ITALO GIOVANI A. STEFANI

MÉTODO DE REFINAMENTO MACHĬNA

Belo Horizonte
Março de 2007

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

MÉTODO DE REFINAMENTO MACHĬNA

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.

ITALO GIOVANI A. STEFANI

Belo Horizonte
Março de 2007

Resumo

O modelo de Máquina de Estado Abstrata (ASM - *Abstract State Machine*) vem sendo amplamente utilizado para especificação formal de diversos tipos de sistemas devido ao seu alto grau de abstração e rigor matemático, o que facilita compreender o sistema modelado e verificar formalmente suas propriedades. Pode-se utilizar uma linguagem baseada no modelo ASM para escrever uma especificação em alto nível, conhecida como Modelo Básico, e posteriormente submetê-la a um processo de transformação baseado no Método de Refinamento ASM para obter a implementação validada e verificada.

O principal objetivo do trabalho Método de Refinamento Machĭna (MRM) é propor um método de especificação em alto nível que represente aspectos de ASM e com a possibilidade de validar e verificar o modelo construído independente da implementação. O processo de refinamento permite obter, automaticamente, o código executável em Machĭna e realizar a verificação utilizando a ferramenta NuSMV. Assim, pode-se verificar automaticamente a implementação de acordo com a especificação.

Abstract

The Abstract State Machine (ASM) has been used as language for formal specification to various systems due to the high level of abstraction and the mathematical rigor. It facilitates to understand the modeled system and to formally verify properties. Using an ASM language, is possible to make a high level specification, called Ground Model, to be transformed, based on The ASM Refinement Method, in an executable code, validated and verified.

The main goal of The Machĭna Refinement Method is to propose a high level specification method that represents aspects of ASM with the possibility to validate and to verify the builded model independent of the implementation. The refinement process automatically generates the executable Machĭna code and to carry through the verification using the NuSMV tool. Thus, the implementation can be automatically verified in accordance with the specification.

Agradecimentos

Agradeço aos meus pais, Italo e Maria do Carmo, pelo apoio e carinho; ao professor Bigonha pela confiança; aos amigos do LLP pela companhia, e à Linda pelo carinho, compreensão, apoio e torcida.

Sumário

Lista de Figuras

Capítulo 1

Introdução

O trabalho Método de Refinamento Machĭna tem como objetivo apresentar a proposta de um método para especificação em alto nível utilizando uma linguagem baseada em diagramas e considerando aspectos do modelo ASM. O método descrito permite validar o sistema durante a fase de projeto e, por meio de transformações, obtém-se o código Machĭna executável e realiza-se automaticamente a verificação. Dessa forma, as tarefas de solucionar um problema e identificar possíveis erros de projeto são mais facilmente realizadas. Este capítulo contextualiza o processo de refinamento e cita as diferentes abordagens para o tema. Por fim, é apresentado o resumo da proposta do Método de Refinamento Machĭna.

1.1 Método de Refinamento

As dificuldades encontradas na especificação e desenvolvimento de sistemas com os conhecidos métodos de desenvolvimento e implementação localizam-se na enorme facilidade de se cometer erros nos vários estágios que devem ser cumpridos [Ber02]. Esse fato tem motivado pesquisas sobre diversos métodos nos quais a técnica de implementação ou desenvolvimento seja separada completamente dos estágios de especificação e solução do problema.

Podem ser citados três principais estágios. No primeiro, são realizadas análises sobre o problema no qual é possível identificar as responsabilidades que o futuro sistema deverá suportar, suas propriedades, as partes que o compõem e os possíveis pontos de interação com o ambiente onde o sistema estará presente. Pode-se dizer que este estágio apresenta o problema que se deseja resolver e o motivo pelo qual o sistema é necessário.

No segundo estágio, é realizada a especificação, que consiste na representação em alto nível do sistema onde são consideradas apenas características, propriedades e funcionalidades ligadas à solução do problema. Isso significa abstrair os detalhes de imple-

mentação devido às características de linguagem e da plataforma de desenvolvimento. Deve-se validar as funcionalidades modeladas de acordo com as que foram identificadas no primeiro estágio e previstas para o sistema. Portanto, um método de desenvolvimento deve prover recursos que permitam a validação durante o estágio de especificação. Dessa forma, pode-se trabalhar isoladamente na solução do problema dando-lhe a devida atenção e garantir que as necessidades reconhecidas na análise do problema sejam satisfeitas.

No último estágio, ocorre a implementação do modelo que foi especificado no segundo estágio. A especificação é transformada em um código executável onde, então, são consideradas as características de plataforma e linguagem, obtendo uma representação do sistema com as funcionalidades e propriedades equivalentes às definidas anteriormente. Deve-se seguir um método sistemático para realizar a implementação que seja possível garantir formalmente a correta correspondência entre o modelo abstrato e sua representação executável.

Em qualquer um dos estágios, é possível cometer erros que podem resultar em uma funcionalidade não identificada, ausência de um determinado comportamento na especificação, desrespeito à uma propriedade ou codificação equivocada de funcionalidades. Qualquer que seja o erro existente, o sistema pode não corresponder às expectativas ou, pior ainda, gerar resultados incorretos. A separação bem definida dos estágios de desenvolvimento permite identificar e tratar os erros com maior precisão e melhores condições. Por exemplo, uma implementação errada de uma funcionalidade pode fazer com que desenvolvedores percam um tempo precioso procurando pelo erro no lugar errado, na especificação por exemplo, caso os estágios não estejam bem definidos.

Além dos três estágios mencionados, a etapa de verificação é fundamental para garantir que a implementação esteja correta de acordo com o modelo abstrato. A verificação deve estar presente durante todo o processo de implementação, assim é possível assegurar que cada transformação realizada em direção ao código executável preserva as propriedades conforme descrito na especificação.

Uma maneira muito eficiente de abordar essas etapas de desenvolvimento é utilizando um método de refinamento. Refinamento é um processo no qual uma representação de alto nível é transformada em outra de nível mais baixo, visando um código executável. Essa transformação ocorre por meio do uso de regras que substituem construções abstratas por outras com maiores detalhes. As regras são definidas a partir dos tipos de refinamento elaborados em cada método. Comumente, as regras, além de realizarem as passagens de níveis, geram provas obrigatórias para garantir a correta correspondência entre o modelo abstrato e o modelo refinado. Os tipos de refinamento são padrões do método normalmente reconhecidos pela experiência com o uso, e por isso é um campo ainda em aberto para pesquisas em muitos dos métodos de refinamento.

É desejável que esse processo seja realizado de forma automatizada, com a menor participação possível dos desenvolvedores, a fim de não somente evitar erros de interpretação ou implementação, mas também minimizar o trabalho humano, acelerando o processo de codificação. Por isso, os padrões reconhecidos no método, que dão origem aos tipos de refinamento, são de fundamental importância para a automatização tanto da aplicação das regras quanto para verificação das propriedades do sistema. Sabe-se que a completa automatização é difícil por causa das diferentes interpretações que podem ser dadas na solução de um problema [Bör03a, BS03]. As provas podem ser feitas utilizando provadores de teorema, mas as estratégias de provas devem ser informadas pelo desenvolvedor [Wie06]. Uma outra possibilidade é utilizar verificadores de modelos, que são ferramentas de auxílio para verificação automática [McM93].

O principal resultado de um método de refinamento é a implementação do sistema em um código executável, chamado de código alvo. O código executável faz parte do método do refinamento como artefato de implementação do sistema que foi especificado e não deve interferir no método como um todo para preservar a propriedade de independência de plataforma. Elementos do ambiente de desenvolvimento, como plataforma ou linguagem de desenvolvimento, devem estar presente no método de refinamento de forma discreta para não gerar preocupações com comportamentos indesejáveis quando se trabalha na especificação em alto nível.

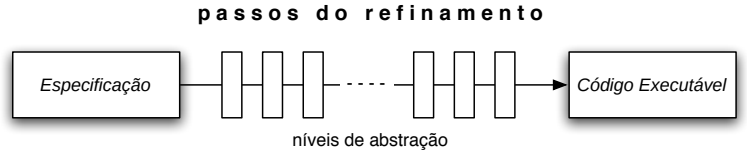


Figura 1.1: Passos do refinamento.

A partir de uma especificação de alto nível, o processo de refinamento passa por vários níveis de abstração até a obtenção do código alvo, como mostra a Figura 1.1 de forma bem simples. Embora o objetivo do refinamento seja obter o código executável, podem existir códigos alvo que sejam entradas de outros processos de refinamento, caracterizando um processo de refinamento intermediário. A união destes processos intermediários resultam na implementação correta e completa do sistema especificado inicialmente no nível mais abstrato. Por exemplo, é possível modelar um sistema em UML e, com o uso de uma ferramenta adequada, o projeto pode ser transformado em um esqueleto de código Java posteriormente completado pelo desenvolvedor. Em seguida, o código Java é compilado para sua representação em *ByteCode*, que, por fim, é interpretada pela JVM executando instruções de máquina física correspondente aos

comandos recebidos. Então, existem quatro processos de refinamento, sendo o segundo com intervenção humana, onde a saída de um é a entrada para outro, ou o código alvo de um é a especificação em alto nível do seguinte, conforme ilustrado na Figura 1.2.

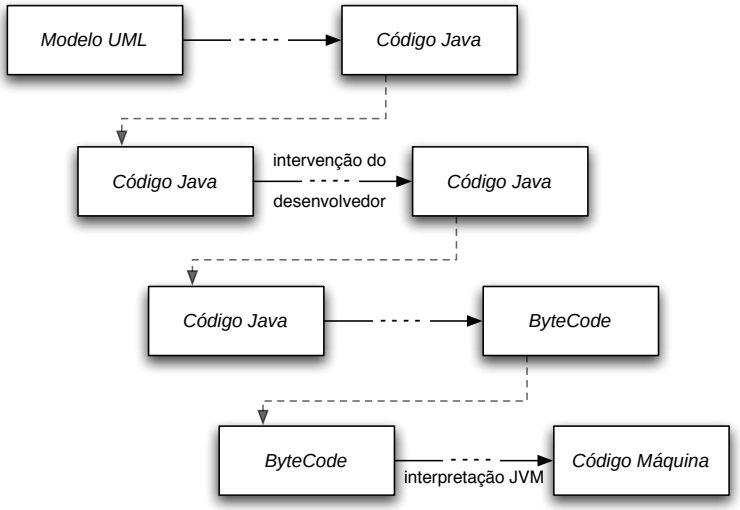


Figura 1.2: Vários processos de refinamento.

1.2 Estado da Arte

O refinamento passo a passo (*Stepwise Refinement*) foi inicialmente proposto por Dijkstra [Dij68, Dij76] e Wirth [Wir71]. Propostas seguintes descrevem um arcabouço para diferentes métodos de refinamento, que se diferenciam na linguagem de especificação em alto nível, regras de refinamento, método de verificação e no código gerado.

Um método bastante conhecido foi descrito, em paralelo, por Back [Bac80], Morris [Mor87] e Morgan [Mor94], que apresentam trabalhos semelhantes em refinamento de cálculos (*Refinement Calculus*). São aplicadas as teorias de refinamento e as regras são definidas em termos de cálculo de predicado. Existem várias ferramentas que propõem um suporte a este método, como por exemplo o trabalho de Manuela Xavier e Ana Cavalcanti [XC05], que descrevem a ferramenta *Refine* e a compara com outras de características semelhantes. O trabalho de Faitelson, Welch e Davies [FWD05] também é realizado sobre *Refinement Calculus* e utiliza as notações formais Z e B como base para descrever operações em modelos de objetos, mais precisamente, armazenamento de dados os quais são organizados como coleções de objetos.

Descrições em esquemas, ou desenhos, gráficos proporcionam um alto nível de abstração. A UML, por exemplo, é utilizada para este fim e foi criado um método, chamado de Processo Unificado (*Unified Process* [AN05]), que descreve como transformar

a descrição em alto nível em elementos mais detalhados. Dentro deste processo, existem abordagens direcionadas a diversas linguagens, como Java, C, C++, C#, Fortran, Visual Basic e COBOL. Uma ferramenta de desenvolvimento de sistemas bastante difundida chama-se *Rational* [IBM07], onde a especificação UML é transformada em código Java.

Todos estes métodos de refinamento são descritos com o objetivo de acrescentar determinado formalismo à proposta de Dijkstra [Dij68, Dij76], tornando a especificação, validação e verificação de sistemas uma tarefa mais simples por meio de um método sistemático. A maioria das descrições de refinamento são baseadas no princípio da substituição[1] [DB01, p. 47].

> [*É aceitável substituir um programa por outro, contando que seja impossível para um usuário do programa perceber que a substituição ocorreu.*]

Vários conceitos de refinamento são definidos na literatura para atender a este princípio, e como resultado, são restringidos em várias formas, limitando sua utilização. Este fato foi constatado por Börger [Bör03a, BS03] ao analisar os principais métodos de refinamento [Abr96a, Bac81, BvW98, DB01, dRE98, FL98, WD96].

A vantagem do Método de Refinamento ASM [Bör03a, BS03] é que ele fornece um arcabouço que agrega conceitos destes principais métodos de refinamento. A liberdade de abstração fornecida em ASM provê o instrumento necessário para melhor correspondência entre a máquina abstrata e a refinada, de forma que ambas as execuções possam ser observadas quanto a equivalência. O foco do método ASM não é prover noções genéricas de refinamento que seja aplicável a qualquer contexto, mas sim fornecer suporte à utilização disciplinada do refinamento no qual captura corretamente e documenta explicitamente o projeto desejado.

1.3 Objetivos do Trabalho

O modelo de Máquina de Estado Abstrata (ASM - *Abstract State Machine*) [Gur95] vem sendo amplamente utilizado para especificação formal de diversos tipos de sistemas devido ao seu alto grau de abstração e rigor matemático, o que facilita compreender o sistema modelado e verificar formalmente suas propriedades. Pode-se utilizar uma linguagem baseada no modelo ASM para escrever a especificação em alto nível, conhecida como Modelo Básico, que posteriormente é submetida a um processo baseado no

[1] Principle of substituitivity: it is acceptable to replace one program by another, provided it is impossible for a user of the programs to observe that the substitution has taken place.

Método de Refinamento ASM [Bör03a] para obter a implementação validada e verificada.

Com base nestes fatos, o principal objetivo do trabalho Método de Refinamento Machĭna (MRM) é propor um método de especificação em alto nível que represente aspectos de ASM e com a possibilidade de validar e verificar o modelo construído independente da implementação. O processo de refinamento permite obter automaticamente o código executável verificado e validado de acordo com a especificação.

O Método de Refinamento Machĭna é composto de: uma linguagem de especificação para escrita do Modelo Básico chamada de Linguagem de Modelagem Machĭna; um conjunto de definições de tipos e regras de refinamento com propriedades definidas a serem comprovadas; suporte à validação manual durante a fase de projeto; e suporte à verificação automática. Como linguagem executável alvo utiliza-se Machĭna, que é uma linguagem ASM desenvolvida no Laboratório de Linguagens de Programação da Universidade Federal de Minas Gerais [BTIB05].

A Linguagem de Modelagem Machĭna (LMM) possui recursos gráficos para especificação em alto nível e é definida por uma semântica bem definida, ou seja, livre de ambigüidades. Baseada em diagramas de controle de fluxo e definição de tipos abstratos de dados, a LMM possibilita a validação ainda durante a especificação e facilita a participação de projetistas, desenvolvedores e possíveis usuários do sistema, que nem sempre são da área da Computação. A LMM é composta de elementos que permitem trabalhar em vários níveis de abstração, representar elementos do mundo real, modelar comportamento de agentes e relações entre eles. Além disso, é possível descrever preocupações ortogonais simples de forma modular e definir propriedades inerentes ao problema.

Em LMM, escreve-se o Modelo Básico definindo os membros envolvidos no problema, características, seus comportamentos e a relação entre eles. O desenvolvimento é feito em partes e permite obter visão local e global do problema. Então, o Modelo Básico é refinado com a aplicação iterativa das regras definidas no método proposto a partir dos tipos de refinamento identificados. São obtidos diversos níveis de abstração até o código Machĭna.

Ao longo do processo de refinamento de MRM, é formada uma coleção de propriedades que podem ser divididas em grupos, de acordo com as garantias que produzem. O primeiro é formado pelas propriedades que provam o correto refinamento, extraídas das aplicações das regras. O segundo grupo é obtido pela união das propriedades definidas na especificação em alto nível com as propriedades implícitas à especificação LMM, que também são extraídas a partir das aplicações de regras, para realizar a verificação do Modelo Básico.

Faz parte do objetivo de MRM realizar a verificação da especificação de forma

automática. Portanto, o Modelo Básico é transformado automaticamente no formato de entrada para o verificador de modelos NuSMV [CCG+02, CCO+06], onde é possível conferir as propriedades do sistema.

Figura 1.3 ilustra o escopo deste trabalho, onde o Modelo Básico é construído a partir da LMM e, com o uso do método de refinamento, obtém-se o código Machĭna e a representação em NuSMV. O resultado da verificação, junto ao código executável gerado, ao Modelo Básico e sua validação, formam a documentação completa de um sistema especificado.

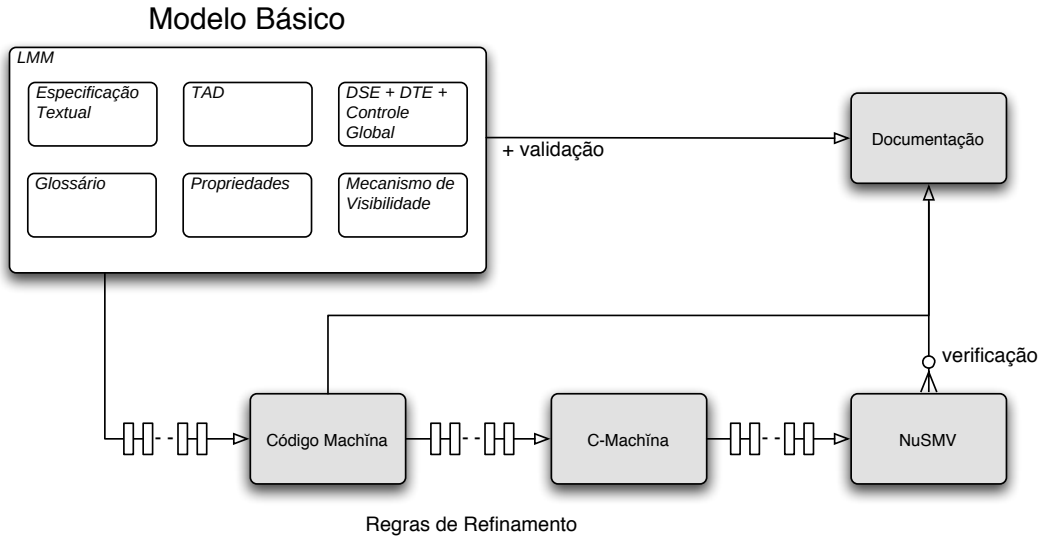


Figura 1.3: Esquema do Método de Refinamento Machĭna

Para avaliação do trabalho proposto, a implementação do Método de Refinamento Machĭna abrange a representação interna do Modelo Básico, aplicação das regras e tipos de refinamento para obter o código Machĭna correspondente e sua transformação, em dois passos, para uma entrada de NuSMV. A ferramenta desenvolvida representa o núcleo principal de Método de Refinamento Machĭna, e extensões para ambiente gráfico podem ser inseridas posterioremnte de forma simples.

Por meio do Método de Refinamento Machĭna, pretende-se obter as seguintes contribuições:

- definição de uma linguagem de especificação de alto nível baseada em diagramas, voltada ao modelo de Máquina de Estado Abstrata, intuitiva e com alto poder de expressão;
- separação bem definida dos níveis de abstrações utilizados na especificação;
- facilidade na validação do problema especificado;

- definição de um método de refinamento para a linguagem Machĭna, no qual o processo de transformações é realizado automaticamente;
- verificação transparente e automática de sistemas descritos em Machĭna.

1.4 Organização Deste Texto

Capítulo 2 apresenta o modelo formal ASM, seu método de refinamento e a linguagem de especificação formal Machĭna, que é baseada no modelo ASM. O Método de Refinamento Machĭna é descrito no Capítulo 3, onde são definidos os elementos da linguagem proposta e as regras de refinamento. No Capítulo 4, apresenta-se o método de verificação automática para o Método de Refinamento Machĭna, e a validação deste trabalho é realizada no Capítulo 5. Por fim, os resultados obtidos e propostas de trabalhos futuros são resumidos no Capítulo 6.

Capítulo 2

Modelo Formal ASM

Máquina de Estado Abstrata (ASM - *Abstract State Machine*) é um modelo formal de especificação e verificação de sistemas proposto por Gurevich [Y. 85, Gur95] e aperfeiçoado por Börger [BS03]. Baseado em conceitos conhecidos da matemática, ASM proporciona facilidade de compreensão e permite demonstrar formalmente propriedades contidas na especificação. Este capítulo apresenta a definição do modelo ASM e, em seguida, são apresentados os conceitos do método de refinamento ASM [BS03]. Por fim, é apresentada a definição da linguagem de especificação formal Machĭna, que é baseada no modelo ASM.

2.1 Máquina de Estado Abstrata

Uma Máquina de Estado Abstrata (ASM) é um modelo de transição de estado que consiste em uma máquina abstrata formada por nomes de funções e relações, que definem seu vocabulário Υ. Todo vocabulário possui implicitamente os símbolos booleanos *true* e *false* e seus operadores usuais, assim como o sinal de igualdade e o valor *undef*, indicando valor não definido.

Um conjunto U é chamado de universo de um nome do vocabulário se todos os seus elementos satisfazem uma relação, ou podem ser associados a uma função, na qual o nome se refere. Um estado S é um conjunto X, denominado superconjunto de S, constituído dos nomes do vocabulário e suas interpretações em um determinado instante. Se f é um nome de uma função de aridade r, então f é interpretada como

$$f : X^r \rightarrow X$$

Se f é um nome de relação de aridade r, então f é interpretada como

$$f : X^r \rightarrow \{true \mid false\}$$

Sendo U o nome de relação pertencendo a Υ, então o conjunto

$$U = \{x : U(x) = true\}$$

é um universo contido em X. Neste caso,

$$U(x) = true$$

é equivalente a

$$x \in U$$

Por exemplo, suponha o nome da relação *Primos* como pertencente ao vocabulário Υ, então o conjunto

$$Primos = \{x : Primos(x) = true\}$$

é um universo contido em X.

Alterações na interpretação dos nomes no vocabulário são provocadas pelas atualizações, contidas na regra de transição, que associa um nome a um valor e produz um novo estado. Uma das regras de transição pode ser definida como um conjunto de construções

$$if\ cond\ then\ update$$

onde *cond* é uma expressão booleana e *update* é uma lista de atualização, que tem seu efeito percebido apenas no estado seguinte. Portanto, seja i o estado corrente e R a regra de transição, as atualizações pertencentes a R realizadas no estado i, somente terão validade no estado j seguinte.

O par (f, v) indica uma atualização no nome f de aridade r com o valor v, e será consistente se existir na lista de atualizações apenas um par (f, v) ou, na presença de dois pares (f, v) e (f, v'), onde $v = v'$. Caso contrário a atualização é inconsistente.

Uma inconsistência na lista de atualizações faz com que a transição também seja inconsistente, e o modelo ASM de Gurevich deixa em aberto qual o tratamento a ser dado neste caso. A definição de Börger [BS03] estabelece que uma transição inconsistente não permite a obtenção do estado seguinte. Neste caso a máquina interrompe a execução em uma situação de erro.

Uma característica notável de ASM é a ausência de comandos de repetição, pois este conceito é implícito ao modelo. Uma computação da máquina consiste em executar repetidas vezes a regra de transição obtendo sempre um novo estado. Pode-se definir diferentes critérios para a terminação natural da execução da máquina. Por exemplo, se não for mais possível executar nenhuma atualização; ou a máquina determina uma atualização vazia; ou os estados não mais se alteram na presença de atualizações.

2.1.1 Regras Básicas

Definem-se as seguintes regras básicas para construir a regra de transição.

Regra de Atualização

Uma atualização é da forma

$$f(t_1 \dots t_r) := t$$

onde f é um nome de função r-ária e t, $t_1 \dots t_r$ são termos. No estado seguinte, f será interpretada como uma função que no ponto $t_1 \dots t_r$ tem o valor t.

Regra Bloco

A regra bloco é uma regra da forma

$$R_1, R_2$$

onde R_1 e R_2 são regras de atualização. A semântica é executar R_1 e R_2 em paralelo, sendo o efeito de suas atualizações percebidas apenas no estado seguinte, desde que não existam atualizações inconsistentes. É importante ressaltar que não existe sequenciamento de regras em uma transição. A ordem relativa entre R_1 e R_2 não é diferenciada no modelo.

