression>
              <expression>
                <literalinteger>4</literalinteger>
              </expression>
            </abstractioncall>
          </rule>
          <rule>
            <abstractioncall>
              <expression>
                <dynamicfunctioncall>
                  <name>g</name>
                </dynamicfunctioncall>
              </expression>
              <expression>
                <literalinteger>4</literalinteger>
              </expression>
            </abstractioncall>
          </rule>
        </block>
      </rule>
```

```
      </block>
    </rule>

  </transition>

</module>
```

B.1.32 AbstractionsContext.mas

```
<mas>
  <name>AbstractionsContext</name>
  <agents>
    <agent>
      <name>TheAgent</name>
      <moduleref><name>Abstractions</name></moduleref>
    </agent>
  </agents>
  <dispatches>
    <name>TheAgent</name>
  </dispatches>
</mas>
```