Regra Condicional

A regra condicional permite a execução de R_1 ou R_2 dependendo da avaliação de uma expressão booleana g e é dada da seguinte forma:

$$if\ g\ then\ R_1\ else\ R_2\ end$$

2.1.2 Especificação ASM

Uma especificação ASM é definida pela tupla

$$(\Upsilon, A, S_0, R)$$

onde Υ é o vocabulário com os nomes de funções e relações, A é o conjunto de estados, S_0 é o conjunto de estados iniciais tal que $S_0 \subset A$ e, por fim, R é a regra de transição. A instância de uma especificação ASM é chamada de agente.

A seguir é apresentado um exemplo para a função fatorial, extraído de [BTIB05]. Define-se o vocabulário contendo os nomes i e fat, e seus valores iniciais conforme

S_0. Em seguida, a regra de transição R é constituída de um bloco contendo duas atualizações.

$$\begin{aligned} \Upsilon &= \{i, fat\} \\ S_0 &= i \leftarrow 0 \\ &\quad fat = \lambda x.x = 0 \rightarrow 1, undef \\ R &= fat(i+1) := (i+1) * fat(i) \quad , \quad i := i+1 \end{aligned}$$

Ao executar a regra R a partir do estado inicial S_0, obtêm-se os seguintes passos:

Estado	0	1	2	3	4	$\cdots$	n
S_0	1	*undef*	*undef*	*undef*	*undef*	$\cdots$	*undef*
S_1	1	1	*undef*	*undef*	*undef*	$\cdots$	*undef*
S_2	1	1	2	*undef*	*undef*	$\cdots$	*undef*
S_3	1	1	2	6	*undef*	$\cdots$	*undef*
S_4	1	1	2	6	24	$\cdots$	*undef*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
S_n	1	1	2	6	24	$\cdots$	$n!$

Ao executar a regra no estado S_1, obtém-se um novo estado S_2, no qual i é interpretado como 2 e a interpretação de f, no ponto 2, passa a ser igual a 2. Da mesma forma, obtemos os estados S_3, S_4 e assim por diante. Intuitivamente, no estado S_k, $k \geq i$, a interpretação de f é uma função que, aplicada a i, retorna $i!$. Esta interpretação pode ser provada por meio de indução matemática sobre o conjunto dos naturais.

2.1.3 Regras Avançadas

A seguir, são apresentadas outras construções que complementam uma regra de transição.

Funções Externas

Os sistemas especificados em ASM podem se relacionar com o ambiente, por exemplo, entrada de dados fornecidas pelo usuário ou saída de resultados. Para isso, definem-se as funções externas, que se comportam como oráculos, ou seja, uma especificação fornece os argumentos e o oráculo retorna o resultado de forma transparente para o sistema.

Regra Import

É possível estender o vocabulário Υ do estado S utilizando a regra *import*, que possui a seguinte forma:

$$import\ v\ R_0\ endimport$$

para v sendo o novo nome em Υ e R_0 a regra a ser executada onde v aparece livre. O exemplo a seguir estende Υ com um novo nodo v de uma árvore binária e define seus filhos da esquerda e direita como indefinidos.

```
import v
  Nodo(v)      := true
  Esquerda(v)  := undef
  Direita(v)   := undef
endimport
```

Regra Choose

ASM permite a execução não determinística por meio da regra *choose* com a seguinte forma:

$$choose\ v\ in\ U\ satisfying\ c\ R_0\ endchoose$$

Seu significado é escolher não deterministicamente um elemento v pertencente ao universo U que satisfaça a condição c. Então, v é utilizado na regra R_0, onde aparece livre. Por exemplo, o trecho a seguir atribui o valor 1 a um nodo do universo *Nodos* de uma árvore.

```
choose v in Nodos
  Valor(v) := 1
endchoose
```

Neste caso, a condição foi omitida, então a escolha de v é qualquer um dentre os elementos pertencentes à *Nodos*.

Regra Var

A regra *var* tem a seguinte forma:

$$var\ v\ ranges\ over\ U\ R_0\ endvar$$

Seu significado é executar a regra R_0 para cada $v \in U$ de forma independente e em paralelo. O nome v deve aparecer livre dentro de R_0. Essa regra equivale a criar uma

instância de R_0 para cada $v \in U$. Por exemplo:

```
var v ranges over UmDoisTres
  f(v) := 0
envar
```

onde *UmDoisTres* é o conjunto {1,2,3}, é equivalente a fazer

```
f(1) := 0,
f(2) := 0,
f(3) := 0
```

2.1.4 Exemplo: Sistema Operacional

O exemplo a seguir, extraído de [BTIB05] modela o núcleo de um sistema operacional simples e tem o objetivo de ilustrar a utilização das regras apresentadas. Para isso, especificam-se as seguintes operações:

- criação de um novo processo;
- escalonamento de um processo;
- difusão de uma mensagem para todos os processos ativos;
- recebimento de um sinal de interrupção.

O vocabulário Υ para o sistema operacional pode ser definido da seguinte forma:

- O universo *Processes*, é o conjunto dos processos criados no sistema operacional;
- A função *id*, associa cada processo à sua identificação;
- A função *numProc*, contabiliza o número de processos criados;
- A função *messages*, associa os processos às suas caixas de mensagens;
- A função *owner*, associa cada recurso ao processo que está utilizando;
- A relação *waiting*, relaciona recursos aos processos que estão esperando para utilizá-los.

Assim, pode-se definir as regras de atualização conforme a seguir:

Criação de novo processo p:

```
import p
  Processes(p) := true,
  id(p)    := numProc + 1 ,
  numProc := numProc + 1
endimport
```

Escalonamento de processos;

```
choose p in Processes
satisfying waiting(p, resource)
  owner(resource) := id(p)
endchoose
```

Difusão de mensagem:

```
var p ranges over Processes
  if ativo(p) then
    messages(id(p)) := append(messages(id(p)),msg)
  endif
endvar
```

Recebimento de interrupção:

```
if interrupted then
  // catch
else
  // continue operations
endif
```

Os conceitos e regras apresentados resumem o modelo ASM proposto por Gurevich e aperfeiçoado por Börger. Como foi mostrado nos exemplos, é possível especificar sistemas em alto nível de forma simples e elegante devido ao grande poder de expressão de ASM assim como sua facilidade de compreensão.

2.1.5 ASM Multiagentes

Uma ASM Multiagente M [Gur95], também conhecida como ASM Distribuída, contém um número finito agentes computacionais que executam concorrentemente um número

finito de programas. A cada agente está associado um programa. Formalmente, uma ASM Multiagente M é uma tupla

$$(\Upsilon_M, A, C_0, P)$$

tal que Υ_M é um vocabulário que contém os nomes de função que aparecem nas regras de P, com exceção de *Self*. Pertencem a Υ_M os nomes de funções de zero argumento que representam os nomes de módulos (elementos de P) e uma função unária, *Mod*, que representa a relação entre agentes e módulos. Um elemento a é um *agente* em um dado estado S se houver um nome de módulo m tal que

$$S \models Mod(a) = m$$

Da mesma forma, o programa de a é

$$Prog(a) = \pi_m$$

A é um conjunto de estados e C_0 é uma coleção de estados denominados estados iniciais de M. Todos os elementos de C_0 compartilham as mesmas interpretações dos nomes de função de Υ_M, com exceção de *Self*, que é interpretado como o agente associado ao estado.

Por fim, P é um conjunto finito indexado de programas π_m (os módulos), identificados pelos nomes de módulos $m \in \Upsilon$; cada programa pertencente a P é uma regra de transição.

A função especial de zero argumento *Self* permite a auto-identificação de agentes. *Self* é interpretado como a por cada agente a e pode ser usada, por exemplo, para modelar algum estado local dos agentes. Diz-se que um agente a realiza uma transição em S se o conjunto de atualizações

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

é disparado em S, isto é, se a regra que forma o programa do agente a for disparada no estado $View_a(S)$, que é a expansão do estado S, em que a variável *Self* é interpretada como a. Sobre esta definição de transição, pode-se definir diversos tipos de execução.

A seguir, tem-se a definição de execução parcialmente ordenada, que é uma importante noção de computação distribuída.

Gurevich [Gur95] define uma execução parcialmente ordenada ρ de uma ASM Multiagentes M como sendo uma tripla (M, A, σ) que satisfaz as seguintes condições:

1. M é um conjunto parcialmente ordenado, no qual todos os conjuntos de transição

$\{y : y \preceq x\}, x \in M$, são finitos: os elementos de M representam transições realizadas pelos vários agentes durante a execução. Duas transições x e y são tais que $y \prec x$, se a transição x iniciou após o término de y.

2. A é uma função sobre M tal que todo conjunto não-vazio $\{x : A(x) = a\}$ é linearmente ordenado: $A(x)$ é o agente que realiza a transição x (supõe-se que as transições de um único agente sejam linearmente ordenados).

3. σ é uma função que associa um estado de M a cada segmento inicial 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. Um segmento inicial de um conjunto parcialmente ordenado P é uma sub-estrutura X de P tal que

$$x \in X \wedge y < x \Rightarrow y \in X.$$

4. 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 $\sigma(Y)$.

Esta definição garante que não haverá inconsistências na execução de dois ou mais agentes, pois cada transição é executada por um agente de maneira atômica.

2.1.6 Máquina de Estado Finita

O modelo de Máquina de Estado Abstrata (*ASM - Abstract State Machine*) [Gur95] descreve uma classe específica de máquina na qual os estados são finitos. Essa classe recebe o nome de Máquina de Estado Finita (*FSM - Finite State Machine*) [Bör03a, Bör03b, BS03, E. 99] e ela representa um diagrama de estrutura de controle que possibilita o manuseio de estrutura de dados que definem o sistema.

O diagrama FSM é baseado no diagrama de atividades da UML [RJB04, BRJ05], e sua notação é mostrada a seguir. Um círculo rotulado representa um estado da máquina, e setas entre os estados representam o fluxo de controle. Um hexágono representa uma condição (*cond*) e um retângulo representa a regra (*rule*) a ser executada para a transição de estados. Um retângulo associado a um hexágono significa que a execução da regra está condicionada à restrição representada. Um retângulo cinza rotulado representa o nome da transição.

O esquema geral de uma transição em um diagrama FSM é mostrado na Figura 2.1. Existem dois estados, i e j. O fluxo $i \rightarrow j$ executa a regra *rule_1* desde que a condição *cond_1* seja satisfeita.

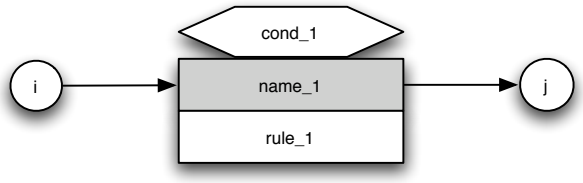


Figura 2.1: Exemplo de um diagrama FSM

Uma transição do FSM é definida como as regras de uma máquina, podendo ter o seguinte formato, considerando *ctl_state* como sendo o estado interno de um FSM:

```
if ctl_state = i then
  if cond_1 then
    rule_1,
    ctl_state = j_1
  endif,
    . . .
  if cond_n then
    rule_n,
    ctl_state = j_n
  endif
endif
```

Dado o estado i, cada condição $cond\_k$ satisfeita, para $i \leq k \leq n$, faz com que a máquina execute a respectiva regra $rule\_k$, e o controle interno (ctl_state) seja atualizado para j_k. Se nenhuma condição $cond\_k$ for satisfeita, nenhuma transição de estado ocorre. O diagrama correspondente a este código é mostrado na Figura 2.2. O diagrama mostra graficamente o funcionamento normal da máquina. Uma seta indica qual o estado de origem e destino em uma transição. Estados iniciais e finais podem não ser identificados diretamente na representação gráfica, portanto, devem ser explicitados na especificação.

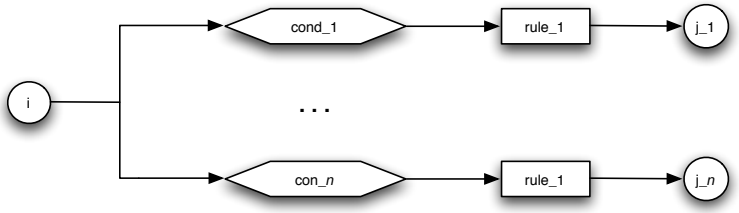


Figura 2.2: Exemplo de diagrama FSM tipo I.

Figura 2.3 exemplifica outro tipo de diagrama no qual, para o estado i, a *rule_1* é executada e a atualização do estado depende das condições representadas no hexágono.

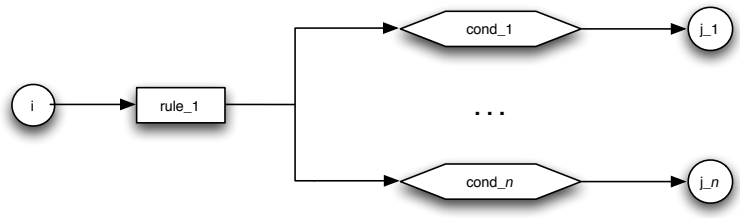


Figura 2.3: Exemplo de diagrama FSM tipo II.

O código correspondente ao diagrama da Figura 2.3 é mostrado a seguir.

```
if ctl_state = i then
  rule_1,
  if cond_1 then
    ctl_state = j_1
  endif,
    . . .
  if cond_n then
    ctl_state = j_n
  endif
endif
```

2.1.7 Conclusões

Uma das grandes vantagens do modelo ASM é a possibilidade de se executar uma especificação, o que pode facilitar a tarefa de encontrar erros. O modelo possui também recursos para modelar concorrência e não-determinismo, execução síncrona e assíncrona. Além disso, pode-se citar também a existência de uma teoria matemática subjacente, a teoria de Álgebras Evolutivas, que permite a prova de propriedades da especificação [BTIB05].

2.2 Método de Refinamento ASM

O Método de Refinamento ASM (*The ASM Refinement Method*) [Bör03a, BS03, E. 99] aproveita-se do poder de abstração e do rigor matemático oferecidos pelos conceitos de Máquina de Estado Abstrata (*Abstract State Machine - ASM*), definidos por Gurevich em [Y. 85] e aperfeiçoados em [Gur95]. De acordo com Börger [Bör03a], o método ASM de refinamento proporciona um *meta-framework* que integra conceitos mais específicos de refinamento. Como o modelo matemático é dividido em partes mais básicas, o nível

de abstração e modularização podem ser definidos livremente para melhor representar o sistema, desde que o modelo seja respeitado.

As principais características que fazem do refinamento ASM um método bastante utilizado são: (a) o fato da verificação (provar que é verdade) estar inclusa no processo, de forma que as transformações garantam o correto funcionamento e respeitem as propriedades impostas na especificação; (b) a possibilidade de, ainda na fase de projeto, validar o sistema, ou seja, provar que o sistema realmente realiza as tarefas que lhe foram designadas; (c) a especificação do problema é legível a projetistas e desenvolvedores, independente do código executável e plataforma de desenvolvimento.

No método ASM, o refinamento ainda é um processo manual sujeito a interpretações do desenvolvedor. As regras formam um mecanismo de auxílio às provas de propriedades e às transformações entre os níveis de abstração. O primeiro estágio do refinamento ASM visa a descrição do funcionamento do sistema de acordo com o problema a ser resolvido. São identificados os requisitos e as propriedades do sistema, e com essas informações constrói-se o modelo em alto nível, chamado de Modelo Básico (*Ground Model*) [Bör03a, Bör03b, BS03, E. 99], que deve ser legível a qualquer membro envolvido com o projeto.

O segundo estágio consiste na transformação do Modelo Básico em representações mais detalhadas pelas aplicações de regras bem definidas dentro do modelo. Normalmente, o objetivo é obter o código executável da aplicação, mas também pode ser desejável obter uma outra representação abstrata que sirva de Modelo Básico para outro processo de refinamento com tipos e regras diferentes.

As regras no método de refinamento ASM são diretivas para as corretas transformações entre os níveis e são aplicadas de acordo com os tipos de refinamento definidos no método. Cada tipo sugere um conjunto de regras de transformação podendo ser automática ou exigindo a interpretação do desenvolvedor. Cada aplicação de regra gera um conjunto de propriedades que devem ser garantidas ao longo de todo o processo de refinamento. Ao final, obtém-se a máquina refinada correspondente à máquina abstrata descrita no Modelo Básico.

Figura 2.4 ilustra o esquema de refinamento ASM em termos da equivalência de estados e execuções. Dada uma máquina abstrata M e um estado inicial S, realiza-se um conjunto de m passos (τ_1 ... τ_m) para se obter o estado seguinte S'. Na máquina refinada M*, deve existir um estado S*, correspondente ao S de M, e um conjunto diferente de n passos (σ_1 ... σ_n) que leva a um estado S*', que deve ser correspondente ao S'. O símbolo $\equiv$ no diagrama significa a equivalência entre os estados de M e M*, sendo $S \equiv S$* e $S' \equiv S$*'.

O diagrama da Figura 2.4 é conhecido como diagrama comutativo [Bör03a, Sch01a], onde os estados e operações das máquinas abstrata e refinada são relacionados. Utiliza-

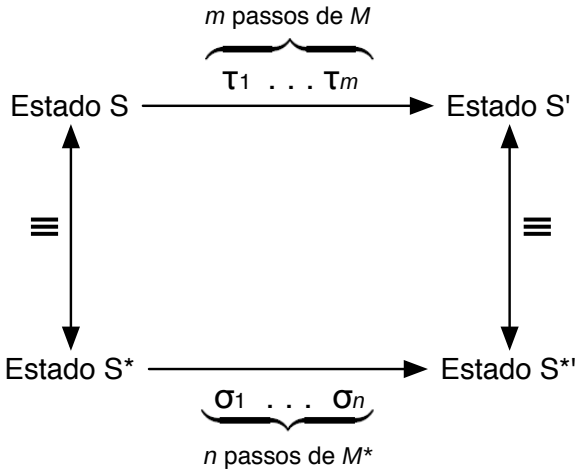


Figura 2.4: Esquema do Método de Refinamento ASM.

se a notação $m : n$ para dizer que a partir de S chega-se a S' com m passos de execução de M e que de S^* chega-se a $S^{*\prime}$ com n passos de execução de M^*, garantindo que $S \equiv S^*$ e $S' \equiv S^{*\prime}$. Então o diagrama da Figura 2.4 é um diagrama comutativo $m : n$, ou apenas diagrama $m : n$. A relação existente é entre uma transição de estado de M e outra de M^*, logo deve-se obter quantos diagramas comutativos forem necessários para representar a máquina abstrata para que o refinamento para M^* seja completo.

Para definir um método de refinamento baseado no método ASM, deve-se definir inicialmente a linguagem do Modelo Básico de fácil legibilidade e com características para melhor aproveitamento dos tipos de refinamento já identificados neste método. Conforme [Bör03a], também devem ser definidos os seguinte conceitos:

- estado abstrato S;
- estado refinado S^*;
- estados de interesse, que são os estados a serem observados no refinamento incluindo conceitos de estados iniciais e finais;
- correspondência ($\equiv$) entre estados: $S \equiv S^*$;
- computação abstrata τ_1 ... τ_m, onde τ_i representa um passo de execução máquina abstrata M;
- computação refinada correspondente σ_1 ... σ_n, onde σ_i representa um passo de execução de M^*, uma computação deve levar a máquina abstrata de um estado de interesse a outro;

- espaço de interesse e espaços correspondentes, espaços são conjuntos abstratos de dados. Por exemplo, pares de espaços que se deseja relacionar nos estados correspondentes;

- equivalência de dados nos espaços de interesse, normalmente é definida junto à equivalência de estados de interesse.

2.2.1 Modelo Básico

O desenvolvimento de sistemas exige o cumprimento das etapas de identificação das necessidades, projeto e especificação antes da efetiva implementação do *software*. A primeira etapa consiste em identificar e compreender as necessidades que o sistema deve satisfazer. Nos diversos campos, de trabalho existem pessoas com as mais variadas necessidades: melhorar o controle de produção; reduzir custo de vendas; aumentar a divulgação dos produtos e serviços; integrar as unidades da empresa espalhadas pelo mundo. Além de compreender as necessidades, um projetista deve analisar quem e quais partes da instituição serão afetados para definir o problema e propor uma solução adequada.

Normalmente, as descrições de necessidades são feitas em linguagem natural sob o ponto de vista de quem não projeta *softwares* (clientes), e devem ser compreendidas por quem provavelmente não conhece o problema (projetistas e desenvolvedores), que por sua vez podem descrever a solução que deve estar clara para os clientes. Fica claro o envolvimento de pessoas pertencentes a grupos distintos, e elas deverão trabalhar em conjunto para caracterizar o problema e propor uma solução, que posteriormente será convertida em um *software*. Deve-se ter um cuidado especial nesta etapa e escolher um modelo adequado, pois a divergência de interpretações pode levar a um sistema não satisfatório [Ber02, FPB87].

As descrições das necessidades e da solução realizadas na primeira etapa, embora essenciais, são comumente feitas de maneira informal e não adequadas a uma interpretação precisa. Portanto, a segunda etapa consiste em projetar o que será feito apresentando a solução sem ambigüidades em modelos formais, que constituem as características do sistema ("O que fazer") [LW99]. Por exemplo, para atender as necessidades de reduzir custos de venda e aumentar a divulgação de produtos e serviços, pode-se projetar a solução como um *web-site*. É possível dizer que o projeto representa a solução que irá satisfazer as necessidades reconhecidas e analisadas na primeira etapa. Além de descreverem com precisão o sistema, os modelos devem ser simples e suficientemente abstratos para serem compreendidos por todos os envolvidos no trabalho, clientes, projetistas e desenvolvedores, e permitir conferir se as necessidades foram

atendidas. Por esse motivo, essa etapa deve apresentar apenas detalhes da solução independentes do *software* que será gerado.

As características do sistema não tratam de detalhes necessários ao funcionamento do *software* ("Como fazer"), que são da terceira etapa. Tendo sido feita a descrição das necessidades na primeira etapa e características do sistema na segunda, deve-se dizer como o sistema se comportará, informando os requisitos do *software*, que constituem a terceira etapa. Os requisitos do *software* são mais detalhados que as características do sistema, embora ambos possuam um alto nível de abstração. Eles ilustram a diferença entre "O que fazer" e "Como fazer". Por exemplo, o *web-site* projetado na segunda etapa, pode ser especificado como uma implementação de um Comércio Eletrônico como é conhecido atualmente. Os requisitos de *software* tratam de assuntos direcionados ao processo de refinamento para obter o código executável automaticamente, devendo ser transparente para o cliente.

Denomina-se Modelo Básico [Bör03b] a especificação em alto nível de um sistema que abrange as atividades de identificação das necessidades, projeto da solução, definição de requisitos e propriedades do *software*, sendo o objeto para validação e ponto de partida da verificação e aplicação das regras de refinamento.

O Modelo Básico deve ser compreensível a dois domínios diferentes. O primeiro é chamado em ASM de Domínio da Aplicação [Bör03a, Bör03b, BS03] e contém os clientes, que possuem necessidades e normalmente não têm conhecimento em desenvolvimento de *software*. Este grupo também pode ser chamado de Domínio do Problema [LW99]. O segundo grupo, denominado Domínio dos Modelos, é constituído de projetistas e desenvolvedores, que são da área de computação e devem compreender as necessidades dos clientes e traduzir a solução em um *software*. Também pode ser chamado de Domínio da Solução.

Börger, em [Bör03b], descreve as seguintes propriedades para o Modelo Básico exercer o seu papel.

Preciso: apresentar os detalhes necessários à compreensão do sistema sem ambigüidades, minimizando o impacto no sistema existente.

Abstrato: apresentar a solução sem se prender a detalhes externos ao problema; ser compreensível no Domínio da Aplicação e no Domínio dos Modelos.

Flexível: suportar alterações e expansões de projeto sem adicionar informações desnecessárias ao projeto.

Completo: descrever as características do sistema e os requisitos do *software* atendendo a todas as necessidades identificadas.

Validação: permitir que o sistema seja validado ainda na fase de projeto, facilitando a identificação e correção de erros; garante que a solução esteja completa.

Semântica precisa: possuir rigor formal que permita a verificação do sistema e do refinamento.

O Modelo Básico deve conter a relação das necessidades identificadas apresentando suas definições e características, criando-se então uma introdução ao problema que o sistema deverá resolver. Em seguida, deve-se apresentar qual a solução adotada e como se chegou nela, preservando as iniciativas e decisões tomadas pela equipe de projeto. A partir desses dados, realiza-se a especificação do sistema destacando como as funcionalidades resolvem os problemas, em termos das necessidades. Por fim, no Modelo Básico, determina-se como as funcionalidades serão realizadas no sistema, considerando aspectos computacionais.

Com o objetivo de prover fácil legibilidade, normalmente utilizam-se descrições textuais para introduzir o problema e o sistema a ser desenvolvido. A escolha de uma linguagem de especificação apropriada é importante para obter alto nível durante o projeto, que garante a participação de clientes na validação. Assim, pode-se reduzir as possibilidades de um item importante ter sido esquecido. Uma técnica bastante eficiente é a utilização de linguagens gráficas, pois são bastante intuitivas e têm grande poder de expressão com construções bem simples. Porém, essas vantagens têm preço; a especificação fica incompleta e informal para ser submetida a um processo automático de refinamento. Então, as funcionalidades podem ser descritas em uma linguagem onde é possível expressar maiores detalhes sobre o que será realizado em cada parte. Essa formalidade é vantajosa para aplicação das regras de refinamento e verificação do sistema.

Assim, construído a partir do Domínio da Aplicação, o Modelo Básico exerce seu papel de registrar as necessidades e apresentar as características do sistema. Sendo fácil de compreender e modificar, contribui para a análise de requisitos de *software* ainda em alto nível, não sendo necessário aguardar o final do desenvolvimento para que outra equipe (de homologação por exemplo) ou usuários finais identifiquem problemas relacionados com a especificação. Com isso, o Modelo Básico é submetido a validação, verificação e são aplicadas as regras de refinamento.

O Modelo Básico tem um papel de documentação, pois contém todas as descrições formais e informais do problema e detalhes da solução independente do *software* e de plataforma. Como ele é construído em diversos níveis de abstrações, é possível que na presença de pessoas com perfis diferentes, como projetistas e desenvolvedores, ou mesmo na ausência da equipe original, seja possível compreender facilmente a proposta que o Modelo Básico especifica.

2.2.2 Tipos de Refinamento

A experiência em desenvolvimento em alto nível de sistemas utilizando ASM permite identificar certos padrões em especificações que dão origem aos tipos de refinamento [Bör03a]. Essa é uma maneira de classificar as regras de refinamento de acordo com as características do método contribuindo com a automatização do processo de refinamento. Embora existam alguns tipos já definidos, este é um tema em aberto para análise e pesquisas com a finalidade de identificar novos tipos de refinamento ASM.

Börger [Bör03a, BS03] apresenta os principais tipos de refinamento ASM e alguns deles são detalhados por Schellhorn [Sch01a]. A seguir, são apresentadas as características principais de cada um deles.

Extensão Conservativa

O tipo de refinamento Extensão Conservativa (*Conservative Extension*) adiciona um novo comportamento a uma máquina já existente, como tratamento de exceção ou características de robustez. Também é conhecido como Refinamento Incremental (*Incremental Refinement*)

O primeiro passo é definir a condição $bCond$ para o novo comportamento, indicando em que estado de uma máquina M, já definida, o novo comportamento deve atuar. Em seguida, define-se, em uma nova máquina B, o comportamento a ser adicionado em M, por exemplo, o tratamento de exceções ou medidas de segurança. Por fim, as duas máquinas devem ser integradas restringindo-se a execução de M por $bCond$, onde, então, B passa atuar e depois permite ou não que M retome sua execução.

Extensão Conservativa gera uma máquina mais completa e robusta, portanto mais complexa em termos de funcionalidade, onde as novas características adicionadas podem ter o mesmo nível de abstração que a máquina original. Este efeito é decorrente da separação entre o comportamento normal e situações não previstas ou de controle da execução, resultando em uma nova máquina.

Refinamento de Dado

O Refinamento de Dado (*Data Refinement*) consiste no mapeamento de estados e regras abstratas em representações concretas, preservando a relação operação-dado existente no modelo abstrato. A forma mais simples é realizar o mapeamento sem trocar assinaturas das operações e preservando as propriedades existentes (sem adicionar ou alterar alguma delas).

Freqüentemente, utiliza-se exemplificação de uma ASM, onde suas regras são preservadas e apenas as ações, normalmente as utilizadas externamente, são posteriormente

especificadas sem troca da assinatura. Isso possibilita o refinamento preservando características de modularização.

Refinamento de Sub-rotinas

O Refinamento de Sub-rotinas (*Procedural Refinement*), também conhecido como Refinamento de Sub-máquinas (*Submachine Refinement*), consiste em substituir uma máquina por outra mais complexa. O resultado é uma especificação mais detalhada onde é possível perceber novas características não presentes no nível de abstração anterior.

Um caso notável deste tipo é o Refinamento de Controle de Estado ASM (*Procedural Refinement of Control State ASMs*), aplicado sobre FSMs (Seção 2.1.6). Dada a semântica do diagrama, apresentada na Seção 2.1.6, pode-se definir as transições de uma FSM por meio da abstração

$$FSM(i, \textbf{if } cond \textbf{ then } rule, j)$$

Isso significa que, estando no estado i, se *cond* for satisfeita, a regra *rule* é executada, e o estado é alterado para j, caso contrário, nada é feito. Então tem-se uma abstração como esta para cada transição representada no diagrama FSM, ou seja, para $1 \leq k \leq n$ para todo estado i existe

$$FSM(i, \textbf{if } cond_k \textbf{ then } rule_k, j_k)$$

representando cada transição. Deve-se garantir a exclusão mútua de todas as condições $cond_k$ que possuem o mesmo estado i de origem.

O refinamento age sobre esta definição substituindo **if** $cond_k$ **then** $rule_k$ por uma nova sub-máquina M_k com mais detalhes e respeitando as restrições anteriores existentes. A máquina M_k pode conter inúmeras outras ações que são executadas atomicamente em paralelo. No Refinamento de Controle de Estado ASM, essas ações são padronizadas como síncronas e caso seja necessário representar assincronismo, o mesmo deve ser feito explicitamente, caracterizando o Refinamento Assíncrono de Ações (*Asynchronous Procedural Refinement of Atomic Actions*).

O Refinamento de Sub-rotinas troca uma definição abstrata por outra com mais operações, que antes eram supostas como existentes ou verdadeiras. A aplicação desta regra de refinamento em uma FSM representa a leitura de seu diagrama e o detalhamento de suas regras, podendo ter conjuntos de ações executando de forma síncrona ou assíncrona.

2.2.3 O Processo

Dada uma especificação de um sistema, devem ser feitos testes, especialmente simular situações de uso, de forma a confrontar as necessidades existentes com as funcionalidades descritas. Esta é uma prática realizada com o objetivo de validar o sistema perante o ambiente, mundo real, no qual está inserido. Muitas vezes, é possível modelar um sistema a partir dos casos de uso, tendo as funcionalidades definidas a partir de simulações no Modelo Básico. Assim, é possível definir, antes da implementação, uma aceitação mais precisa do projeto, já que o alto grau de abstração do Modelo Básico permite facilmente a participação dos clientes e usuários do sistema.

Uma especificação ASM possui uma característica operacional que provê noções implícitas de execução. Podem ser realizadas simulações manualmente ou utilizando algumas das ferramentas de execução ASM descritas em [E. 99, Capítulo 8]. Assim, o Modelo Básico ASM pode ser utilizado para identificar propriedades pertencentes ao Domínio da Aplicação que serão utilizadas para verificar o sistema de acordo com sua especificação.

Portanto, utiliza-se o Modelo Básico para especificação de requisitos e modelo de testes, onde condições do mundo real são consideradas. Além disso, propriedades são identificadas e é possível definir pontos de observação, elementos importantes para definição do modelo e noções de relação entre máquinas abstrata e refinada.

O Modelo Básico é inspecionado no Domínio da Aplicação para garantir que ele capture as pretensões iniciais de solução do problema. A relação de um problema e um sistema é descrita em termos das funcionalidades e propriedades, e a consistência desta relação deve ser garantida em todos os níveis de abstração. A verificação de um método de refinamento ASM é um processo que deve ser formalmente realizado a fim de mostrar que cada regra aplicada ao Modelo Básico preserva a relação problema-sistema. Dessa forma, garante-se que a implementação resultante está de acordo com o modelo abstrato.

A Figura 2.5 ilustra de forma simples como um refinamento deve proceder e como seus elementos, descritos anteriormente, se interagem. A definição do Modelo Básico é feita a partir dos modelos e dados da aplicação. A validação pode ser realizada antes (ainda no Modelo Básico) e depois da geração do código alvo, por meio de simulação e execução, respectivamente, comparando os resultados obtidos nos dois momentos. Cada passo do refinamento deve ser verificado provando-se as propriedades definidas até a obtenção do código executável. Com isso garante-se que cada nível de abstração atingido está correto.

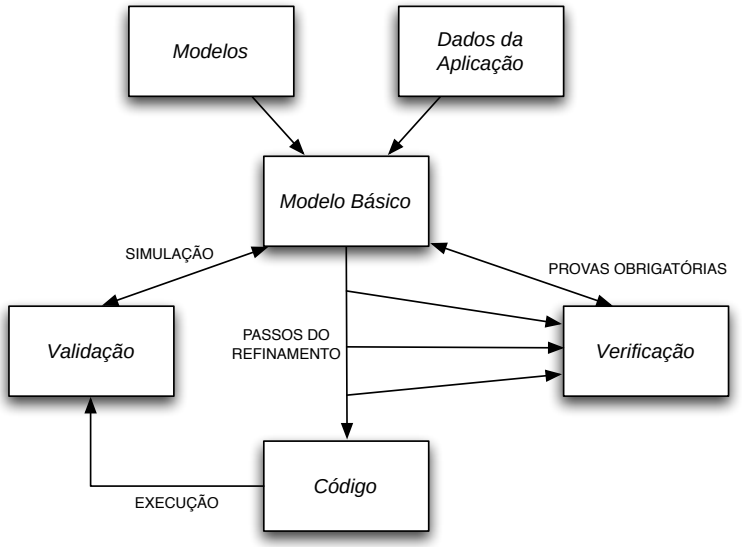


Figura 2.5: Processo de projeto de sistemas em alto nível. Figura adaptada de [BS03]

2.2.4 Aplicação do Método de Refinamento ASM

É possível encontrar inúmeras soluções de problemas reais que fazem uso do método de refinamento ASM com diferentes finalidades, como especificação de *software*, *hardware*, verificação de linguagens, descrição de algoritmos e protocolos. Por exemplo, o trabalho de Castillo e Päppinghaus [CP02] descreve o desenvolvimento em alto nível do núcleo de funcionamento de uma entidade de rede de telefonia e apresenta o refinamento para código C++. Já o relatório de Börger, Päppinghaus e Schmid [BPS00] mostra o uso de ASM na *Siemes AG* para a especificação de um processo do FALKO, modelo de organização de uma estrada de ferro na Alemanha. A partir do Modelo Básico descrito, utilizando *The ASM Workbench* [Del99], mostrou-se em [Sch01b] o esquema de compilação para o refinamento em C++, preservando a estrutura da especificação e sem gerar código ineficiente.

No âmbito de verificação de linguagens, os trabalhos [SSB01, BS98b, BS98a] apresentam a descrição completa em alto nível de Java e da JVM, onde são apresentadas análises (verificação e validação) matemática e experimental além de mostrar a compilação de programas Java para o código da JVM. Para a descrição em ASM, foi utilizada a ferramenta AsmGofer [Scha].

Ainda tratando-se de linguagens de programação, a formalização de Prolog é apresentada nos artigos [Bör90a, Bör90b, Bör92] em termos de ASM e seguindo o modelo de padronização de Prolog (WG17 of ISO/IEC JTCI SC22). A especificação proposta define a linguagem e realiza o refinamento passo a passo onde são descritas e provadas características ortogonais por meio de conjuntos modulares de regras.

O trabalho [Dis99] apresenta a modelagem do sistema operacional MINIX e da arquitetura x86 utilizando ASM, e posteriormente refinadas em uma implementação correspondente em Java e em C.

Considerando especificação de *hardware*, pode-se citar o caso de estudo do controle de iluminação (*The Light Control Case Study*), problema apresentado em [BG00] aplicando o controle em um andar de uma universidade. Considerando a arquitetura do prédio e implantação de *hardwares*, o trabalho [BRS00b] apresenta a discussão e a solução em ASM, onde são capturadas características informais e realizadas documentação do sistema. Então, o Modelo Básico é refinado utilizando AsmGofer, podendo validar e verificar a implantação do sistema de controle.

Um conhecido problema de especificação de *hardware* é o sistema de controle de produção (*Production Cell*), que em [Mea97], descreve-se a especificação em ASM e o refinamento para código C++.

Outro trabalho bastante conhecido, e muito utilizado como exemplos em várias descrições em alto nível, é a especificação de uma caldeira a vapor conforme a descrição de Abrial [Abr96b]. No trabalho [BBD+96], é apresentada a definição em ASM para a caldeira a vapor, ilustrando como a especificação em alto nível e verificação de sistemas complexos podem ser exploradas para um desenvolvimento bem documentado e validado, preservando a inspeção formal e aceitando alterações futuras. O Modelo Básico modularizado permite obter uma especificação com diversas visões do problema e é apresentada a transformação para C++. O problema da caldeira foi apresentado no seminário *Dagstuhl Seminar on Methods for Semantics and Specification* em Junho de 2005 com o desafio de obter a melhor especificação em alto nível. Os resumos dos principais resultados, incluindo [BBD+96], estão listados em [ABL95].

Estes são exemplos de aplicação do Método de Refinamento ASM, onde mostra-se a especificação em alto nível de diferentes tipos de sistema, incluindo *software*, *hardware*, algoritmos, linguagens e protocolos. É possível mostrar a facilidade de se expressar e refinar sistemas simples e complexos, possibilitando fácil documentação e validação enquanto define-se o Modelo Básico. O rigor formal exigido pelo modelo ASM, sem prejudicar a legibilidade, permite a verificação do sistema e prova do correto refinamento. Assim, o Método de Refinamento ASM é utilizado para especificação de sistemas reais contribuindo para soluções de problemas encontrados atualmente.

2.3 A Linguagem Machĭna

A linguagem Machĭna foi criada pelo grupo de pesquisa do Laboratório de Linguagens Programação da Universidade Federal de Minas Gerais (LLP - UFMG), e atualmente

se encontra na versão 2.0 [BTIB05]. Ela é baseada no conceito de Máquina de Estado Abstrata (ASM) [Gur95] e possui suporte à modularidade, criação de novos tipos e construções de alto nível. Nesta seção são abordados os elementos relevantes para o trabalho apresentado. Para a completa definição da linguagem Machĭna, deve-se consultar [BTIB05].

Um programa consiste na definição de um vocabulário, do estado inicial e da regra de transição que promove mudança de estados. Além disso, há também na linguagem construções para definição de invariantes de execução, controle de visibilidade intermódulos e comunicação entre agentes.

A linguagem Machĭna possui como características principais:

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

2.3.1 Módulos de Programa

Um módulo de programa especifica a regra que um agente executa, seu vocabulário, interpretação do vocabulário no estado inicial e o invariante da execução. Um módulo contém um mecanismo de controle de visibilidade, permitindo organizar o vocabulário de um agente em unidades encapsuladas, reduzindo sua complexidade.

Um módulo de programa tem a seguinte forma:

Módulo em Machĭna

```
module nome-do-módulo
  import elementos importados
  include elementos incluídos
  algebra:
    funções e tipos
  abstractions:
    ações
  initial state:
    inicializações de funções dinâmicas
  transition:
    regras
  invariant:
    invariantes de execução
end nome-do-módulo
```

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

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

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

A seção *transition* define a regra de transição de estado do agente, na qual é executada repetidamente de uma só vez, ou em uma seqüência de passos, quando o agente é disparado. Esta seção representa o corpo do programa dos agentes associados ao módulo.

A seção *invariant* define a condição envolvendo os elementos de um módulo que deve ser invariante durante sua execução. Entre duas iterações da regra de transição do módulo, o invariante é verificado pelo sistema de execução Machĭna. Caso seja violado, uma mensagem de erro é emitida e a execução, interrompida.

2.3.2 Regra de transição

A regra de transição de um agente é definida na seção *transition* de um módulo. Pode ser uma regra de transição de um único passo ou então uma seqüência de passos de

execução. No primeiro caso, a regra é executada repetidamente quando o agente é disparado e finaliza quando se atinge uma condição de terminação.

No caso de múltiplos passos, a regra executa um passo de cada vez, indicados pela cláusula **step**, iniciando com **step := 1**. A atualização direta de **step** não é permitida em Machĭna, para isso utiliza-se a variável implícita **next**, inicialmente 1. Antes de cada passo de transição, **next** é automaticamente incrementada, e após toda transição, faz-se **step := next**. Se não houver um passo especificado, ou seja, se for um passo omisso, nada é feito na transição a não ser o incremento de **next** e a atribuição **step := next**.

O valor de **step** pode ser explicitamente alterado em um passo atribuindo-se um valor à **next**, o qual então prevalece sobre o incremento automático. A transição é dita concluída após a execução do último passo, neste caso, o valor corrente de **step** supera a definição do último passo, então as funções implícitas **step** e **next** são reiniciadas com o valor 1 e a regra se repete, realizando o mesmo processo.

Considere a seguinte regra de transição com seqüência de passos e as regras R1 e R2:

Regra com seqüência de passos

```
1  . . .
2  transition:
3    step 1: R1;
4    step 2: R2;
5    step 4: if g then next := 2; end
```

Neste exemplo, a regra se inicia com a execução de R1 e depois de R2. No terceiro passo, o omisso, nada é feito, exceto incrementar a função **next** e a atribuir à **step**. No quarto passo, a guarda g é avaliada, se for verdadeira, **next** é atualizada para executar o passo 2. Quando g for falsa, nada é feito e, no passo seguinte, **step** vale 5, excedendo os passos definidos. Então, **step** e **next** são reiniciadas e o processo recomeça do passo 1.

2.3.3 Mecanismos de Visibilidade

Os elementos (funções e tipos) que compõem a álgebra de um agente são naturalmente privados, ou seja, visível apenas no módulo onde são declarados. A classificação *public* significa que a função de um módulo A será incorporado à álgebra de um módulo B que inclui A pela cláusula *include*. Neste caso, há a duplicação das funções incluídas, fazendo os módulos A (incluído) e B (que inclui) terem diferentes interpretações dessas funções.

Funções declaradas como compartilhadas, classificação *shared*, possuem a mesma interpretação entre os agentes de um mesmo módulo. Caso sejam públicas, a duplicação não ocorre, havendo assim o compartilhamento dessas funções entre os agentes dos módulos que as declarou e dos módulos que as incluem.

A cláusula *include* incorpora em um módulo os elementos públicos de outro, devendo-se qualificá-los quando utilizados. Pode-se explicitar os elementos incluídos sendo possível utilizá-los diretamente sem a necessidade de qualificação.

O exemplo a seguir mostra a álgebra de um módulo A. Na linha 3, define-se a função *funcA* privada. A função pública *funcB* na linha 4 pode ser incorporada à álgebra de outros módulos. Na linha 5, declara-se a função *sA*, a qual é compartilhada entre os agentes do módulo A. Em seguida, na linha 6, declara-se a função *psA*, que é compartilhada entre os agentes de A e agentes de módulos que incluem A.

Módulo A

```
1  module A
2    algebra:
3      funcA              : Y;
4      public funcB       : T;
5      shared sA          : T;
6      public shared psA  : T;
7    . . .
8  end A
```

A seguir, o módulo B inclui o módulo A, sendo assim, B tem a visibilidade dos elementos públicos de A (a função *funcB*) e compartilha a função *psA*. Os demais elementos de A não são visíveis em B. Neste caso de inclusão, os elementos de A devem ser utilizados em B com qualificação, como mostrado na linha 9.

Módulo B inclui A

```
1  module B
2    include A;
3
4    algebra:
5      funcD : T;
6
7    abstractions:
8      action actB is
9        funcD := A.funcB;
10     end
11   . . .
12 end B
```

O módulo C a seguir inclui a função $funcB$ pública de A explicitamente, linha 2, portanto, não é preciso qualificá-la na linha 9.

Módulo C inclui A

```
1  module C
2    include A(funcB);
3
4    algebra:
5      funcE : T;
6
7    abstractions:
8      action actC is
9        funcE := funcB;
10     end
11   . . .
12 end C
```

O mecanismo de visibilidade permite a encapsulação de um módulo de Machĭna, uma vez que um agente não pode acessar livremente dados de outros. Isso reduz a complexidade dos módulos e contribui para reusabilidade.

2.3.4 Interfaces de Agentes

Um agente pode disponibilizar serviços estabelecendo uma comunicação entre agentes, contribuindo para a modularidade. Isso é realizado por meio da definição da interface, que deve ter o mesmo nome do módulo e pode conter tipos da álgebra e abstrações de regras, chamadas de ação. O exemplo a seguir mostra uma possível definição para a interface de um módulo A.

Interface do módulo A

```
1  interface A
2    type T;
3    action actA;
4  end A
```

Um módulo B importa a interface de A por meio da cláusula *import* da seguinte forma:

Módulo B importa A

```
1  module B
2    import A;
```

```
3     . . .
4   end B
```

Assim, um agente do módulo A disponibiliza seus serviços (abstrações e tipos) e um agente de B os utiliza pela importação da interface de A, linha 2 do módulo B.

2.3.5 Definições de Machĭna

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

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

A diretiva *agent of M*, dentro do módulo de definição *machina*, cria um agente, a partir do módulo M, e dispara sua regra de transição. O vocabulário do agente é formado por uma cópia própria de todas as declarações contidas no módulo M e nos módulos a partir dele incluídos via a diretiva *include*. A interação entre agentes é possível mediante troca de mensagens mediada pelo *superagente* ou por acesso à funções compartilhadas *shared*.

Após a criação de todos os agentes no módulo *machina*, a execução de cada um é iniciada. As regras de transição dos agentes são executadas repetidamente em paralelo e simultaneamente. Cada ciclo de execução da regra de transição de estado de um agente é atômico, sem qualquer tipo de interrupção ou bloqueio. Somente entre uma execução da regra da transição e a seguinte é que agentes podem ser bloqueados, por exemplo, por falta de recursos solicitados de outros módulos, solicitação explícita ou alguma inconsistência.

O exemplo a seguir ilustra o módulo *machina* que cria o *superagente* X, em seguida, cria um agente do módulo A e seis agentes de B, todos independentes entre si.

Módulo de Definição *machina*

```
machina X
  agent of A;
  agent of B(5);
  agent of B;
end
```

2.3.6 Ações

Ações são abstrações de regras para definição de tipos abstratos de dados com o seguinte esqueleto:

Ação

```
action A (X) is
  require: P;
  ensure Q;

  L;

  // body
end
```

A execução em *body* contém um conjunto de regras de transição sobre as funções do vocabulário global e local. Uma ação A pode receber um conjunto X de parâmetros tipados, que podem ser qualificados com **in**, passagem por valor, **out** ou **inout**, ambos são passagem por nome (*call by name*). Opcionalmente, uma ação pode conter um conjunto L de declarações locais.

As atualizações realizadas em A somente serão percebidas no estado seguinte. Valores das funções locais são preservados para a próxima chamada de S, caso não exista inicialização de funções locais em L. A cláusula *require* exige que A seja chamada satisfazendo a pré-condição P. A cláusula *ensure* mostra o que a ação A deve garantir após sua execução, ou seja, ao final da transição, a pós-condição Q deve ser satisfeita. Qualquer falha em P ou Q termina a execução do agente em uma situação de erro. O escopo de P e Q abrange o escopo do módulo mais os parâmetros formais contidos em X. Declarações locais não fazem sentido nessas expressões, uma vez que *enquire* e *ensure* são cláusulas que relacionam a ação com o contexto que ela está imerso, onde os elementos de L não são visíveis.

A pré-condição P de uma ação A é avaliada no momento da chamada, e os valores das funções utilizadas são obtidas no estado corrente. No caso da pós-condição Q, o superagente armazena a expressão de Q em uma lista de pós-condições. Ao final da transição, o superagente verifica todas as condições armazenadas e os valores das funções e relações são obtidos no estado seguinte.

A ação *inc* mostrada a seguir incrementa um o argumento x passado. A pré-condição é omitida, portanto *inc* pode ser chamada sem nenhuma restrição adicional. A pós-condição diz que essa ação garante, ao final de sua execução, que o valor de x obtido será uma unidade maior que o valor de x anterior, para isso, utiliza-se o

qualificador *old*.

Ação

```
action inc (out x : Int) is
  ensure x = old x + 1;

  x := x + 1;
end
```

Suponha uma transição $i \rightarrow j$, onde i e j são estados, e o valor de x no estado i, x_i, seja 10. Após a execução de *inc* obtém-se o estado j e o valor de x_j igual a 11. O comando *old* na pós-condição se refere ao valor de x_i enquanto x_j já foi calculado. Portanto, a cláusula

$$ensure\ x\ =\ old\ x\ +\ 1;$$

é equivalente a fazer na transição $i \rightarrow j$

$$ensure\ x_j\ =\ old\ x_i\ +\ 1;$$

substituindo os valores

$$ensure\ 11\ =\ 10\ +\ 1;$$

Ações definidas com assinatura $A(X)$ sugestiva e associada às pré e pós-condições permitem uma visão em alto nível de sua tarefa. Com isso, é possível indicar suas condições de chamada e garantias sem conhecer profundamente sua execução.

2.3.7 Comunicação entre Agentes

Agentes podem se comunicar de duas forma. A primeira, é via chamadas a abstrações de regras (*action*) que são declaradas na interface dos módulos principais. Durante a execução de sua regra de transição, um agente a pode solicitar a execução de uma abstração de regra disponibilizada via interface por um outro agente b. Este processo é entendido como o envio de uma mensagem do agente a ao agente b, solicitando a execução do serviço definido pela abstração chamada, cujos parâmetros servem para transmitir informações ao agente b ou dele recebê-las.

Todo agente possui o contador *PendingAnswers* e as filas *RequestsReceived* e *Answers-Received*, controlados pelo *superagente* para coordenar as trocas de mensagens entre agentes. Embora essas estruturas ocupem o espaço de dados do agente, não há meios de

manuseá-las diretamente em Machĭna. A fila *RequestsReceived* armazena os pedidos de execução de abstrações, e seus respectivos parâmetros, feitos por outros agentes. A fila *AnswersReceived* tem o formato de uma lista de atualizações e seu conteúdo é relativo ao valor de retorno de parâmetros **out** e **inout** das requisições enviadas a outros agentes. O contador *PendingAnswers* contabiliza o número de requisições com parâmetros do tipo **out** e **inout** feitas a outros agentes. Um agente somente pode re-executar sua regra de transição quando *PendingAnswers* estiver zerado.

Quando um agente a solicita a execução de uma abstração de um outro agente b, o *superagente* coloca o pedido, e seus parâmetros, na fila *RequestsReceived* de b. Se houver parâmetros do tipo **out** ou **inout**, o *PendingAnswers* de a é incrementado e este agente continua sua transição corrente, mas será impedido de iniciar a próxima.

Quando o agente b atende a solicitação de a, o *superagente* retira a requisição da fila *RequestsReceived* de b e, no caso de parâmetros **out** e **inout**, os valores destes parâmetros são calculados e colocados na fila *AnswersReceived* de a e seu contador *PendingAnswers* é decrementado. Então o agente a atualiza seus dados de acordo com a fila *AnswersReceived* e começa a próxima iteração. A fila *RequestsReceived* de b pode conter várias requisições de diversos agentes, assim, cada pedido é atendido um a um, de forma atômica.

As chamadas a abstrações de regras de outros agentes são assíncronas em relação à execução da transição corrente, isto é, o envio de mensagens não causa bloqueio do agente durante a transição corrente, porém a próxima transição somente será retomada após o recebimento das informações solicitadas pelas mensagens enviadas durante a última iteração da regra de transição, isto é, quando o *PendingAnswers* for igual a zero. O bloqueio de execução da transição seguinte evita erros de acesso a valores ainda não disponíveis, preservando a propriedade de que as atualizações terão validade na iteração seguinte. A fim de evitar o bloqueio da próxima transição, um serviço pode ser solicitado de forma assíncrona marcando a chamada a abstração correspondente com a palavra **asynchronously**. O exemplo Mestre-Escravo a seguir, Seção 2.3.8, ilustra este comportamento.

O *superagente* controla automaticamente as solicitações de serviços realizadas pelos agentes, e este é o único meio de controle existente. O *superagente* é responsável por realizar o controle de concorrência de dados de forma transparente ao programa escrito em Machĭna, ficando de acordo com o modelo de Máquina de Estados Abstrata, [Gur95]. Assim, problemas como condição de corrida e *deadlock* são mais facilmente resolvidos nesta linguagem.

Outra forma de comunicação entre agentes é via funções compartilhadas (*shared*), as quais suas interpretações são as mesmas para todos os agentes. Uma função declarada apenas como *shared* em um módulo A é compartilhada entre os agentes de A. Se for

public shared, o compartilhamento acontece entre os agentes de A e agentes dos demais módulos que incluem A.

Como os agentes executam em paralelo, fica evidente a possibilidade da existência de condição de corrida, e os meio necessários para este controle devem ser realizado pelo desenvolvedor.

2.3.8 Exemplo: Mestre-Escravo

O algoritmo Mestre-Escravo é bastante utilizado na área de redes. Consiste na execução paralela de um determinado número de processos, denominados escravos (*slaves*). Um processo mestre (*master*) coordena o momento do início da execução dos escravos e aguarda a finalização de todos eles. Este é um exemplo de execução assíncrona e será mostrado como é implementado em Machĭna.

O algoritmo funciona da seguinte forma: antes do processo mestre dar a ordem de execução aos processos escravos, ele pergunta se todos estão disponíveis e aguarda a resposta. Caso verdadeiro, o mestre ordena que todos executem sua tarefa paralelamente e de forma assíncrona, caso contrário é dada a ordem para cancelar a execução. Pode ser possível que o mestre execute, também em paralelo, sua própria tarefa durante o tempo de execução dos escravos.

Para a simplicidade do exemplo, são consideradas comunicações mestre-escravo sem falhas e o mestre fica infinitamente tentando dar ordem de execução aos demais processos. A solução aqui apresentada foi baseada na descrição de Börger em [BS03, Capítulo 6].

A seguir, define-se o módulo para escravo, denominado *Slave* com os seguintes elementos:

Order Tipo público simbolizando a ordem de execução (*job*) ou cancelamento (*cancel*);

Answer Tipo público simbolizando a possível resposta do escravo, informando que está disponível (*accept*) ou indisponível (*refuse*);

order Função que representa a ordem recebida.

O mestre pergunta aos escravos por meio da abstração *GetAnswer* (linha 15), onde a resposta é dada no parâmetro *ans*, passado como **out**. O agente escravo, inicialmente, aguarda pela ordem do mestre (linha 33). Ao receber a ordem, o agente executa sua tarefa, definida na ação *Task*, se a ordem for *job*, caso contrário nada é feito. Posteriormente, finaliza-se a execução (linhas 35 e 36).

Ao informar ao mestre que a execução foi finalizada (linha 36), o agente escravo utiliza a abstração *ReportResult*, em uma chamada assícrona, que é importada da interface do módulo que define o mestre (linha 2).

Módulo *Slave*

```
1  module Slave
2    import Master;
3
4    algebra:
5      public type Order  = enum {job, cancel}    default undef;
6      public type Answer = enum {accept, refuse} default undef;
7
8      order : Order;
9
10   abstractions:
11     action Clear is
12       order := undef;
13     end
14
15     action GetAnswer (out ans : Slave -> Answer) is
16       choose a : Answer
17       do ans(self) := a;
18       end
19     end
20
21     action SetOrder (in ord : Order) is
22       order := ord;
23     end
24
25     action Task is
26       // doSomething
27     end
28
29   initial state:
30     Clear;
31
32   transition:
33     step 1: if order = undef then next := 1;
34             else if order = job then Task; end
35                  Clear;
36                  Master.ReportResult(self) asynchronously;
37         end
38
39 end Slave
```

O módulo *Slave* define como canal de comunicação, que será utilizado pelo agente mestre, os tipos *Order* e *Answer*, as abstrações *GetAnswer*, utilizada para obter uma resposta sobre a disponibilidade do escravo, e *SetOrder*, utilizada para dar a ordem de execução ou cancelamento.

Inteface *Slave*

```
1  interface Slave
2    public type Order;
3    public type Answer;
4    action GetAnswer   (out ans : Slave -> Answer);
5    action SetOrder (in ord : Order);
6  end Slave
```

O módulo mestre, denominado *Master*, é apresentado a seguir com os seguintes elementos:

slaves Grupo de processos escravos;

n Número de processos escravos definidos externamente;

order Ordem a ser dada aos escravos;

answersArrived Mapeia um escravo na sua respectiva resposta;

resultsArrived Mapeia um escravo na sua condição de execução;

allReported Indica se todos os escravos terminaram sua execução.

Inicialmente são criados os n agentes escravos que ficam aguardando pela pergunta. Ao iniciar sua regra de transição, o processo mestre pergunta a todos os escravos se é possível executar via ação *Enquire* (linha 52). Esse procedimento é realizado requisitando a ação *GetAnswer* do módulo *Slave* (linha 24), e como a chamada é síncrona, o agente mestre fica aguardando que todos os escravos respondam para seguir ao passo seguinte. Este passo tem o papel de sincronizar o mestre com todos os escravos.

Em seguida, é certo que todas as respostas chegaram ao mestre, então o mestre as avalia verificando se existe alguma resposta recusada e define qual ordem a ser dada na ação *EvalAnswersArrived* (linha 27). No passo seguinte, a ordem é enviada aos escravos, utilizando a ação *SetOrder* definida em *Slave*. Como a chamada é feita de forma assíncrona, não ocorre bloqueio para a próxima iteração do mestre, que executa sua própria tarefa (*MasterTask* na linha 55) até que todos os escravos tenham terminado (linha 56). Por fim, o mestre reinicia o algoritmo (linha 57).

Módulo *Master*

```
1  module Master
2    import Slave;
3
4    algebra:
5      slaves                : agent of Slave;
6      external n            : Int;
7      order                 : Order;
8      answersArrived        : Slave -> Answer;
9      resultsArrived        : Slave -> Bool;
10     derived allReported   : Bool := all s : slaves satisfying
11                                     resultsArrived(s) := true;
12   abstractions:
13     action Clear is
14       order  := undef;
15       alReported := false;
16       forall s : slaves
17       do answersArrived(s) := undef;
18          resultsArrived(s) := false;
19       end
20     end
21
22     action Enquire is
23       forall s : slaves
24       do s.GetAnswer(answersArrived); end
25     end
26
27     action EvalAnswersArrived is
28       if exists s : slaves satisfying answersArrived(s) = refuse
29       then order  := cancel;
30       else order  := job;
31     end
32
33     action SendOrder is
34       forall s : slaves
35       do s.SetOrder(order) asynchronously; end
36     end
37
38     action MasterTask is
39       // master does something while wait for slaves
40     end
41
42     action ReportResult (in s : Slave) is
43       resultsArrived(s) := true;
44     end
```

```
45
46      initial state:
47        create agent slaves(n);
48
49      transition:
50        step 1: forall s : slaves do dispatch s; end
51                Clear;
52        step 2: Enquire;
53        step 3: EvalAnswersArrived;
54        step 4: SendOrder;
55        step 5: MasterTask;
56        step 6: if not allReported then next := 5;
57        step 7: Clear; next := 2;
58
59    end Master
```

A notificação enviada ao mestre informando que um escravo terminou sua tarefa é realizada pela ação *ReportResult*. *ReportResult* é declarada na interface de *Master*, conforme a seguir, para ser corretamente utilizada em *Slave*. Ela também não causa nenhum bloqueio de iterações, pois sua chamada é feita com a marca **asynchronously**, portanto preserva as execuções em paralelo e de forma assíncrona dos agentes escravos e o mestre.

Inteface *Master*

```
1   interface Master
2     action  ReportResult (in s : Slave);
3   end Master
```

Este exemplo ilustra uma execução multiagentes implementada em Machĭna. A característica principal do problema é a execução paralela e assíncrona das tarefas realizadas pelo mestre e demais escravos, assim como é possível criar pontos de sincronização dos processos. O passo 3 (linha 53) da transição do processo mestre representa um ponto de sincronia do sistema, conforme a exigência da especificação de que todos os escravos devem estar disponíveis para execução de suas tarefas no mesmo instante. Nos demais pontos das regras de transição, tanto de *Slave* quanto de *Master*, a execução é assíncrona em relação aos demais agentes do sistema, como é a proposta do algoritmo. Assim, foi possível mostrar como Machĭna aborda execuções assíncronas com determinados pontos de sincronização.

2.3.9 Exemplo: Produtor-Consumidor

O problema Produtor-Consumidor consiste em um *buffer*, um conjunto de produtores responsável por incluir elementos no *buffer* e um conjunto de consumidores responsável por retirar elementos incluídos pelo produtor. Deve-se controlar o acesso ao *buffer* para evitar leitura incorreta de dado e sobrescrita de um dado ainda não consumido. A seguir é apresentada uma solução em Machĭna.

O módulo *Buffer* define os seguintes elementos:

buffer Lista de valores que o *buffer* armazena;

maxSize Tamanho máximo do *buffer*, privado e não pode ser alterado (*static*);

empty Função derivada que indica se o *buffer* está vazio;

full Função derivada que indica se o *buffer* está completamente cheio.

Módulo *Buffer*

```
1  module Buffer
2    algebra:
3      buffer  : list of Int;
4      static maxSize : Int := 10;
5      derived empty : Bool := length(buffer) = 0;
6      derived full  : Bool := length(buffer) = maxSize;
7
8    abstractions:
9
10     public action Write(in x : Int, out ok : Bool) is
11       if full then   ok := false;
12       else   ok := true;
13              buffer := buffer :: x;
14       end
15     end
16
17     public action Read(out x : Int, out ok : Bool) is
18       if empty then
19         ok := false;
20       else
21         ok     := true;
22         x      := head(buffer);
23         buffer := tail(buffer);
24       end
25     end
26 end Buffer
```

As ações *Write* e *Read* do módulo *Buffer* provêm uma maneira de um Produtor incluir o elemento produzido e um Consumidor o ler. Por isso são declaradas na interface do módulo. É possível notar a ausência de regra de transição, caracterizando *Buffer* como um módulo provedor de serviços.

Interface *Buffer*

```
1  interface Buffer
2    public action Write(in x : Int, out ok : Bool);
3    public action Read(out x : Int, out ok : Bool);
4  end Buffer
```

O módulo *Consumer* define o comportamento de um consumidor. Primeiramente, é definida uma função *unit*, linha 6, que armazena o valor recuperado. A função *ok* é utilizada para indicar se a leitura foi realizada quando solicitada e *buffer* é uma referência ao agente de *Buffer*. *Consumer* importa o módulo *Buffer*, conforme mostrado na linha 2, tendo acesso aos seus serviços disponibilizados. A regra de transição define que um agente consumidor tenta ler os elementos de *buffer* infinitamente. Quando solicita-se a leitura de *buffer*, este retorna em *ok* se foi ou não possível, assim, um consumidor pode processar o dado lido ou realizar nova tentativa.

Módulo *Consumer*

```
1  module Consumer
2    import Buffer;
3
4    algebra:
5      ok     : Bool := false;
6      unit   : Int  := 1;
7      buffer : Buffer := Buffer.theAgent(1);
8
9    transition:
10     if ok then
11       // processa unit
12       ok   := false;
13     else
14       buffer.Read(unit, ok);
15     end
16 end Consumer
```

O módulo *Producer* define o comportamento de um produtor e possui características muito semelhantes ao módulo *Consumer*, com a diferença que tenta-se escrever continuamente o novo dado produzido, conforme descrito na regra de transição.

Módulo *Producer*

```
1  module Producer
2    import Buffer (buffer,full);
3
4    algebra:
5      ok     : Bool := false;
6      unit   : Int  := 1;
7      buffer : Buffer := Buffer.theAgent(1);
8
9    transition:
10     if ok then
11       unit := unit + 1;
12       ok   := false;
13     else
14       buffer.Write(unit, ok);
15     end
16 end Producer
```

A máquina *Host* define um número arbitrário de agentes consumidores e agentes produtores. Todos serão executados em paralelo e de forma independente. O agente de *Buffer* também é criado, e pela definição nos módulos *Consumer* e *Producer*, ambos possuem a mesma referência ao agente de *Buffer*. As funções *numberOfConsumers* e *numberOfProducers* são funções externas, permitindo criar quantidades variadas de consumidores e produtores.

Máquina *Host*

```
1  machina Host
2    agent of Buffer;
3    agent of Consumer(numberOfConsumers);
4    agent of Producer(numberOfProducers);
5  end
```

Dessa forma, é possível controlar o acesso ao *Buffer* sem criar condição de corrida. Cada pedido para escrever ou ler é armazenado na lista de requisições do módulo *Buffer* e processadas uma a uma atomicamente. A garantia de sucesso na operação, de acordo com a disponibilidade (cheio ou vazio) do *buffer*, é dado na função *ok* passada como parâmetro **out** nas ações *Write* e *Red*. A comunicação entre agentes realizada via troca de mensagens síncronas garante que o sistema será executado de forma correta e segura, no que diz respeito ao compartilhamento de informações de *Buffer*.

2.3.10 Exemplo: Semáforo

Outra possível solução para o problema Produtor-Consumidor pode ser dada via compartilhamento de funções. Assim, deve-se controlar o acesso para evitar condição de corrida. Então, utiliza-se a implementação de um semáforo, conforme descrito a seguir.

O módulo *Buffer* compartilha as funções *buffer*, *empty* e *full* entre produtores e consumidores.

Módulo *Buffer*

```
1  module Buffer
2
3    algebra:
4      static maxSize : Int := 10;
5      public shared buffer        : list of Int;
6      public shared derived empty : Bool := length(buffer) = 0;
7      public shared derived full  : Bool := length(buffer) = maxSize;
8
9  end Buffer
```

Para o controle de acesso às funções compartilhadas, utiliza-se a definição de semáforo a seguir. A função *resources* permite a generalização para um número qualquer de recursos compartilhados, no caso de Produtor-Consumidor, existe apenas um, o *buffer*.

Quando um agente necessita de um recurso, ele faz a solicitação via ação P e a resposta é dada em *permission*. Ao final da seção crítica, o agente libera o recurso via ação V.

Semaphore

```
1  module Semaphore
2    algebra:
3      resources : Int := 1;
4
5    abstractions:
6      public action P (out permission : Bool) is
7        if resources > 0 then
8          permission := true;
9          resources  := resources - 1;
10       else
11         permission := false;
12       end
13     end
14
15
```

```
16      public action V is
17        resources := resources + 1;
18      end
19  end Semaphore
```

O módulo *Consumer* importa os serviços de *Semaphore* e inclui explicitamente os elementos *buffer* e *empty* do módulo *Buffer*. Além de *unit*, declara-se *sem*, obtendo o agente de *Semaphore* declarado na máquina de execução. Por fim, *permitted* é a função que armazena a resposta dada quando é feita a solicitação do recurso.

A regra de transição é dividida em dois passos, no primeiro, faz-se a solicitação do recurso via chamada do serviço P do agente *sem*. No segundo passo, a resposta, dada em *premitted*, é analisada. Caso a permissão tenha sido concedida, o consumidor tem acesso exclusivo aos elementos compartilhados, podendo então obter um elemento de *buffer*, caso o mesmo não esteja vazio, e atualizá-lo. Ao final, libra-se o recurso e atribui-se falso à *permitted*.

Módulo *Consumer*

```
1   module Consumer
2     import Semaphore;
3     include Buffer(buffer, empty);
4
5     algebra:
6       unit      : Int := 0;
7
8       sem       : Semaphore := Semaphore.theAgent(1);
9       permitted : Bool := false;
10
11    transition:
12      step 1:
13        sem.P(permitted);
14
15      step 2:
16        if permitted and not empty then
17          unit   := head(buffer);
18          buffer := tail(buffer);
19
20          sem.V;
21          permitted := false;
22        end
23  end Consumer
```

O módulo *Producer* possui as mesmas características de controle das funções com-

partilhadas, como pode ser visto em sua regra de transição.

Módulo *Producer*

```
1  module Producer
2    import Semaphore;
3    include Buffer(buffer, full);
4
5    algebra:
6      unit      : Int := 0;
7
8      sem       : Semaphore := Semaphore.theAgent(1);
9      permitted : Bool := false;
10
11   transition:
12     step 1:
13       sem.P(permitted);
14
15     step 2:
16       if permitted and not full then
17         buffer := buffer :: unit;
18         unit   := unit + 1;
19
20         sem.V;
21         permitted := false;
22       end
23 end Producer
```

A máquina *Host* define o agente de semáforo, responsável por controlar o acesso ao recurso compartilhado, e os agentes de consumidores e produtores.

Máquina *Host*

```
1  machine Host
2    create Semaphore;
3    create Consumer(numberOfConsumers);
4    create Producer(numberOfProducers);
5  machine
```

Este exemplo mostra uma segunda solução para o problema Produtor-Consumidor realizando a comunicação entre os agentes por meio de funções compartilhadas. Os problemas relacionados à concorrência são solucionados via implementação de semáforo, e a definição da seção crítica deve ser feito pelo desenvolvedor

2.4 Conclusões

A abstração presente em ASM torna o modelo legível, e a possibilidade de executar uma especificação facilita a depuração do sistema. O modelo também dispõe de recursos para descrição de concorrência, não-determinismo e execução síncrona e assíncrona. O formalismo existente em ASM é fundamental no auxílio a provas de propriedades da especificação.

O conjunto de máquinas nas quais os estados são finitos, as FSMs, são importantes para definição de estruturas de controle que possibilitam o manuseio de estruturas de dados do sistema. Assim, pode-se tornar a modelagem legível e intuitiva, simplificando a tarefa de validação do sistema.

Tirando proveito do rigor matemático e da legibilidade do modelo ASM, o Método de Refinamento ASM tem as características de verificação inclusa no processo de refinamento, validação ainda durante o projeto e especificação legível e independente de plataformas. O Modelo Básico é o recurso de especificação de sistemas, que deve ser compreensível aos domínios da Aplicação e do Modelo. Além disso, sua construção contém características de certos padrões de refinamento que facilitam a aplicação das regras de transformação, que são classificadas e podem fornecer propriedades a serem garantidas.

Machĭna é uma linguagem baseada no modelo ASM que possui suporte à modularidade, é fortemente tipada e possui construções em alto nível. Além disso, possibilita extensão de tipos e definição de invariantes de sistema. A execução multiagente é independente e pode ser síncrona ou assíncrona. A herança de características do modelo ASM permite a definição de um método de refinamento para esta linguagem, baseada nas definições do Método de Refinamento ASM.

Capítulo 3

Método de Refinamento Machĭna

O Método de Refinamento Machĭna (MRM) proposto nesta dissertação é baseado no Método de Refinamento ASM de Egon Börger [Bör03a]. MRM tem como objetivos propor uma linguagem em alto nível para descrição do Modelo Básico, que permita validar a especificação ainda durante o projeto, descrever as regras de refinamento para obter o código Machĭna correspondente e propor um método de verificação automática. Devido à sua complexidade, a verificação no MRM é descrita separadamente no Capítulo 4. Neste capítulo, é apresentada a linguagem de MRM para especificação do Modelo Básico. Em seguida, descreve-se como realizar a validação do Modelo Básico e são apresentadas as regras de refinamento para obter o código Machĭna.

3.1 Linguagem de Modelagem Machĭna

Linguagem de Modelagem Machĭna (LMM) é uma linguagem de especificação para um Modelo Básico fundamentada em diagramas de fluxo de controle e de definição modular por meio de tipos abstratos de dados. Além das características comuns de Modelo Básico, LMM suporta especificação de processos assíncronos e é baseada em desenhos gráficos de representação em alto nível. Todos os conceitos inerentes da especificação da linguagem Machĭna, definida em [BTIB05] e resumida na Seção 2.3, são aplicadas à LMM, inclusive palavras chaves e sintaxe de descrição. Linguagem de Modelagem Machĭna é definida com os seguintes elementos:

Descrição Textual: Contém a descrição em linguagem natural do problema e das soluções adotadas. Cada parte da solução deve conter o texto que a descreve e, em seguida, a formalização constituída pelo Diagrama de Transição de Estados, que modela o comportamento do sistema, e pelo Glossário. Expressa as necessidades que o *software* deve satisfazer e pertence ao Domínio da Aplicação.

Tipo Abstrato de Dados (TAD): Descreve as características de novos tipos de

dados de acordo com a interpretação das necessidades identificadas. Descreve os elementos do mundo real em termos de requisitos do sistema. Modela aspectos estáticos e pertence ao Domínio dos Modelos.

Diagrama de Transição de Estados (DTE): Modela o comportamento de parte do sistema ou o sistema inteiro. Mostra o fluxo de controle das atividades e as transições (e suas ações) realizadas entre os estados. Consiste na representação dos requisitos e formaliza a solução adotada de maneira abstrata, sendo o nível mais alto de abstração no modelo formal. Modela aspectos dinâmicos e pertence ao Domínio dos Modelos.

Glossário: É onde definem-se condições e ações utilizadas no Diagrama de Transição de Estados. Ajuda a controlar o grau de abstração nos diagramas e também é o primeiro passo de refinamento, assim como a primeira intervenção humana no processo de refinamento. Modela aspectos estáticos e pertence ao Domínio dos Modelos. O Glossário é dividido em duas partes. A primeira é chamada de **Álgebra**, onde são definidas as estruturas de dados e funções que constituem um TAD, e a segunda é chamada de **Abstrações**, onde definem-se as ações que foram descritas no Diagrama de Transição de Estados.

Mecanismo de Visibilidade: Diagrama que modela a composição de módulos e interação entre agentes (instâncias de módulos). Um módulo (TAD) pode incorporar elementos declarados como públicos em outro módulo para formar seu vocabulário. Agentes podem se comunicar utilizando serviços disponibilizados na interface de seus módulos. O primeiro mecanismo está vinculado à definição da Interface na descrição de TAD, e o segundo diz respeito à Álgebra no Glossário.

A especificação de todos esses elementos define a máquina de execução que representa todo o sistema. Assim, LMM permite a formulação de um Modelo Básico com alto nível de abstração, portanto fácil legibilidade, alto poder de expressão e bastante flexível. A Figura 3.1 mostra como a LMM aborda cada etapa de especificação de sistemas e como se relaciona com os dois domínios apresentados. Em seguida, é dada a notação para a LMM.

3.1.1 Descrição Textual

O problema a ser solucionado é descrito informalmente em textos que expressam suas características e necessidades. Então, define-se e documenta-se a proposta de solução, decisões e estratégias utilizadas. Essas informações contribuem fortemente para o reconhecimento e caracterização dos Tipos Abstratos de Dados, que representam os módulos que irão compor o sistema.

Na descrição do Método de Refinamento Machĭna, é necessário definir uma formatação padrão para o texto resultante. Portanto, utiliza-se a notação LaTeX [Knu84] para

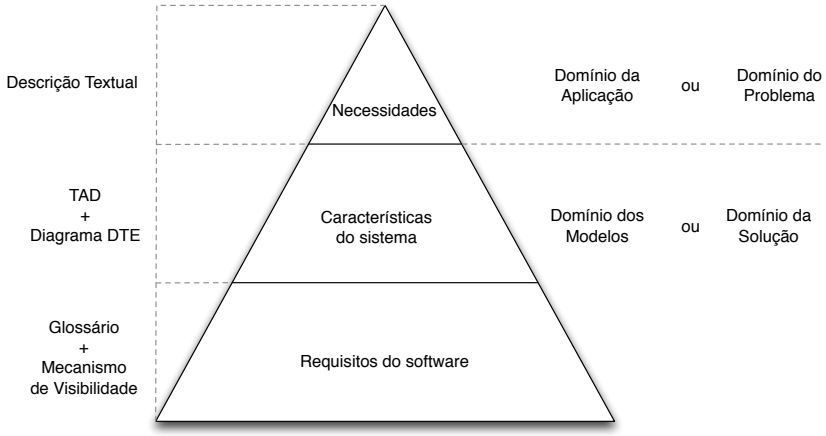


Figura 3.1: Etapas de especificação de sistemas.

edição do texto devido suas características de independência de ambiente, plataforma livre e facilidade de uso [Knu83].

3.1.2 Tipo Abstrato de Dados

Um Tipo Abstrato de Dados (TAD) é a representação em LMM de um elemento do mundo real constituído de um conjunto de características e propriedades, identificadas e organizadas na especificação informal. Cada TAD define um determinado módulo com abertura para futuras especificações de suas funcionalidades. Aplica-se o conceito de programação por contrato para definir as operações, abstraídas por ações em LMM, que o módulo deve prover [Mey97]. Demais ações e características, importantes internamente ao TAD, são definidas posteriormente no Glossário, Seção 3.1.4.

Os seguintes elementos são necessários para se definir um TAD em LMM:

- nome: identifica o TAD dentro do sistema e deve ser sugestivo à sua funcionalidade;
- álgebra: mínima representação do vocabulário que caracteriza o TAD. São definidos nomes de funções e tipos abstratos sem preocupação com tipagem, pois o objetivo é obter uma representação abstrata e ser independente de linguagens;
- invariante: conjunto de invariantes de execução de um agente;
- interface: abstrações (*actions*), que representam as operações, e tipos abstratos que são visíveis externamente ao módulo para o cumprimento de seu contrato. Tem importante papel na comunicação entre agentes, pois define os serviços e tipos que podem ser utilizados por outros agentes.

A notação de Tipo Abstrato de Dados, Figura 3.2, é dada por um retângulo representando o módulo, o nome, que aparece no canto superior esquerdo, e possui duas regiões. A primeira, chamada *algebra*, descreve minimamente os principais elementos, funções, relações e tipos abstratos, que caracterizam o TAD. O segundo, chamado de *invariant*, define os invariantes em função da álgebra e deve seguir a mesma sintaxe do bloco *invariant* de Machĭna.

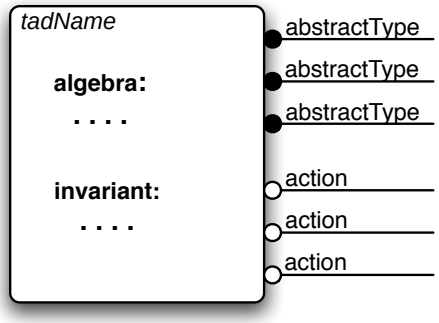


Figura 3.2: Notação para Tipo Abstrato de Dados

À direita do retângulo existem dois tipos de segmentos de reta que podem ser utilizados para definição da interface do Tipo Abstrato de Dados. Os segmentos acoplados por um círculo cheio definem os tipos abstratos que serão visíveis externamente. Uma vez que um tipo foi utilizado no segmento, ele não precisa estar na seção álgebra do TAD. Os segmentos acoplados por um círculo vazio definem os nomes dos serviços providos pelo TAD.

Em LMM, os Tipos Abstratos de Dados expressam as características básicas do conjunto de elementos responsáveis pelas tarefas do sistema. Posteriormente, durante o processo de refinamento, cada Tipo Abstrato de Dados será mais detalhado em sua estrutura, fluxo de controle e relações entre agentes de outros módulos.

3.1.3 Diagrama de Transição de Estados

Diagrama de Transição de Estados (DTE) é uma derivação de FSM (Seção 2.1.6) para se adequar às necessidades e especificações da Linguagem de Modelagem Machĭna. DTEs são utilizados para modelar o comportamento e descrever as características de módulos em alto nível com o objetivo de ser facilmente interpretado por qualquer um que esteja envolvido no projeto. Um diagrama trata de aspectos dinâmicos da especificação e é o elemento pertencente ao Domínio dos Modelos que está mais próximo do Domínio da Aplicação. Por isso, é construído em um alto grau de abstração sem deixar de lado aspectos formais do modelo.

Cada Diagrama de Transição de Estados está associado a um Tipo Abstrato de Dados e representa o comportamento de seus agentes. É constituído de um conjunto de estados relacionados pelas transições, que contêm condições e ações. O estado corrente do diagrama também pode ser chamado de estado interno, e é possível estar em apenas um único estado em um determinado instante. As condições e ações devem estar de acordo com o modelo ASM e seguir a sintaxe da linguagem alvo adotada no método de refinamento.

A notação DTE é derivada da notação de diagramas de estados de UML [RJB04, BRJ05] e FSM com algumas alterações com o objetivo de atender melhor o modelo ASM e permitir um método de refinamento direcionado à linguagem Machĭna. A seguir, é definida a notação do diagrama apresentando cada parte que o compõe.

3.1.3.1 Estados

Um estado de um DTE representa a situação que satisfaz alguma condição, realiza uma atividade ou aguarda que uma condição seja satisfeita. É composto de:

- Nome: identifica unicamente um estado;
- Atividade: ação executada a cada iteração enquanto o agente permanecer em um determinado estado. O nome da atividade é dado explicitamente na forma *atv: atvStateName*, podendo vir dentro ou fora do círculo que representa o estado, como mostra a Figura 3.3. As ações são definidas na Seção 3.1.3.4.

(a) Nome da atividade dentro.

(b) Nome da atividade fora

Figura 3.3: Representação de um estado, *state* indica o nome do estado.

3.1.3.2 Estado Interno

Todo módulo contém um único estado interno, chamado de *ctl_state*, que designa o estado corrente dentre os possíveis que se pode assumir. O controle sobre essa função é totalmente realizado pelo Método de Refinamento Machĭna, não sendo permitido seu uso na especificação em alto nível LMM. A função predefinida *current_state* permite obter o valor de *ctl_state* para consulta, mas não aceita atualizações explícitas.

3.1.3.3 Transições

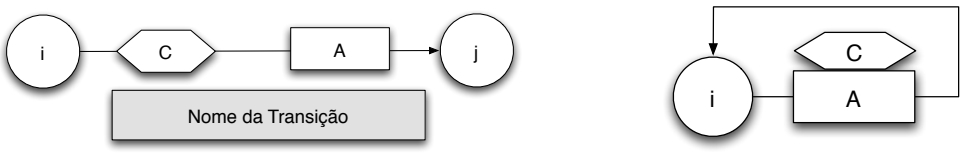
Uma transição é uma relação entre dois estados. Representada por uma seta, sendo composta de:

- condições: fatos que devem ser satisfeitos (C);
- ações de transição: conjunto de ações a serem executadas (A);
- estado de origem (i);
- estado de destino (j);
- nome: opcional, representado por um retângulo preenchido.

A Figura 3.4(a) exemplifica a representação gráfica de uma transição entre dois estados do DTE.

O fluxo de uma transição consiste na ordenação temporal de seus elementos seguindo a orientação *origem* → *destino*. Como as ações são executadas em paralelo, essa ordenação determina o controle do fluxo, ou seja, quais ações serão executadas e qual será o estado de destino. Se o primeiro elemento for uma condição C, a transição é acionada quando C é satisfeita no estado de origem do fluxo. Assim, C pode ser considerada o disparador da transição. Caso o primeiro elemento seja uma ação, a transição será fatalmente disparada. O agente apenas altera seu estado se o fluxo atingir o estado de destino. A alteração de estado e o efeito da execução das ações somente são percebidos na iteração seguinte, estando de acordo com o modelo ASM.

Uma transição pode conter o mesmo estado para origem e destino, como o exemplo da Figura 3.4(b). Muito embora, no contexto do diagrama, o estado não seja alterado nessa transição, o estado interno do módulo (*ctl_ state*) é atualizado com o mesmo valor de antes. Assim, as transições representadas nas Figuras 3.4(a) e 3.4(b) concorrem pela atualização de *ctl_ state*. Este assunto será melhor abordado posteriormente na Seção 3.1.3.5.



(a) Transição e elementos (b) Origem igual destino

Figura 3.4: Representação de transição de estado.

3.1.3.4 Ações de transição

As ações de transição representam um conjunto de atualizações do vocabulário realizadas durante uma transição de estado. Todas as ações que devem ser executadas em uma transição têm o seu efeito percebido apenas na próxima iteração.

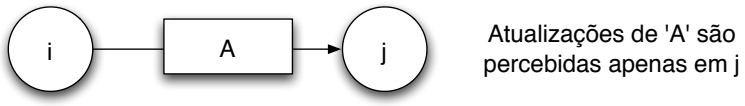


Figura 3.5: Representação de uma ação.

Uma ação é representada por um retângulo e um nome, como na Figura 3.5. Este nome deve ter um correspondente em Abstração no Glossário para uma definição mais detalhada. Caso não haja, o método de refinamento deve realizar o tratamento adequado ou reportar o erro. Por exemplo, a ação A mostrada na Figura 3.5 poderia atualizar o vocabulário do módulo como mostra o código Machĭna a seguir.

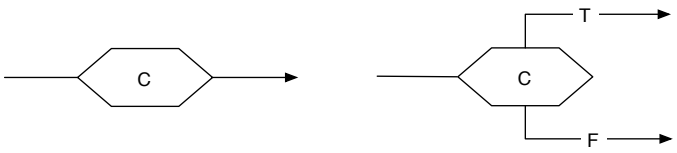
Exemplo de uma Ação

```
1  action A is
2    f(t) := x;
3    h(t) := y;
4  end
```

3.1.3.5 Condições

Uma condição é uma expressão booleana presente na transição. Sua avaliação determina a passagem do fluxo. Os fluxos de entrada e saída são determinados pelos estados de origem e destino da transição. Uma condição deve conter um único fluxo de entrada e deve existir pelo menos um fluxo de saída, e.g. Figura 3.6(a), permitindo ou não sua continuidade, e no máximo dois, e.g. Figura 3.6(b), ramificando o fluxo em cada caso de avaliação (verdadeira ou falsa). No caso de um único fluxo de saída a avaliação verdadeira da condição permite a continuidade, enquanto que a avaliação falsa impede que a transição seja efetuada, conseqüentemente o estado não é alterado. No caso de dois fluxos de saída, o diagrama terá ramificações e deve-se explicitar qual fluxo se refere à avaliação verdadeira (T) e à falsa (F).

Uma seqüência de condições em uma transição corresponde ao aninhamento destas e a alteração de estado somente é realizada se o fluxo atingir um estado de destino. Isso possibilita a execução de ações sem a efetiva alteração do estado. Figura 3.7(a) ilustra essa situação, e para melhor compreensão do que ocorre, a Figura 3.7(b) mostra



(a) Um fluxo de saída (b) Dois fluxos de saída

Figura 3.6: Representação de uma condição.

o pseudo código correspondente. Repare que a atualização de *ctl_state* é feita apenas na condição mais interna.

(a) Seqüência de condições

Pseudo código correspondente

```
if C1 then
   A1
   if C2 then
      A2
      ctl_state ← j
   end
end
```

(b) Pseudo código correspondente

Figura 3.7: Aninhamento de condições.

Um estado pode ser origem de várias transições, sendo assim, pode ser possível executar mais de um fluxo em uma transição de estado. Porém, se os destinos forem diferentes, a primeira condição de cada fluxo alternativo deve ser mutuamente exclusiva, caso contrário, a transição de estado é inválida, pois poderá haver uma inconsistência na atualização do estado interno do módulo. O modelo ASM não especifica o que deve ser feito diante de uma inconsistência na atualização [Gur95]. Considera-se a transição como inválida, pois uma propriedade do Diagrama de Transição de Estados é que somente pode-se estar em um único estado em um dado instante. O controle das condições mutuamente exclusivas deve ser feito pelos projetistas e desenvolvedores. Na Figura 3.8, as condições $C1$ e $C2$ podem gerar inconsistências e devem ser mutuamente exclusivas.

Quando os fluxos de mesma origem possuem o mesmo estado de destino a inconsistência não ocorre. O comportamento seriam fluxos opcionais para execução de ações com uma guarda, e neste caso, mais de um fluxo opcional pode ser executado. Na Figura 3.8, as condições $C2$ e $C3$ não geram inconsistências. Vale lembrar que as considerações feitas sobre condições são válidas tanto para transições com origem di-

ferente de destino como para transições com origem igual ao destino, já que neste tipo de transição o estado interno do módulo é atualizado. Isso quer dizer que, sejam i, j_1 e j_2 estados, a primeira condição de uma transição $i \rightarrow j_1$ e a primeira condição de uma transição $i \rightarrow j_2$ ou $i \rightarrow i$ devem ser mutuamente exclusivas.

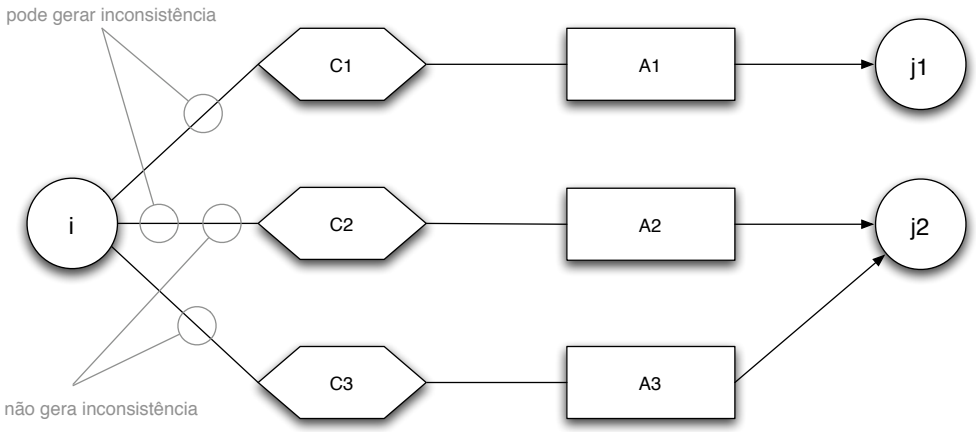


Figura 3.8: Situações de concorrência de fluxos.

Mostrou-se aqui, a inconsistência no Diagrama de Transição de Estados no momento da transição de estado. É possível também ocorrer inconsistências nas ações que atualizam a Álgebra do módulo. Essas podem não ser visíveis no diagrama e sim no Glossário, devendo ser controladas por desenvolvedores e projetistas para estar de acordo com o modelo ASM.

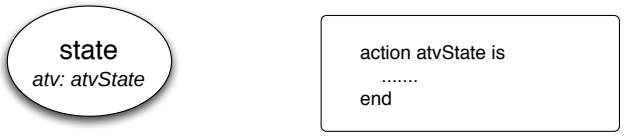
3.1.3.6 Atividades

Uma atividade é uma ação associada a um estado que é executada a cada iteração, enquanto o agente permanecer no mesmo. A atividade é opcional e é semelhante a um fluxo que possui origem igual ao destino, porém não atualiza o estado interno do agente *ctl_state*. Sua execução é independente dos fluxos de saída do estado, ou seja, a cada iteração é executada antes de avaliar qual fluxo seguir. Outra diferença é que como a atividade não possui estado de destino, ela não cria situação de inconsistência. quanto a atualização de *ctl_state*.

Uma atividade é associada ao estado escrevendo-se seu nome após a palavra reservada *atv* seguido de ':', como mostra a Figura 3.9(a). Então, deve-se definir a atividade no Glossário, na parte de Abstrações, de acordo com o nome dado, Figura 3.9(b).

3.1.3.7 Estado inicial

Todo Diagrama de Transição de Estados deve conter um estado especial no qual a execução tem início. Este deve ser um estado de preparação, onde as ações de inicialização



(a) Estado. (b) Definição no Glossário.

Figura 3.9: Representação e definição de atividade.

do vocabulário do agente devem ser realizadas.

O estado inicial é representado com um pequeno círculo cheio com a letra *I* no meio, destacando-se dos demais estados, Figura 3.10. Ele é obrigatório e único para cada diagrama, não pode ser destino de nenhum fluxo, não contém atividade e o Controle Global, descrito na Seção 3.1.3.9, não tem efeito sobre ele. Deve existir um único fluxo de saída sem a presença de condições, apenas ações são permitidas.

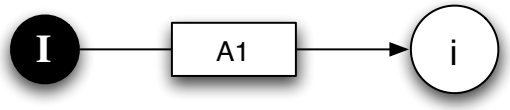


Figura 3.10: Representação do estado inicial.

3.1.3.8 Estado final

Um estado final de um Diagrama de Transição de Estados é identificado no modelo como um estado que não é origem de nenhum fluxo. Essa definição faz sentido uma vez que se não há nada a ser feito em um estado, as iterações seguintes não afetam as funções e relações do agente. Porém, um estado final pode conter uma atividade como artifício de ação pré-finalização, e essa atividade é executada uma única vez antes da execução do agente ser finalizada. Ao contrário do estado inicial, o estado final não é obrigatório nem único, e não existe a necessidade de uma palavra reservada. A Figura 3.11 exemplifica um estado final, percebe-se que não há fluxos saindo do estado.

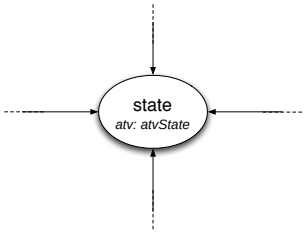


Figura 3.11: Representação do estado final.

3.1.3.9 Controle Global

O Controle Global contém condições e ações que deverão ser aplicadas a todas transições de estados, exceto a que tem o estado inicial como origem. São restrições de prioridade maior, ou seja, são realizadas antes de quaisquer atividade, ação ou restrição do fluxo normal do DTE. Para sua representação, utilizam-se diagramas DTE reduzidos. São ditos reduzidos pois descrevem um pequeno trecho do sistema seguindo as definições de DTE, podendo não conter explicitamente o estado de destino. Utiliza-se *current_ state* como estado de origem indicando que todos os estados, origem de algum fluxo, serão submetidos ao Controle Global.

Dado o estado corrente *current_ state*, podem ser definidos dois tipos de execução global. O primeiro adiciona um DTE reduzido a todos os estados e o fluxo continua conforme descrito no DTE principal. Pode-se notar que sua representação na Figura 3.12 não mostra o estado de destino. Este tipo de controle global devolve o fluxo para o DTE principal.

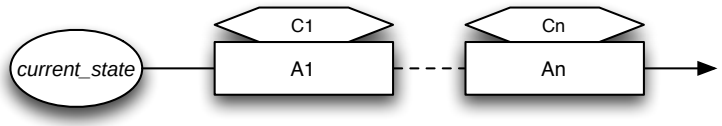


Figura 3.12: Controle Global: devolve o fluxo.

O segundo tipo de execução global desvia o fluxo da transição de estados para um estado definido, como mostrado na Figura 3.13. Neste caso, as regras e restrições definidas no fluxo normal do DTE podem não ser avaliadas.

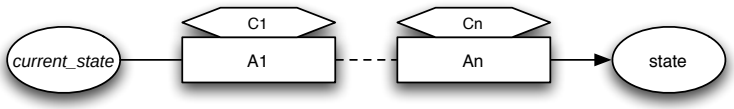


Figura 3.13: Controle Global: desvia o fluxo.

Na área global, apenas é permitido um único DTE reduzido de forma a evitar inconsistências na transição de estados. Porém, é possível fazer a composição dos DTEs reduzidos necessários, criando assim um conjunto de condições e ações que será executado antes de qualquer fluxo do DTE principal. Todas as considerações feitas para o DTE são válidas também para o diagrama reduzido. A Figura 3.14 mostra um esquema de composição de DTEs reduzidos utilizando os dois tipos exemplificados nas Figuras 3.12 e 3.13. Dado o estado corrente *current_ state*, antes de seguir o fluxo normal, a condição *C1* é verificada. Se for verdadeira, desvia o fluxo para o estado

stateE, caso contrário, a condição *C2* é avaliada e se for verdadeira executa a ação *A1*, se for falsa, executa a ação *A2*, e em ambos os casos, o fluxo retorna ao fluxo normal do DTE principal.

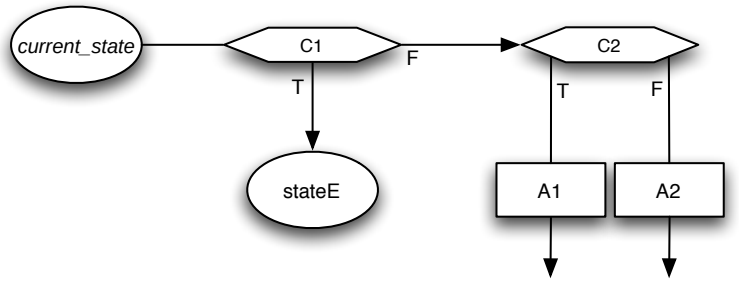


Figura 3.14: Controle Global: composição.

O Controle Global de um DTE é executado na saída de cada estado, exceto o inicial, de um agente de um módulo. Os círculos cinzas mostrados na Figura 3.15 indicam os pontos onde o diagrama reduzido do Controle Global é executado.

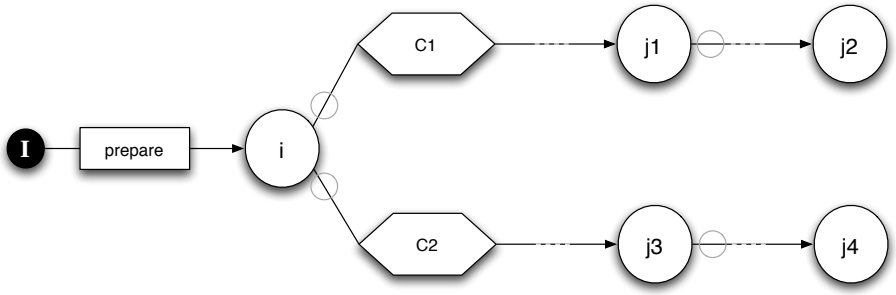


Figura 3.15: Atuação do Controle Global.

3.1.4 Glossário

Os DTEs modelam o comportamento e descrevem as características de agente de um módulo em alto nível, com o objetivo de ser facilmente compreendidos por qualquer pessoa. Essas informações são essenciais para o entendimento do problema e solução adotada, porém não fornecem informações suficientes para a implementação direta do sistema. O Glossário trata de aspectos estáticos do modelo, tendo o papel de definir funções, abstrações, estruturas de dados e expressões apresentadas nos diagramas assim como em outras definições do próprio Glossário.

Com essas definições mais detalhadas, é possível controlar o nível de abstração adotada nos diagramas, deixando detalhes necessários à implementação para o Glossário.

Assim, tem-se o primeiro passo de refinamento mapeando os nomes de ações, atividades, funções e expressões presentes em um DTE em trechos mais apropriados para a tradução na linguagem alvo do refinamento.

O método de refinamento deve ser o mais automatizado possível, porém, são necessários alguns pontos de intervenção humana para que se consiga obter uma especificação do sistema em alto nível de abstração. O Glossário, então, é o primeiro ponto de intervenção humana no processo de refinamento, permitindo a aplicação posterior das regras de refinamento para obter o código executável de forma automatizada.

As partes que formam o Glossário têm relação com a linguagem alvo adotada no método de refinamento. Seguem, portanto, a mesma sintaxe de acordo com cada bloco que representa. Em LMM, as definições do Glossário seguem a sintaxe da linguagem Machĭna.

3.1.4.1 Álgebra

DTE modela o comportamento de um agente de um módulo e suas atividades abstraindo sua construção interna, o que é desejável até este momento. Porém, existem elementos, que não são relevantes no diagrama abstrato, mas são fundamentais para atender os requisitos do sistema. Esses elementos são definidos na Álgebra como funções, tipos e estruturas de dado seguindo a sintaxe de Machĭna no bloco *algebra*.

As condições descritas nos fluxos do DTE são definidas na Álgebra. Além das condições, a Álgebra deve também atender às ações definidas em Abstrações, ou seja, funções presentes em condições e ações devem estar especificadas nesta parte do Glossário. Os elementos presentes na descrição do TAD (Seção 3.1.2) também devem ser definidos na Álgebra.

Os itens da Álgebra são definidos em uma caixa rotulada de *Glossário: Álgebra*.

Glossário : Álgebra

```
1  type Name1;
2  type Name2
3
4  public functionF1 : Int;
5  functionF2 : Bool;
6  external functionF3 : Int;
7  ......
```

As definições da Álgebra tratam de aspectos estáticos do módulo, uma vez que determinam sua estrutura interna e controlam a visibilidade dos elementos, interferindo

nos Mecanismos de Visibilidade (Seção 3.1.5), quando diz respeito ao mecanismo de inclusão.

3.1.4.2 Abstrações

Cada ação que é definida no fluxo de transição ou como atividade de um estado no DTE deve ser definida no Glossário na parte de Abstrações. As ações podem fazer uso de outras ações, que também devem ser definidas. O bloco correspondente em Machĭna é denominado *abstractions* e deve-se seguir a mesma sintaxe, inclusive considerando pré e pós condições.

De acordo com o modelo ASM, uma inconsistência ocorre quando existem duas atualizações diferentes para um mesmo elemento da Álgebra na mesma transição de estado. Portanto, desenvolvedores e projetistas devem ter o cuidado de não gerar inconsistências durante a especificação do Glossário. Caso exista uma inconsistência em uma transição, esta pode ser abortada, e a máquina tem sua execução interrompida.

As abstrações são definidas em uma caixa rotulada de *Glossário: Abstrações*.

Glossário : Abstrações

```
1  action actionName (actionParams) is
2    require: P;
3    ensure:  Q;
4
5    doSomething;
6    ......
7  end
```

As ações descrevem as especificações de *software* necessárias para a implementação em código alvo. A regra de transição de um módulo de Machĭna é composta, além das funções estruturais do DTE, de um conjunto de ações definidas em Abstrações. Inconsistências devem ser tratadas para evitar interrupções no funcionamento da máquina.

3.1.5 Mecanismos de Visibilidade

Um módulo pode incorporar, por um mecanismo chamado *Include*, elementos declarados como públicos em outro módulo para formar seu vocabulário. Agentes podem se comunicar, por meio do mecanismo *Import*, utilizando serviços disponibilizados na interface de seus módulos. Além disso, é necessário informar qual módulo dará origem à máquina que representa o sistema. O diagrama de Mecanismos de Visibilidade modela essas relações conforme a seguir.

- *Include*: Mecanismo no qual um módulo A incorpora em seu vocabulário elementos (Álgebra) definidos em outro, por exemplo B. Quando A inclui B, todas as funções públicas de B ficam visíveis em A e devem ser qualificadas quando utilizadas. A inclusão mútua entre dois módulos não é permitida no modelo e deve ser contornada pelo projetista com a fusão destes módulos;

- *Import*: Mecanismo no qual um agente de um módulo B estabelece um canal de comunicação com outro agente de um módulo A por meio da interface de B. Sendo assim, pode-se dizer que agentes de A utilizam os serviços disponibilizados por B, ou A importa a interface de B;

- *Máquina*: Define unicamente a máquina que representa o sistema.

Para a modelagem dos mecanismos citados, utiliza-se o Diagrama TAD com a seguinte notação:

- **TAD**: São representados por caixas e devem ser identificadas pelo mesmo nome do TAD. Por exemplo, a Figura 3.16 mostra os possíveis relacionamentos entre TAD A com o TAD B.

- **Relação**: Relaciona dois TADs e é representada por uma seta de sentido único da inclusão, ou seja, $B \rightarrow A$ significa que A inclui B, se a seta tiver na origem um círculo preenchido, Figura 3.16(a). Para importação pode-se utilizar a seta de sentido único $B \rightarrow A$, com um círculo vazio na origem, indicando que A importa a interface de B, Figura 3.16(b), ou de ambos os sentidos, indicando que tanto A importa a interface de B quanto B a de A, Figura 3.16(c). A utilização de um mecanismo não impede a existência do outro, ou seja, é possível que A inclua os elementos públicos e importe a interface de B ao mesmo tempo. Neste caso, utiliza-se a seta com ambos os sentidos, por exemplo $A \leftrightarrow B$.

- **Máquina**: Uma caixa cinza indica qual TAD representa a máquina do sistema. A definição da máquina é única e obrigatória. Por exemplo, se o TAD A for representado por uma caixa cinza, é criada a máquina contendo um único agente do tipo A que tem sua regra de transição disparada. Quando existe apenas um módulo no sistema, o diagrama TAD não existe, portanto cria-se a máquina de acordo com este único módulo.

O Diagrama TAD visa modelar o relacionamento entre Tipos Abstratos de Dados por meio de Mecanismos de Visibilidade e definir a máquina do sistema. Dessa forma, pode-se estabelecer a composição de módulos por incorporação de elementos ou comunicação entre agentes por utilização de serviços disponibilizados. A contribuição

(a) Módulo A inclui B

(b) Agentes de A importam serviços de B

(c) A importa B e B importa A

Figura 3.16: Mecanismos de visibilidade.

é um alto grau de modularidade e reúso de comportamentos modelados em diversos sistemas.

3.1.6 Palavras Reservadas

Além das palavras reservadas de Machĭna, Linguagem de Modelagem Machĭna possui as seguintes:

- *ctl_state*: função que representa o estado interno de um módulo no Diagrama de Transição de Estados, seu uso não é permitido em nenhum lugar;
- *current_state*: função derivada que permite consultar o valor de *ctl_state*, seu uso é permitido apenas para leitura e é visível externamente ao módulo;
- *InsternalSate*: tipo enumerado que define os valores que *ctl_state* pode assumir;
- *initialState*: valor padrão do tipo *InternalState*;
- *atv*: indica o nome da ação que representa a atividade de um determinado estado;
- *GlobalControl*: é utilizada como nome da abstração que representa o Controle Global.

3.1.7 Documentação

Um Modelo Básico contém todos os registros do problema e da solução adotada, primeiramente sendo uma descrição textual e posteriormente formalizada na notação de LMM. Portanto, nada mais natural que do Modelo Básico se extraia a documentação

completa do sistema. Essa é uma proposta de Knuth no trabalho sobre *Literate Programming* [Knu83]. Essa técnica traz a vantagem de sistema e documentação serem obtidos a partir da mesma origem, mantendo ambos compatíveis, consistentes entre si.

Segundo as diretivas de *Literate Programming*, todo problema deve ser solucionado por partes. Em Linguagem de Modelagem Machĭna, sugere-se que cada parte seja um estado, então, descreve-se o trecho do problema e suas respectivas características e soluções na Descrição Textual. O DTE pode ser mostrado com o foco no estado em questão, como será apresentado na Seção 3.3.3, definindo apenas seus fluxos de saída, deixando o modelo formal mais legível. Consequentemente, a parte do Glossário conterá somente as definições das atividades do estado e elementos dos fluxos mostrados, e que não tenham sido definidos anteriormente.

Esse processo deve ser realizado para cada módulo, e ao final, são apresentados os diagramas e os glossários completos. Depois de todas as especificações realizadas, descrevem-se as relações entre os Tipos Abstratos de Dados por meio do Diagrama TAD.

Dessa forma, a solução apresentada fica mais legível e concentrada nos estados e suas ações sem perder a visão geral de um sistema. Pela flexibilidade do Modelo Básico, novos requisitos podem ser facilmente incluídos e modificações podem ser facilmente realizadas. A utilização da técnica de *Literate Programming* garante a fidelidade da documentação perante as alterações. Os exemplos da Seção 3.2 mostram a aplicação desta técnica.

3.2 Exemplos de Sistemas

Para exemplificar a Linguagem de Modelagem Machĭna, apresentam-se a seguir alguns problemas conhecidos da computação. Embora alguns detalhes de refinamento sejam utilizados nos exemplos, eles apenas serão detalhados na Seção 3.5, onde o Método de Refinamento Machĭna é apresentado.

3.2.1 Pilha e Fila

Pilha e Fila FIFO são duas estruturas de dados básicas da computação com semelhanças estruturais e pequenas diferenças em suas operações. O objetivo deste exemplo é mostrar a definição de um Tipo Abstrato de Dados e como sua abstração permite criar essas duas estruturas de forma simples e elegante.

Pilha

De acordo com [LSRC02], uma pilha é um conjunto dinâmico no qual o elemento removido é o mais recentemente inserido, norma conhecida como $LIFO$ - (*Last In - First Out*). Pode-se definir um tamanho máximo permitido no conjunto e pode ser necessário verificar se a pilha está vazia ou cheia. Então, definem-se os seguintes elementos de uma pilha:

- *elements:* elementos pertencentes a pilha;
- *n:* tamanho atual da pilha;
- *max:* tamanho máximo permitido.

Uma pilha possui as operações básicas de inserção e retirada de um elemento e verificação se está cheia ou vazia, então definem-se as seguintes operações:

- *insert(x):* insere o elemento x na pilha;
- *remove(x):* remove um elemento da pilha e o coloca em x;
- *isFull(t):* verifica se a pilha está cheia e coloca o resultado, verdadeiro ou falso, em t;
- *isEmpty(t)* verifica se a pilha está vazia e coloca o resultado, verdadeiro ou falso, em t.

Pela definição de uma pilha, o tamanho corrente n não pode ser maior que o tamanho máximo max definido e como não existe conjunto com cardinalidade menor que zero, n deve ser sempre maior ou igual a zero. Assim, a Figura 3.17 mostra a representação do TAD em Linguagem de Modelagem Machĭna para uma pilha. É possível perceber que são descritos apenas os elementos necessários para utilizar uma pilha, sem se importar com operações internas.

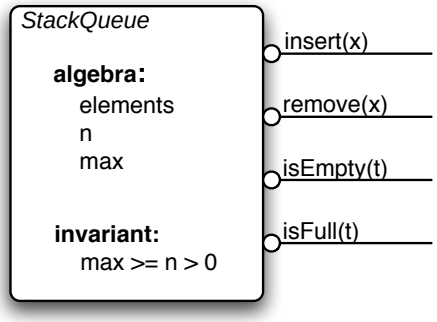


Figura 3.17: TAD de uma pilha.

Para fins de implementação, devem ser detalhados os elementos e operações no Glossário. Os itens *elements*, tamanho n e tamanho máximo *max* são associados a um tipo conforme a sintaxe de Machĭna. Para fins de simplicidade, definiu-se uma pilha de inteiros. Duas novas funções são incluídas, *empty* avalia se a pilha está vazia e *full* se está cheia. Elas são para utilização interna, por isso não fazem parte da descrição do TAD.

Glossário : Álgebra

```
1  elements : list of Int;
2  n, max   : Int;
3
4  derived empty : Bool := n = 0;
5  derived full  : Bool := n = max;
```

Na interface do TAD da pilha, foram definidas quatro operações. Primeiro serão detalhadas as operações que provêm informações sobre o preenchimento da pilha, então criam-se as abstrações *isEmpty* e *isFull*. Pode-se notar o uso interno das funções derivadas *empty* e *full*.

Glossário : Abstrações

```
1  action isEmpty(out t : Bool) is
2    t := empty;
3  end
4
5  action isFull(out t : Bool) is
6    t := full;
7  end
```

Por fim, são definidas as duas operações que realmente caracterizam a estrutura como pilha. A operação *insert* coloca o elemento x recebido por parâmetro no começo da lista *elements* e incrementa a cardinalidade n. Como pré-condição, exige-se que a pilha não esteja cheia, e garante-se na pós-condição que a adição de um elemento na pilha aumenta a cardinalidade em um, onde *old* se refere ao valor de n no estado de origem da transição. A operação *remove* retira o elemento que está no começo da lista *elements* e decrementa a cardinalidade n. Como pré-condição, a pilha não pode estar vazia, e deve-se garantir que a remoção de um elemento da pilha diminui a cardinalidade em um.

Glossário : Abstrações

```
8   action insert(in x : Int) is
9     require : not full;
10    ensure  : n = (old n) + 1;
11
12    elements := x :: elements;   n := n+1;
13  end
14
15  action remove(out x : Int) is
16    require : not empty;
17    ensure  : n = (old n) - 1;
18
19    x := head(elements);   elements := tail(elements);   n := n-1;
20  end
```

Com este exemplo é possível mostrar a facilidade de se expressar um módulo utilizando a Linguagem de Modelagem Machĭna em termos de um Tipo Abstrato de Dados. A abstração obtida na definição do TAD é suficiente para compreender a estrutura básica relevante de uma pilha e seus serviços disponíveis. Detalhes internos são encapsulados e somente tratados em um nível mais detalhado, separando claramente o objetivo do contrato de um TAD de suas características de implementação. O mesmo processo de programação por contrato é utilizado na definição de ações, onde seu nome, parâmetros formais, pré e pós condições mostram o seu propósito. Ao final tem-se uma especificação de um Tipo Abstrato de Dados legível com os níveis de abstração bem separados e objetivos.

Fila

A definição de Fila em [LSRC02] é semelhante a de uma pilha, sendo também um conjunto dinâmico, porém o elemento removido é o primeiro inserido, norma conhecida como *FIFO* - (*First In - First Out*). É possível aproveitar toda a definição de TAD e Glossário feita para pilha e apenas alterar a abstração *insert* para

Glossário : Abstrações

```
21  action insert(in x : Int) is
22    require : not full;
23    ensure  : n = (old n) + 1;
24
25    elements := elements :: x;   n := n+1;
26  end
```

onde percebe-se que o elemento é inserido no final da lista. Ao retirar um elemento na ação *remove*, o primeiro que entrou na fila será o escolhido.

Com essa simples alteração foi possível definir duas estruturas de dados básicas e muito utilizadas na computação. O alto nível de abstração dado à definição permite reutilização de módulos com os diferentes comportamentos de pilha e lista detalhados. Este exemplo mostra a facilidade de expressar elementos do mundo real em termos de TAD dentro de LMM com todas as propriedades necessárias à programação por contrato, fornecendo legibilidade e escalabilidade aos módulos assim definidos.

3.2.2 Pesquisa Binária

A Pesquisa Binária consiste em procurar por um elemento em um conjunto, armazenado como uma lista ordenada. A cada passo, compara-se o elemento com o registro que está na posição do meio da lista, chamado de pivô. Se o elemento for menor, então o registro procurado está na primeira metade, contendo os valores menores que o pivô. Se for maior, o registro procurado está na segunda metade, contendo valores maiores que o pivô. Este processo deve ser repetido até que se encontre o elemento procurado ou o conjunto acabe, indicando que a pesquisa não obteve sucesso.

Solução 1

Para a solução, assume-se que o conjunto para pesquisa já esteja ordenado, e sua cardinalidade n e o elemento procurado k sejam fornecidos. Então, criam-se dois delimitadores do subconjunto no qual deve-se procurar o elemento, sendo representado por *inf* para o limite inferior e *sup* para o limite superior. A posição que indica o pivô é obtida pela função *middle*. Essa função fornece a posição do meio considerando os limites inferiores e superiores. Na Pesquisa Binária, deve-se resguardar que o limite inferior nunca ultrapasse o limite superior, senão pode ocorrer situações indesejadas como erro no cálculo do pivô. Além disso, os limites devem ser maiores que zero e menores que a cardinalidade do conjunto utilizado na pesquisa.

A partir dos elementos básicos de um módulo que realiza a pesquisa binária, é possível definir o TAD conforme a Figura 3.18, onde são apresentados a estrutura básica interna e os invariantes.

A pesquisa inicia comparando-se o pivô com o elemento procurado, se for igual, a pesquisa obteve sucesso, caso contrário verifica se o final do conjunto foi alcançado comparando os limites inferiores com superiores. Caso verdadeiro, a pesquisa não obteve sucesso. Caso contrário, verifica se o elemento procurado é menor ou maior que o pivô. Se for menor, atualiza o limite superior para conter a posição do antecessor do pivô. Se for maior ou igual, atualiza o limite inferior para conter aposição do sucessor.

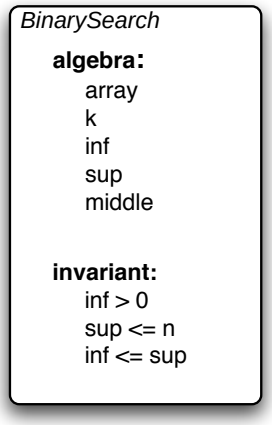


Figura 3.18: TAD para pesquisa binária.

Neste momento, o subconjunto é dividido pela metade, e deve-se obter o novo pivô atualizando a função *middle*. Então repete-se o processo até que seja alcançado um dos estados finais. A Figura 3.19 mostra o DTE para a Pesquisa Binária. Por se tratar de um problema pequeno, não dividiu-se o diagrama em partes, como sugere o Método de Refinamento Machĭna.

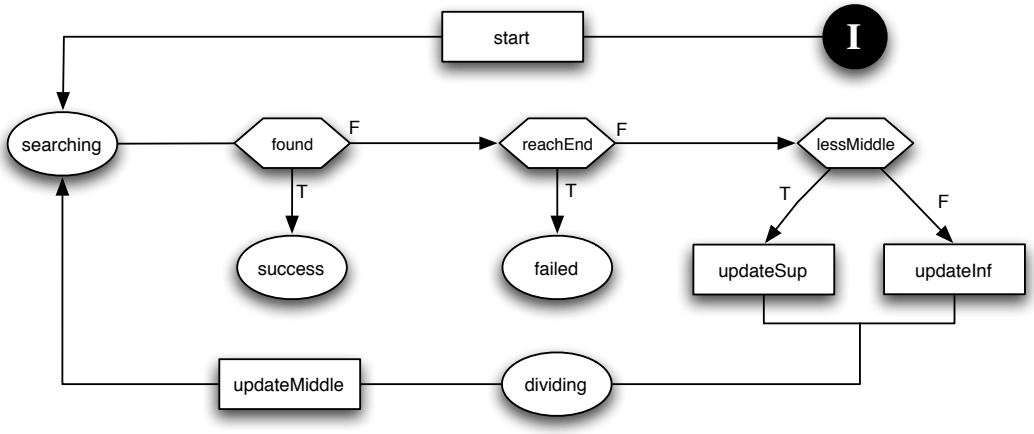


Figura 3.19: Diagrama FSM para Pesquisa Binária, primeira abordagem

A seguir, pode-se definir na Álgebra os elementos contidos no TAD e no DTE. Para fins de simplicidade, as funções *key*, *n* e *orderedArray* são definidas para serem utilizadas como entrada de dados para a chave a ser procurada, cardinalidade do conjunto e o conjunto ordenado.

Glossário : Álgebra

```
k, inf, sup, middle   : Int;
array : Int -> Int;

external key : Int;    external n : Int;
external orderedArray : Int -> Int;

derived found      : Bool := array(middle) = k;
derived reachEnd   : Bool := inf = sup;
derived lessMiddle : Bool :=  k < array(middle);
```

As ações apresentadas no diagrama e explicadas anteriormente são definidas conforme Abstrações no Glossário. Percebe-se que o conjunto, sua cardinalidade e a chave a ser localizada são inicializadas conforme a entrada de dados pela função externa correspondente.

Glossário : Abstrações

```
action start is
  array := orderedArray;
  k := key;    inf := 1;    sup := n;    middle : = (1+n)/2;
end

action updateInf is
  inf = middle + 1;
end

action updateSup is
  sup = middle - 1;
end

action updateMiddle is
  middle = (inf+sup)/2;
end
```

Solução 2

O Modelo Básico permite especificar o problema em níveis de abstração diferentes. Esta segunda solução abstrai do diagrama a verificação do pivô para atualizar os limites. Então, onde havia a condição *lessMiddle* existe a ação *updateInfSup*, onde o teste a e atualização dos limites são realizados. A Figura 3.20 ilustra o DTE com esta nova interpretação. O TAD mostrado na solução anterior permanece exatamente o mesmo.

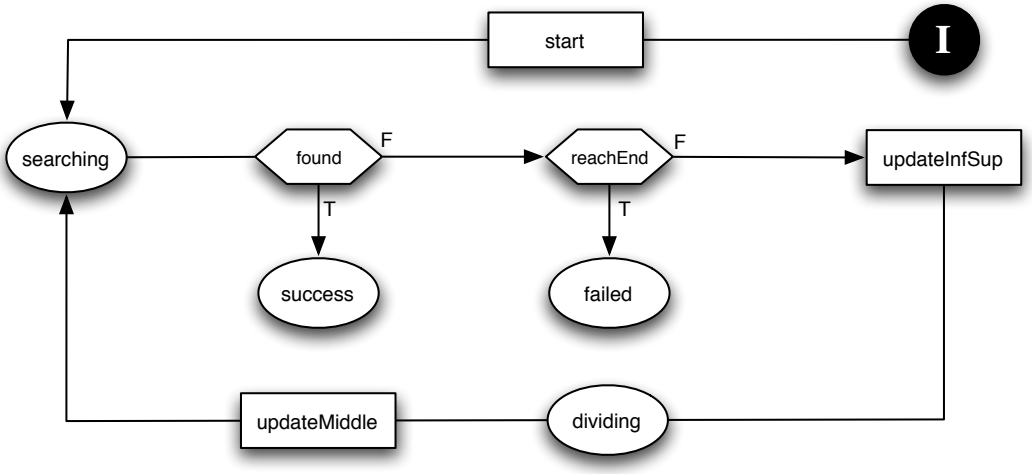


Figura 3.20: Diagrama FSM para Pesquisa Binária, segunda abordagem.

A seguir, é mostrada a inclusão da ação *updateInfSup* em Abstrações, preservando as demais definições do Glossário. Pode-se notar que o teste e atualização dos limites foram transferidos para a ação *updateInfSup*, linha 2.

Glossário : Abstrações

```
1  . . .
2  action updateInfSup is
3    if lessMiddle then
4      updateSup;
5    else
6      updateInf;
7    end
8  end
```

Na primeira solução, foi dada ênfase na atualização dos limites de acordo com a verificação se o elemento procurado é menor ou maior que o pivô. Esta segunda solução não aborda este item no Diagrama de Transição de Estados, deixando essa análise para o Glossário. Como pode ser visto, ambas soluções possuem o mesmo funcionamento, e se diferenciam no nível de abstração do Diagrama de Transição de Estados, mostrando o que interessa de acordo com um determinado contexto.

3.2.3 Jantar dos Filósofos

O problema do Jantar dos Filósofos é um problema clássico da ciência da computação que ilustra concorrência. Consiste de um grupo de filósofos que podem estar em três

situações diferentes: pensando, comendo ou com fome. Os filósofos estão dispostos em uma mesa circular e para comer eles devem estar de posse de dois garfos, um à esquerda e outro à direita, que são compartilhados com seus vizinhos. Assim, quando um filósofo come, seu vizinho da direita e o da esquerda devem esperar.

Quando um filósofo acaba de comer, ele libera os garfos e passa a pensar. No momento em que sente fome, ele aguarda que os garfos da esquerda e direita estejam liberados para ele comer. Deve-se impedir que todos os filósofos peguem somente o seu garfo, por exemplo, da esquerda, pois isso irá fazer com que todos fiquem aguardando o outro garfo ser liberado e ocorre uma situação de *deadlock*. Pode ser desejável controlar a situação de *starvation*, onde um filósofo nunca consegue a vez para comer. A seguir, são apresentadas duas soluções para o problema do Jantar dos Filósofos utilizando Linguagem de Modelagem Machĭna.

Solução 1

Para a primeira solução, são criados dois módulos, o primeiro representa um filósofo e o segundo, chamado de *host*, define a configuração do sistema.

Módulo *Philosopher*: descreve as ações e estados de um filósofo. A partir do estado inicial, coloca-se o filósofo para pensar (estado *thinking*). Após um tempo não determinado, o filósofo começa a sentir fome (estado *hungry*), e então, tenta pegar os garfos enquanto permanecer neste estado. Se os garfos estiverem liberados, o filósofo os pega e começa a comer (estado *eating*), se não, aguarda. Após um determinado tempo, o filósofo termina de comer e libera os garfos que utilizava e passa a pensar novamente. A Figura 3.21 mostra o Diagrama de Transição de Estados para o módulo correspondente.

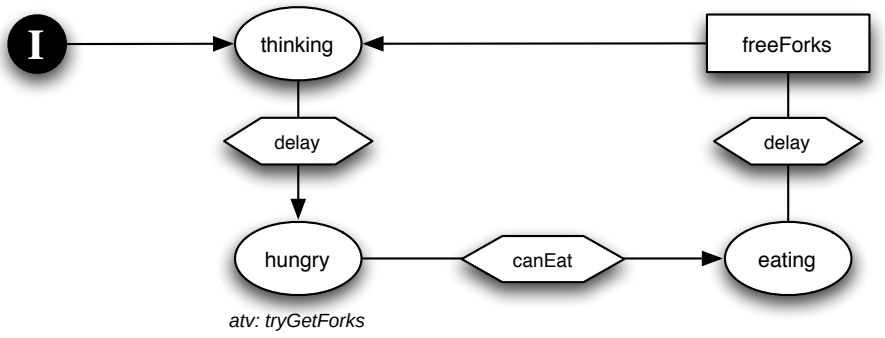


Figura 3.21: Diagrama de Transição de Estados para *Philosopher*.

A estrutura interna de *Philosopher* deve conter dois indicadores de uso dos garfos, um para o da esquerda e o outro para o da direita. *Host* identifica o agente de *Host*

que controla o sistema. As ações *setRighFork* e *setLeftFork* são definidas para atribuir as referências aos garfos da direita e esquerda de um filósofo, assim como *setHost* determina o identificador do agente de *Host*. Deseja-se que isso seja feito pelo agente de *Host*, então, colocam-se essas ações na Interface. Com isso, apresenta-se o TAD para *Philosopher* na Figura 3.22.

Uma condição importante é jamais pegar somente um garfo enquanto aguarda o outro. Ambos devem ser tomados ao mesmo tempo, ou seja, o valor de *leftFork* deve ser o mesmo de *rightFork*. Essa é uma propriedade que deve ser garantida para todos os filósofos, e será discutida na Seção 3.3.4.

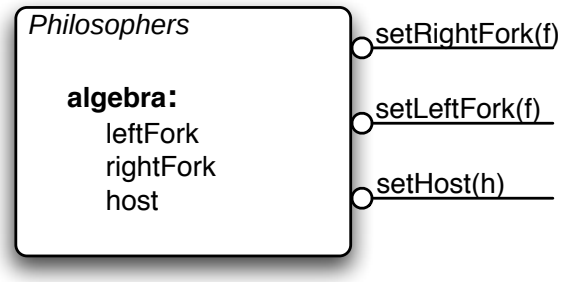


Figura 3.22: TAD para filósofo.

A seguir, são feitas as definições dos elementos presentes no TAD e DTE de *Philosophers*. A condição de *delay* apenas controla o tempo que o filósofo pensa antes de comer e o tempo em que ele come. Para simplicidade do exemplo, será definida como uma função externa. O tipo *ForkId* é importado do módulo *Host*, que será definido posteriormente. A função *canEat* é uma definição para uso interno para auxiliar em algumas ações, como será visto na definição de Abstrações. A função *host* indica o agente de módulo *Host* que é responsável por coordenar a obtenção correta dos garfos pelos agentes de filósofos.

Glossário : Álgebra

```
1  leftFork, rightFork: ForkId;
2
3  canEat : Bool;
4  external delay : Bool;
5
6  host : agent of Host;
```

A tentativa de pegar os garfos é feita enquanto o filósofo permanece no estado *hungry*, então define-se a atividade *tryGetForks* para pegar os garfos necessários. Como o controle de acesso aos garfos é feito por *host*, utiliza-se o serviço *setForks* que este

disponibiliza. Os dois primeiros parâmetros são para informar quais são os garfos da esquerda e direita, respectivamente. A função *canEat*, terceiro parâmetro, conterá o retorno da avaliação dos garfos, permitindo ou não que o filósofo entre no estado *hungry*. Após comer, o filósofo solicita a liberação dos garfos, também controlada pelo *host*, informando seus garfos da direita e esquerda, e atribui o valor falso à sua permissão para comer, função *canEat*.

Assim, no Glossário, pode-se definir as seguintes Abstrações de *Philosophers*.

Glossário : Abstrações

```
1  action tryGetForks is
2    host.getForks(leftFork, rightFork, canEat);
3  end
4
5  action freeForks is
6    host.freeForks(leftFork, rightFork);
7    canEat := false;
8  end
9
10 public action setRighFork(in f: ForkId) is
11   rightFork := f;
12 end setRight
13
14 public action setLeftFork(in f: ForkId) is
15   leftFork := f;
16 end setLeft
17
18 public action setHost (in : h) is
19   self.host := h;
20 end
```

Módulo *Host*: tem o objetivo de preparar o sistema para execução definindo o número de agentes, inicializando seus elementos e posteriormente os disparando. São definidos cinco estados, além do estado inicial, numerados de um a cinco.

Para compor a estrutura interna deste módulo, define-se inicialmente um tipo de dado *ForkId* para caracterizar um garfo. Para referenciar todos os filósofos, é definido um agente de *Philosophers*, denominado *philosophers*. Como elementos auxiliares, são definidos três agentes de *Philosophers*, *p*, *t* e *first*, e o contador *agCount* para controle do número de agentes criados. A função *fork* indica se um grafo está ou não disponível. O tipo *ForkId* é disponibilizado na interface para ser utilizado por *Philosophers*. A ação *getForks* é um serviço disponibilizado para que um filósofo requisite a permissão

para comer de acordo com seus garfos. A resposta é dada na função *canEat*. A ação *freeForks* libera os garfos sob a solicitação de um filósofo. Todas essas informações estão retratadas no TAD da Figura 3.23.

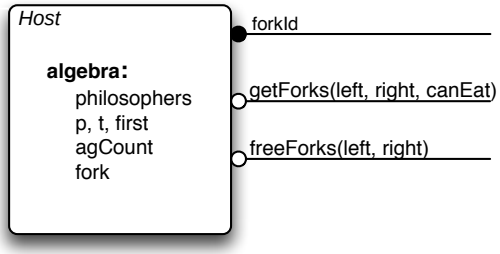


Figura 3.23: TAD para *host* de filósofo.

Primeiro, criam-se os agentes de filósofos de acordo com um número desejado. Isso é feito pela ação *createPhilosophers*. Formatam-se os dados do primeiro filósofo, ação *firstStep*, e em seguida, os demais filósofos são inicializados respeitando a ordenação da mesa e compartilhamento dos garfos, ações *secondStep* e *thirdStep*. O quarto passo, ação *forthStep*, termina as inicializações configurando o primeiro garfo fechando o ciclo da mesa dos filósofos.

Por último, um agente de *Host* dispara os agentes de filósofos na ação *dispatchPhilosophers*. Nota-se que ela é a atividade do estado 5, reconhecido como final por não ser origem de nenhum fluxo. Sendo assim, a atividade é executada uma única vez e o agente de *Host* termina sua execução, deixando os processos de *Philosophers* ativos e aguardando por solicitações de serviços. A Figura 3.24 mostra o DTE para o módulo *Host*.

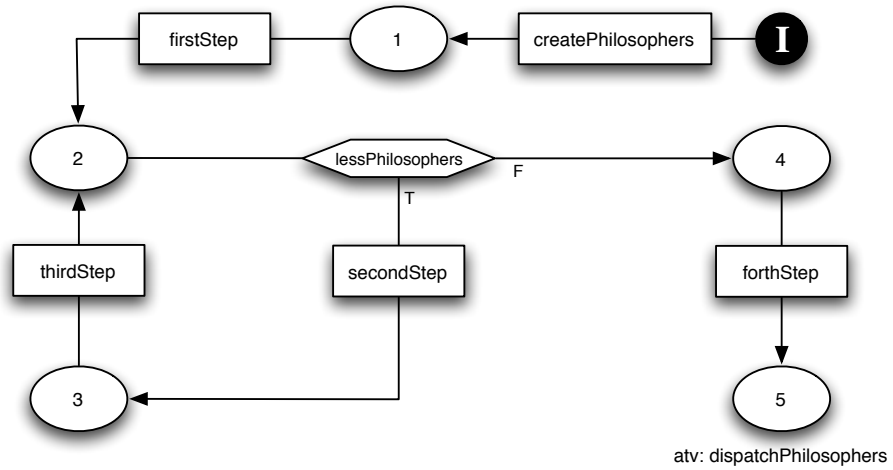


Figura 3.24: Diagrama de Transição de Estados para *Host*.

Define-se a função *numberOfGuests*, que determina externamente o número de convidados a sentar à mesa. Por último, a função *lessPhilosophers* verifica se o número de filósofos criado é menor que o número de convidados desejado. A seguir, apresenta-se a Álgebra com as definições dos elementos do TAD e as funções adicionais auxiliares. Pode-se notar que *philosophers*, *p*, *t* e *first* são declarados como agentes de *Philosophers*, via **agent of**. A criação do agente e o disparo de sua regra de transição dentro deste módulo são posteriormente realizados, por meio das diretivas **create** e **dispatch**, respectivamente. No momento da criação de um agente, pode-se definir se ele representa um único agente, como é o caso de *p*, *t* e *first*, ou como uma coleção de agentes, como é o caso de *philosophers* na ação *createPhilosophers*.

Glossário : Álgebra

```
1  type ForkId = Int;
2  philosophers : agent of Philosophers;
3  p, t, first  : agent of Philosophers;
4  agCount : Int;
5
6  fork : ForkId -> Bool;
7
8  external numberOfGuest   : Int;
9  derived lessPhilosophers : Bool is
10     agCount < philosophers.numberOfAgents;
```

A seguir, encontram-se as definições das ações que estão representadas no DTE e as disponibilizadas na interface de *Host*. O serviço *getForks* avalia a disponibilidade dos garfos do filósofo que solicitou a permissão para comer e o resultado é dado no parâmetro *canEat*. Conforme a definição de Machĭna, todo pedido de um filósofo é síncrono, de forma que este agente não realiza a próxima transição de estado enquanto *host* não responder. Quanto ao serviço *freeForks*, foi visto que o próprio agente de filósofo cuida da função *canEat* quando libera os garfos, assim não existe o bloqueio e a liberação é realizada de forma assíncrona. A abstração *setForks* é uma ação auxiliar que reserva ou libera os garfos, sempre ao mesmo tempo. Pode-se perceber que algumas abstrações utilizam ações pertencentes ao módulo *Philosophers*, por exemplo *setLeftFork* e *setRightFork*. O controle de utilização dos serviços disponibilizados por ambos os módulos será formalizado mais adiante.

Glossário : Abstrações

```
1  action createPhilosophers is
2    create philosophers : agent of Philosophers(numberOfGuests);
3  end
```

```
4
5   action firstStep is
6     p       := philosophers(1);
7     first   := p;
8     agCount := 2;
9     forkId  := 2;
10    p.setHost(self);
11  end
12
13  action secondStep is
14    t := philosophers(agCount);
15    t.setHost(self);
16    agCount := agCount + agCount;
17  end
18
19  action thirdStep is
20    p.setRightFork(forkId);       t.setLeftFork(forkId);
21    forkId := forkId + 1;         fork(forkId) := false;
22    p := t;
23  end
24
25  action forthStep is
26    fork(1) := false;    p.setRightFork(1);   firt.setLeftFork(1);
27  end
28
29  action dispatchPhilosophers is
30    forall p : philosophers do   dispatch p;   end
31  end
32
33  action setForks(in left : ForkId, in right : ForkId, in v : Bool) is
34    fork(left)  := v;
35    fork(right) := v;
36  end
37
38  public action freeForks(in left : ForkId, in right : ForkId) is
39    setForks(left, right, false);
40  end
41
42  public action getForks(in left : ForkId, in right : ForkId,
43                         out canEat : Bool) is
44    if not(fork(left) or fork(right)) then
45      setForks(leftFork, rightFork, true);
46      canEat := true;
47    end
48  end
```

Mecanismo de visibilidade: Agentes de ambos os módulos utilizam serviços que foram disponibilizados em suas interfaces. Utiliza-se o diagrama TAD indicando o tipo de relacionamento entre *Philosophers* e *Host*. A Figura 3.25 mostra, por meio da seta bidirecionada, que um módulo importa a interface do outro. A caixa cinza indica que a máquina a ser criada, que representa o sistema, é em função de *Host*.

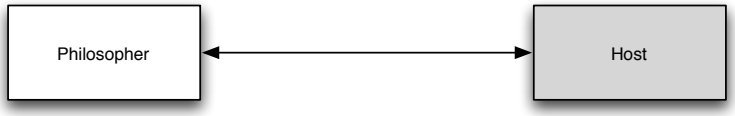


Figura 3.25: Diagrama TAD.

Controle de concorrência

Todos os agentes de *Philosophers* executam suas regras de transição em paralelo e têm os garfos como recursos compartilhados. Quando um filósofo tenta obter os garfos para comer ele não o faz diretamente, é feita uma requisição ao agente *host* pela abstração *getForks*. O agente filósofo então, espera pela resposta para poder continuar. O agente *host* recebe todas as requisições e, ao final de cada transição, as executa atomicamente na ordem em que estão na lista de requisições. O resultado é fornecido na função *canEat* e, caso não tenha recebido a permissão, o filósofo tenta novamente na próximo interação, caso contrário, seu estado é alterado para *eating*. Com isso, o controle é realizado pela sincronia das requisições dos filósofos e o processamento em ordem e atômico delas pelo *host*.

Os dois garfos são reservados para um filósofo de uma só vez, no mesmo ciclo da regra de transição, com isso o problema de *deadlock* é resolvido. Porém, é possível que um filósofo fique eternamente no estado *hungry* sem obter os garfos para comer, essa situação, conhecida como *starvation*, é resolvida na Solução 2 a seguir.

Solução 2

A primeira solução apresentada garante que não ocorrerá *deadlock*, porém não considera o problema de *starvation*. Para garantir que todos os filósofos possam comer, deve-se criar uma lista de prioridades, onde mais fome significa maior prioridade. Esse controle pode ser realizado com a criação da função *hungryLevel*, que contabiliza o nível de fome de um filósofo, sendo incrementada a cada interação em que se permanece no estado *hungry*. Quando tenta-se pegar os garfos deve-se verificar também se existe algum outro filósofo com o nível de fome maior, se não existir, lhe é concedido o direito de comer.

Se existirem dois filósofos com mesma prioridade disputando por um mesmo garfo, o primeiro na lista de requisições será atendido, e no estado seguinte o outro terá mais prioridade, garantindo que em um determinado momento lhe será concedido o direito de comer. Esta verificação é feita no *host*, então deve-se passar como parâmetro para *getForks* o nível de fome do filósofo que faz a requisição.

No módulo *Philosophers*, é acrescentada a função *hungryLevel*, que é sempre incrementada enquanto se permanece no estado *hungry*. Ao liberar os garfos, deve-se zerar o nível de fome e ao solicitar a permissão para comer, deve-se informar o nível de fome atual. No módulo *Host*, a abstração *getForks* é acrescida do parâmetro *hungryLevel* e além de verificar os garfos, verifica-se o nível de fome em relação aos demais filósofos. Percebe-se que o DTE dos dois módulos são exatamente os mesmos da Solução 1.

As alterações em *Philosophers* são:

Glossário : Álgebra

```
1  hungryLevel : Int;
```

Glossário : Abstrações

```
1
2  action tryGetForks is
3    host.getForks(leftFork, rightFork, canEat);
4    hungryLevel := hungryLevel + 1;
5  end
6
7  action freeForks is
8    host.freeForks(leftFork, rightFork);
9    canEat := false;
10   hungryLevel := 0;
11 end
```

As alterações em *Host* são:

Glossário : Abstrações

```
1  public action getForks(in left : ForkId, in right : ForkId,
2                         in hungryLevel, out canEat : Bool) is
3    if not(fork(left) or fork(right)) and
4       not(exist p : philosophers
5           satisfying p.hungryLevel > hungryLevel) then
6      setForks(leftFork, rightFork, true);
7      canEat := true;
8    end
```

```
9 end
```

As alterações realizadas na Solução 2 garantem que todos os filósofos terão sua vez de comer, eliminando o problema de *starvation*. Percebe-se que foram alterados apenas elementos do Glossário, de forma que o Diagrama de Transição de Estados de cada módulo permanece o mesmo. Assim, o tratamento de *starvation* é transparente para quem trabalha com o diagrama, preservando a interpretação do problema feita no Domínio da Aplicação.

3.3 Recursos Avançados da Linguagem de Modelagem Machĭna

Na Seção 3.1 foram apresentados os conceitos básicos da Linguagem de Modelagem Machĭna, no qual é possível realizar a definição completa de um sistema. Porém, existem outros elementos na LMM responsáveis por descrever características do sistema com maiores detalhes.

Um módulo de definição pode conter um conjunto de restrições a serem respeitadas durante a execução do sistema. Em LMM, esse conjunto é definido como invariantes, conforme mostrado na Seção 3.1.2, e qualquer falha em sua avaliação a execução é interrompida. De forma semelhante, é possível definir restrições a estados e ações para garantir o correto funcionamento e auxiliar verificação de propriedades do sistema.

3.3.1 Propriedade de Estado

Todo estado, exceto o inicial, pode estar associado a uma condição que deve ser satisfeita durante a permanência do agente de um módulo neste estado. Caso a condição não seja satisfeita, a execução é interrompida em condição erro. A notação para propriedade de estado é dada conforme a Figura 3.26, onde C_i é expressão booleana representando a propriedade do estado i.

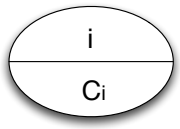


Figura 3.26: Representação de uma propriedade de estado.

Quando um agente chega a um estado i, deve-se verificar, se existir, o invariante C_i. Caso seja satisfeito, prossegue-se com a execução normal. Se permanecer no mesmo

estado, o invariante é novamente verificado na interação seguinte. Caso a avaliação da propriedade falhe, a máquina é interrompida em condição de erro.

3.3.2 Pré e Pós Condições de Ações

Uma ação A é executada sob certas condições que podem ser explicitamente definidas no Modelo Básico na forma de pré-condição e pós-condição. A pré-condição representa uma expressão booleana R_A que deve ser satisfeita imediatamente antes da execução de A. Em LMM, a avaliação de R_A é feita com os valores presentes no estado origem do fluxo na qual A pertence. A pós-condição representa a expressão booleana E_A a ser satisfeita após a execução de A e a avaliação de E_A é feita com os valores do estado de destino. A seguir, é apresentada na Figura 3.27 a notação para pré e pós condições de uma ação.

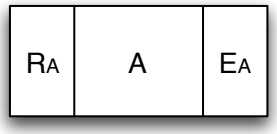


Figura 3.27: Pré e Pós condições de uma ação.

No caso de falha de alguma dessas condições, a ação é considerada inválida e a máquina termina sua execução com erro.

Esta definição diz respeito à restrições em determinados pontos de uma transição de estado, e assim como invariantes (Seção 3.1.2), se difere da definição do Controle Global no ponto de vista em que verificam situações de erros. Invariante e Pré-Pós Condições de Ações são voltados a problemas no Domínio dos Modelos, enquanto o Controle Global intercepta problemas ligados ao Domínio da Aplicação e é possível tomar medidas para corrigi-los e retomar a execução normal.

Um exemplo dessa diferença pode ser visto no problema da Caldeira a Vapor [BBD+96]. Dado um nível de água n, a restrição $N_1 < n < N_2$ define os limites inferior e superior de funcionamento normal. Caso estes limites sejam ultrapassados, o sistema toma medidas para recuperá-los. Os limites $M_1 < n < M_2$ definem os níveis críticos de água em que a máquina deve parar imediatamente o funcionamento. Como estas são restrições definidas na aplicação, elas podem ser tratadas pelo Controle Global, como mostra a Figura 3.28 a seguir.

Com isso, deve-se definir a propriedade que relaciona os limites normais e críticos, devendo manter os limites normais dentro dos limites críticos. Essa definição está ligada ao domínio dos modelos e pode ser definida da seguinte forma:

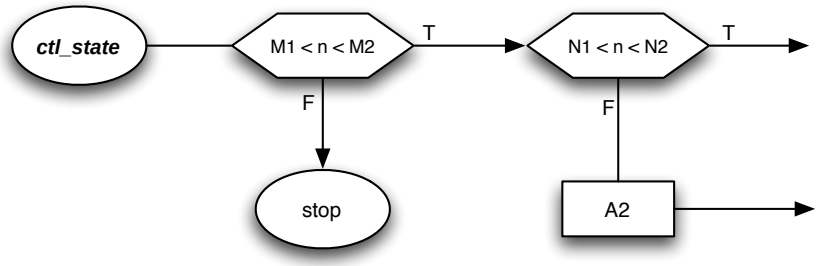


Figura 3.28: Controle Global da Caldeira a Vapor.

Propriedades da Caldeira a Vapor

```
1  invariant:
2    M1 < N1;
3    M2 > N2;
```

Embora seja uma observação trivial, ela é importante para garantir a modelagem correta e auxiliar verificação do sistema.

3.3.3 Diagrama de Saída de Estado

Uma especificação de um módulo em LMM pode conter um número considerável de estados e demais elementos de um Diagrama de Transição de Estados. Adicionalmente, pode-se acrescentar propriedade de estado e pré e pós condições de ações. Todos esses elementos podem resultar em um diagrama denso, que dificulta a legibilidade e compreensão do Diagrama de Transição de Estados e consequentemente do problema especificado.

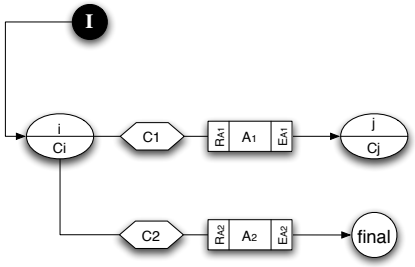
Um Diagrama de Saída de Estado (DSE) é um trecho de um Diagrama de Transição de Estados (DTE) no qual o foco é dado a um único estado i e as transições que possuem origem em i. A notação e semântica de Diagrama de Saída de Estado é a mesma de DTE.

Uma especificação de um módulo LMM contém um DSE para cada estado identificado, com exceção do estado inicial devido sua simplicidade, estando ele incluso no DSE de seu estado de destino. Os estados finais de um DTE não são origem de nenhum fluxo, conforme sua definição, portanto não existirão DSEs para eles.

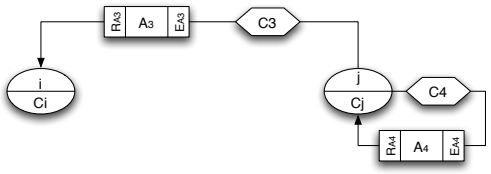
O Diagrama de Saída de Estado de um estado i é denominado DSEi e é construído a partir das transições que possuem i como origem e todos seus elementos. Atividades, propriedade do estado i e pré e pós condições das ações são desejáveis neste diagrama. Todo estado que se relaciona com i, ou seja, todo estado que é destino de uma transição com origem em i, estará presente no DSEi apenas como destino de transição e apresentando apenas informações necessárias para a compreensão deste trecho de espe-

cificação. Ao final, todos os DSEs são unidos para formar o DTE principal, que pode ser visualizado com menos detalhes para fins de legibilidade e interpretação global do módulo.

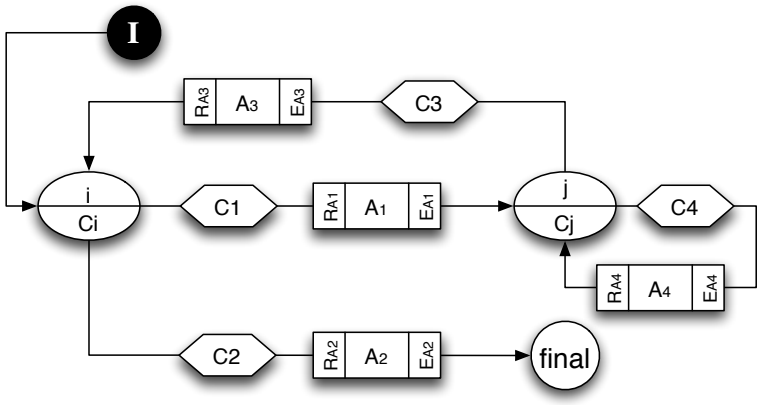
Figura 3.29 mostra de forma simples um conjunto de DSEs formando o DTE completo. Existem três estados, *i*, *j* e *final*. O DSE para o estado *i*, Figura 3.29(a), mostra sua propriedade *Ci* e os fluxos de saídas, e o mesmo ocorre para o estado *j*, Figura 3.29(b). Percebe-se que no DSEj, o estado *final* não aparece pois não existe transição de *j* para *final*. E o estado *final* não apresenta DSE por ser um estado final. O estado inicial é indicado no DSEi e no DTE comleto, Figura 3.29(c).



(a) DSE para o estado *i*



(b) DSE para o estado *j*.



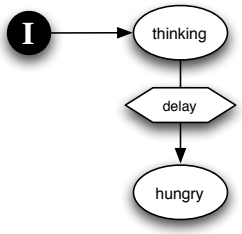
(c) DTE completo.

Figura 3.29: Formação do DTE.

Essa decomposição permite especificar o problema em partes menores, na qual concentra-se a atenção em um único estado e ações pertencentes aos seus fluxos de saída. Então, um módulo não é visto como um grande bloco de ações e sim como um conjunto de pequenos blocos, mais simples de serem gerenciados e compreendidos [Knu83].

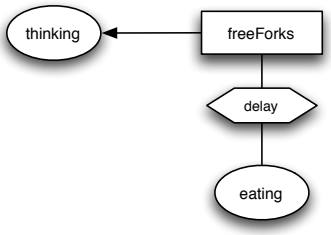
A solução de um problema pode ser dividida de várias formas diferentes, e isso depende do modelo de especificação. A noção de separação em LMM é baseada nos estados, que devem ser identificados de acordo com interpretações de nomes do vocabulário. Assim, um estado representa no mundo real um conjunto de tarefas a serem

realizadas. Por exemplo, os estados de um agente filósofo, representado no problema Jantar dos Filósofos na Seção 3.2.3, são pensando (*thinking*), com fome (*hungry*) e comendo (*eating*). Cada um representa uma atividade que um filósofo deve realizar de acordo com as condições (interpretação) de seus atributos (vocabulário). Figura 3.30 representa o DSE de cada estado do módulo para *Philosophers*. O estado *thinking* é representado na Figura 3.30(a), o estado *hungry* na Figura 3.30(b) e o estado *eating* na Figura 3.30(c). Pode-se notar que a atividade *tryGetForks* do estado *hungry* somente aparece em seu DSE.



(a) DSE*thinking*

(b) DSE*hungry*.



(c) DSE*eating*.

Figura 3.30: DSEs para módulo de Filósofo.

Com essa definição, os estados de um DTE representam pontos ideais para observação no sistema e execução de tarefas. Por isso, um Diagrama de Saída de Estado analisa um estado de cada vez e todas as suas transições, onde apresenta um número menor de elementos permitindo que sejam apresentadas mais informações, sendo então um diagrama mais detalhado, porém com visão parcial do problema. Isso significa ter um nível de abstração mais baixo que um DTE (que tem visão global do problema), conforme desejável no Modelo Básico [Bör03b]. A união de todos os DSEs forma o Diagrama de Transição de Estados completo, fornecendo a visão global da regra de transição de um módulo, e é este DTE que é utilizado no refinamento e obtenção da documentação.

3.3.4 Propriedades

Embora seja possível, utilizando Linguagem de Modelagem Machĭna, especificar em alto nível um sistema, existem ainda propriedades no Domínio da Aplicação que devem ser garantidas no modelo e preservadas no refinamento. Então, é definido em LMM uma maneira de se expressar propriedades considerando as transições de estados.

Uma notação muito utilizada em semântica formal é a CTL (*Computational Tree Logic*) [CGP00, Capítulo 3]. CTL é uma lógica temporal que utiliza preposições em suas construções para fazer afirmações em um sistema de transição de estados. Essas preposições são especificadas em termos de operadores lógicos e temporais. Em LMM, utilizam-se as funções da álgebra do módulo para escrever as fórmulas CTL. É importante ressaltar que Propriedades não utiliza sintaxe Machĭna em sua definição, portanto as palavras chaves desta linguagem, e também de Linguagem de Modelagem Machĭna, não são válidas neste trecho. Assim é possível utilizar as palavras chaves definidas em CTL sem problemas de colisão.

3.3.4.1 Operadores Lógicos

Os operadores lógicos em CTL são os comumente conhecidos. São apresentados em sua notação utilizada em LMM, e sua notação usual é descrita entre parênteses, se for diferente.

- ! ($\neg$) : negação;
- & ($\wedge$) : conjunção;
- xor ($\odot$) : exclusão mútua;
- | ($\vee$) : disjunção;
- $->$ ($\rightarrow$) : implicação;
- $<->$($\leftrightarrow$) : dupla implicação;
- *true* : expressão verdadeiro;
- *false* : expressão falso.

3.3.4.2 Operadores Temporais

Os operadores temporais podem ser classificados em operadores que atuam sobre os possíveis caminhos e os que atuam sobre os estados. Entende-se caminho como uma sequência de estados válidos. A seguir, são definidos os dois grupos de operadores de

acordo com a definição de CTL, e adicionalmente foram incluídos operadores convenientes à especificação de Linguagem de Modelagem Machĭna.

Operadores de Caminho

Dada uma preposição p, os operadores de caminho são:

- **A** p : preposição p deve ser garantida em todos os caminhos a partir do estado corrente;
- **E** p : deve existir pelo menos um caminho partindo do estado corrente no qual a preposição p é garantida.

Operadores de Estado

Considerando p e q como preposições, os operadores de estado são:

- **X** p : preposição p deve ser garantida no estado seguinte;
- **G** p : preposição p deve ser garantida em todos os estados seguintes;
- **F** p : preposição p deve ser garantida em algum estado seguinte;
- p **U** q : preposição p deve ser garantida até que q aconteça, com a garantia de que q será avaliada.

3.3.4.3 Outros Operadores

Para facilidade de expressão, cria-se um operador que interage sobre um determinado conjunto finito de elementos podendo ser definido da seguinte forma

forall

```
1  forall e in {low ..  high} :  CTL_formule_using_e
```

Considerando e como um índice, *low* e *high* como limites inferior e superior, respectivamente, e *CTL_formule_using_ e* como sendo uma fórmula CTL, onde e é índice de alguma função. Por exemplo, dada a propriedade a seguir

```
1  forall e in 0 ..  2 :  p(e)
```

é equivalente a seguinte conjunção, sendo *p(e)* um predicado.

```
1  p(0) and p(1) and p(2)
```

Em CTL, sempre utiliza-se um operador de cada grupo, sendo o primeiro de caminho seguido de um operador de estado. Por exemplo, no problema do Jantar dos Filósofos, um filósofo deve pegar, e liberar, os dois garfos ao mesmo tempo para evitar situação de *deadlock*, onde todos os filósofos têm um garfo na mão e espera pelo outro, que nunca será liberado pelo seu vizinho. Essa propriedade pode ser escrita no módulo *Host* da seguinte forma:

Jantar dos Filósofos: verifica *deadlock*

```
1  forall p in 0..numberOfGuests :
2          AG(fork(philosophers(p).leftFork) = fork(philosophers(p).rightFork))
```

Pode-se entender essa propriedade da seguinte forma: em todos os caminhos possíveis (operador A) a partir do estado corrente, *fork(leftFork) = fork(rightFork)* deve ser verdade em todos os estados seguintes (operador G), ou seja, *fork(leftFork) = fork(rightFork)* é sempre verdade.

Outra propriedade neste problema é que se um filósofo estiver com fome, estado *hungry*, ele deve comer, estado *eating*, alguma vez no futuro. Com isso pode-se verificar situação de *starvation*. Pode-se escrever essa propriedade como

Jantar dos Filósofos: verifica *starvation*

```
1  AG(current_state = hungry -> AF(current_state = eating))
```

Essa propriedade é lida da seguinte forma: é sempre verdade que se o estado interno de um filósofo for *hungry*, em todos os caminhos seguintes o estado será *eating* em algum ponto no futuro.

Assim, pode-se verificar se o problema do Jantar do Filósofo respeita essas duas propriedades e, no caso de falha, sabe-se qual falha ocorreu e é possível obter um exemplo que mostre os problemas. A verificação é discutida com mais detalhes na Seção 4, e será mostrado como as propriedade são avaliadas e como os exemplos são obtidos.

3.4 Validação

A validação no Método de Refinamento Machĭna é realizada de forma manual. Devido ao alto grau de abstração e legibilidade de DTEs e DSEs, é possível realizar simulações que garantam uma aceitação satisfatória do sistema no momento da especificação. As propriedades identificadas são facilmente expressas em CTL, Seção 3.3.4, que é uma

maneira legível, porém formal, de transportá-las ao longo do refinamento para auxiliar na verificação.

A própria construção do Modelo Básico MRM define os principais elementos, os pontos de observação e, durante o refinamento, as propriedades implícitas à LMM são formalizadas, definindo-se assim as relações entre o modelo abstrato e a implementação em Machĭna.

Deseja-se, futuramente, definir um método automático para validação, onde, dada uma determinada entrada de dados, podem ser feitas simulações no Modelo Básico e obter respostas que podem ser comparadas com os resultados obtidos com a execução do sistema implementado. Como Linguagem de Modelagem Machĭna é baseada em diagramas, pode ser útil desenvolver uma ferramenta que ilustre a execução por meio de animação do DTE e DSEs da especificação. Assim, será possível realizar a depuração do sistema e obter melhores resultados no que diz respeito à extração de informações do Modelo Básico e avaliação dos requisitos necessários.

3.5 Regras de Refinamento Machĭna

Após a especificação do sistema utilizando a Linguagem de Modelagem Machĭna, o Modelo Básico é submetido ao processo de refinamento no qual fornecerá o código executável. As Regras de Refinamento Machĭna foram definidas de acordo com cada elemento da LMM e são baseadas nas descrições de Börger em [Bör03a, BS03], resumidas na Seção 2.2.2. Uma regra pode fornecer propriedades a serem garantidas ao longo do refinamento. Estas serão posteriormente verificadas utilizando um verificador de modelos apropriado. Caso alguma não seja garantida, significa um erro de especificação e pode-se mostrar um exemplo de execução que cause o problema.

Além de apresentar como é feito o refinamento, as Regras de Refinamento Machĭna definem formalmente a notação da LMM sob aspectos do modelo ASM. Portanto, é possível mostrar a semântica de cada elemento retirando qualquer ambigüidade na compreensão dos mesmos.

A seguir, é apresentada cada regra, na ordem em que deve ser aplicada, em termos de sua transformação no Modelo Básico e suas propriedades. A primeira atua sobre a definição de um TAD gerando a estrutura de um módulo de Machĭna. Em seguida, o Diagrama de Transição de Estados e Diagramas de Saída de Estado são refinados para indicar novos elementos e o comportamento do módulo. Posteriormente, o Glossário define (ou completa) o conteúdo dos blocos da estrutura e por fim, o Mecanismo de Visibilidade terá efeito sobre a formação dos módulos e interação entre seus agentes.

3.5.1 Tipo Abstrato de Dados

Um Tipo Abstrato de Dados é representado por um módulo de definição em Machĭna e sua interface. Para facilitar a leitura, um bloco do módulo pode ser referenciado neste texto apenas pelo seu nome em negrito, com a finalidade de diferenciá-lo das seções do TAD com os mesmos nomes.

Módulo em Machĭna

```
1  module nome-do-módulo
2    import elementos importados
3    include elementos incluídos
4    algebra:
5      funções e tipos
6
7    abstractions:
8      ações
9    initial state:
10     inicializações de funções dinâmicas
11   transition:
12     regras
13   invariant:
14     invariantes de execução
15 end nome-do-módulo
```

Interface de um Módulo em Machĭna

```
1  interface nome-do-módulo
2    declarações de tipos
3    declarações de ações
4  end nome-do-módulo
```

Primeiramente, um Tipo Abstrato de Dados define o nome que o módulo por ele especificado terá por meio do mapeamento direto de seu nome. Em seguida, a seção álgebra do TAD determina os principais elementos que caracterizam um módulo, sendo estes então parcialmente definidos no bloco **algebra**. A definição parcial significa conter apenas o nome ou a assinatura, sendo ainda necessária uma definição detalhada, que será realizada posteriormente em outra regra.

A seção de invariantes do TAD corresponde ao bloco **invariant**, portanto cada expressão booleana definida no TAD é diretamente mapeada em um invariante na linguagem alvo. Assim, um Tipo Abstrato de Dados, como o da Figura 3.31, define o módulo em Machĭna mostrado a seguir.

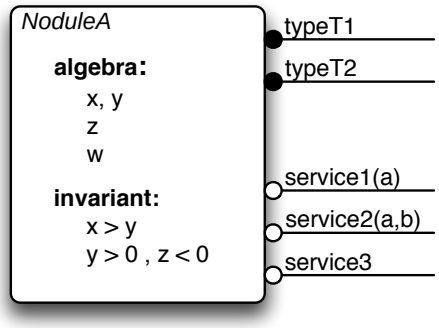


Figura 3.31: Exemplo de Tipo Abstrato de Dados.

Módulo em Machĭna

```
1  module ModuleA
2    algebra:
3      x, y;
4      z;
5      w;
6
7    invariant:
8      x > y;      y > 0;      z < 0;
9
10 end ModuleA
```

A interface de um módulo é criada com o mesmo nome do módulo e seus elementos são obtidos a partir da interface do TAD. Os tipos são mapeados para a **algebra** e os serviços para **abstractions** como abstrações públicas. Estes elementos são também parcialmente declarados, como mostrado no código do módulo a seguir.

Módulo em Machĭna

```
1  module ModuleA
2    algebra:
3      // elementos da interface
4      typeT1;
5      typeT2;
6
7      // álgebra do TAD
8      x, y;
9      z;
10     w;
11
12
13
```

```
14      abstractions:
15        // elementos da interface
16        public action service1(a);
17        public action service2(a, b);
18        public action service3;
19
20
21      invariant:
22        // álgebra do TAD
23        x > y;      y > 0;      z < 0;
24
25    end ModuleA
```

Assim, cria-se uma correspondência entre as assinaturas da interface e suas definições no Glossário. A partir disso, pode-se definir a seguinte interface do módulo, ainda com elementos incompletos.

Glossário : Interface

```
1   interface ModuleA
2     typeT1;
3     typeT2;
4
5     action service1(a);
6     action service2(a, b);
7     action service3;
8   end ModuleA
```

O Tipo Abstrato de Dados é responsável por indicar o nome do módulo, as principais funções, os invariantes e a interface. Tanto a **algebra** quanto a interface do módulo criado apresentam elementos com definições pendentes. Isso significa conter apenas os nomes ou as assinaturas sem a definição de tipos, corpo ou classificação. Esses fragmentos serão posteriormente detalhados na regra de refinamento aplicada sobre o Glossário.

É utilizado o refinamento de dado com instanciação, onde são definidos os dados principais do módulo e descritas, na interface, as ações utilizadas externamente. Além disso, o invariante determina as restrições que a estrutura principal do módulo deve respeitar. A regra de transição é preservada e isso será mostrado posteriormente na Seção 3.5.3.

3.5.2 Diagrama de Transição de Estados

O refinamento de um Diagrama de Transição de Estados acrescenta elementos na **algebra** do módulo referente à sua estrutura interna de controle, define o bloco **initial state** e apresenta o tratamento para os estados finais. Além disso, as condições e ações presentes em suas transições são indicadas na **algebra** e **abstractions** para posteriores refinamentos.

As regras de refinamento que serão apresentadas a seguir fazem parte do refinamento de sub-rotinas, apresentado na Seção 2.2.2, onde um módulo do Modelo Básico é substituído por um de Machĭna com mais detalhes e novas características são apresentas, como propriedades e definições de tipos.

Estados e Estado Interno

O bloco **algebra** é acrescido de um tipo enumerado, *InternalState*, onde os possíveis valores são dados pelos nomes dos estados presentes no DTE. O estado inicial é refinado para o valor *initialState*, que é definido como valor padrão.

Estados

```
1  algebra:
2    type InternalState = enum {initialState,
3                               state1,
4                               state2, ... ,
5                               stateN} default initialState;
```

O estado interno do DTE é dado pela função dinâmica *ctl_state* do tipo *InternalState*. Essa é uma função para controle interno visível apenas no módulo onde foi declarada. A LMM não permite o uso explícito de *ctl_state*, logo, para obter o valor do estado interno, deve-se sempre utilizar a função derivada *current_state*, que reflete publicamente o valor de *ctl_state*. Se um módulo A incluir outro B, $B.current\_state$ será visível em A, apenas para consulta.

Estado Interno

```
1  algebra:
2    ctl_state : InternalState;
3    public derived current_state : InternalState := ctl_state;
```

As alterações internas são realizadas pelas ações que definem as transições do DTE, como será mostrado na Seção 3.5.3, e no bloco **initial state**. Toda atualização é realizada sob o controle do Método de Refinamento Machĭna. Assim, garante-se que

o estado interno terá apenas um dos possíveis valores de estado em um determinando instante, conforme a propriedade de DTE, que pode ser descrita conforme a seguir.

```
1 AG(ctl_state = {initialState, state1 ··· stateN})
```

Múltiplas atribuições à *ctl_ state* em uma mesma transição gera inconsistência e este fato é automaticamente percebido em MRM, por meio de um verificador de modelos, como será discutido na Seção 4.

Para consulta do estado interno por outros módulos, existe a função *current_ state*, que reflete o valor de *ctl_ state*. A cláusula **derived** garante que seu valor não pode ser alterado diretamente com atribuições, preservando o correto valor de *ctl_ state*. Dependendo de futuros refinamentos e verificadores de modelos utilizados, essa propriedade pode ser automaticamente garantida, como será mostrado na Seção 4. Porém, para garanti-la em qualquer circunstância, ela pode ser representada da seguinte forma.

```
1 AG(ctl_state = current_state)
```

As definições dos estados e estado interno fazem parte da estrutura de dados que será utilizada para completar o refinamento aplicado sobre o DTE, conforme mostrado nas seções seguintes.

Estado Inicial

O estado inicial consiste em preparar um agente para a sua execução. Não possui atividade, não pode ser destino de nenhum fluxo e não sofre a atuação do Controle Global. A presença de condições no fluxo de saída não é permitida, pois seriam avaliados termos ainda indefinidos e a máquina não seria corretamente inicializada. Por isso, deve existir apenas um fluxo de saída no estado inicial sem a presença de condições.

O estado inicial de um DTE define o bloco **initial state** de Machĭna, onde são feitas as inicializações de funções e atualização do estado interno *ctl_ state* para o estado de destino. A Figura 3.32 mostra um exemplo de um estado inicial com uma ação de preparação em uma transição para o estado i. O código correspondente, resultado do refinamento de sub-rotinas, é mostrado em seguida.

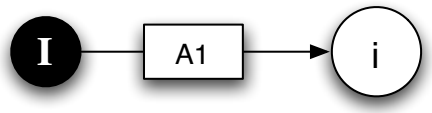


Figura 3.32: Estado inicial, preparação de um agente.

Estado Inicial

```
1  initial state:
2    A1;
3    ctl_state := i;
```

Estado Final

Os estados finais de um diagrama têm a funcionalidade de executar sua atividade, se existir uma, e finalizar a execução do agente, utilizando a função **stop**. Para o estado final *f*, representado na Figura 3.33, cria-se o seguinte código, que será posteriormente incluído na regra de transição do módulo. Testa-se o estado interno, se for o estado final, executa-se sua atividade, se existir, e executa a regra **stop** para finalizar a execução do agente.

Figura 3.33: Estado *f*, finaliza a execução de um agente.

Estado Final: trecho da regra de transição

```
1  transition:
2    if ctl_state = f then
3      atvState_f;
4      stop;
5    end
```

Os estados finais terminam a execução do agente utilizando a cláusula **stop**. Portanto, se um agente estiver uma vez em um estado final, implica que no estado seguinte a variável implícita do módulo, **state**, terá o valor *stopped*. Para todo estado final *f*, essa propriedade pode ser expressa como

```
1  AG((ctl_state = f) -> (next(state) = stopped))
```

Novamente é utilizado o refinamento de sub-rotinas definindo as operações que devem ser realizadas quando uma execução de um agente for finalizada. Com a transformação, surge a propriedade mencionada.

Condições e Ações

As condições e ações encontradas nos fluxos de transições representados no DTE são parcialmente declaradas em **algebra** e **abstractions** respectivamente, e posteriormente serão detalhadas no refinamento de Diagramas de Saída de Estado, Seção 3.5.3 e Glossário 3.5.4. Caso algum destes elementos já tenha sido declarado, nada é feito.

3.5.3 Diagrama de Saída de Estado

A partir de cada Diagrama de Saída de Estado, é possível obter a regra de transição de estado, definir pré e pós condições das ações do DTE, atividades e propriedade de estados. Cada um destes itens é apresentado a seguir. As regras de refinamento fazem parte do tipo refinamento de sub-rotinas, complementando o processo iniciado nas regras aplicadas sobre o Diagrama de Transição de Estados, mostrado na Seção 3.5.2.

Regra de Transição

Para cada fluxo de saída de um estado não final i, é gerada uma ação *flowFrom_ i_ n*, sendo n o índice do fluxo, com a pré condição *ctl_ state* $= i$ a ser respeitada. Neste momento, seu corpo ainda se encontra vazio e será posteriormente refinado. Por exemplo, a transição da Figura 3.34, página 100, é refinada no seguinte código, onde há apenas sua assinatura e pré-condição, ou seja, seu contrato.

Transição de Estado

```
1  abstractions:
2    action flowFrom_i_1 is
3      require: ctl_state = i;
4    end
```

Para cada Diagrama de Saída de Estado, é gerada uma ação *flowFrom_ i* que seu corpo é composto do conjunto de ações *flowFrom_ i_ n* a ser executado sob a condição *ctl_ state* $= i$. Sendo este o único local de chamada às abstrações *flowFrom_ i_ n*, para todo n, a pré condição destas ações serão sempre respeitadas. Assim, para um estado i com n fluxos de saídas, será gerado o seguinte código.

flowFrom_ i

```
1  abstractions:
2    action flowFrom_i is
3      if ctl_state = i then
4        flowFrom_i_1;
5        . . .
6        flowFrom_i_n;
7      end
8    end
```

A regra de transição, bloco **transition**, é construída chamando todas as ações de fluxo *flowFrom_ i*, para $1 \leq i \leq k$, onde k é o número de estados não finais, e operações de finalização correspondentes aos estados finais. Como *ctl_ state* possui apenas um valor, a regra de transição irá escolher corretamente a ação de finalização ou o conjunto de fluxos a ser avaliado de acordo com o estado corrente. Posteriormente, as ações *flowFrom_ i_ n* devem conter exclusão mútua das condições disparadoras para que o fluxo correto seja executado, essa propriedade será tratada mais adiante. Assim, o bloco **transition** é definido da seguinte forma, considerando k estados não finais e m estados finais .

Regra de Tranisção

```
1  transition:
2    flowFrom_1;
3    . . .
4    flowFrom_k;
5
6    if ctl_state = f1 then
7      atvState_f1;
8      stop;
9    end
10   . . .
11   if ctl_state = fm then
12     atvState_fm;
13     stop;
14   end
```

A regra de refinamento apresentada transforma as transições do Diagrama de Transição de Estados completo na regra de transição do módulo, onde são acrescidos detalhes que antes eram implícitos no DTE, como pré condições de *flowFrom_ i_ n* e verificação do estado interno em *flowFrom_ i*. Este é o refinamento de sub-rotinas, que completa o refinamento de dados descrito na Seção 3.5.1. A seguir, apresentam-se as

regras que completam as abstrações de transição (*flowFrom_ i_ n.*)

Transições

Uma transição, como mostrada na Figura 3.34, corresponde a uma *action* com o nome padrão de *flowFrom_ i_ n* e pré condição *ctl_ state* = *i* a ser respeitada, onde *i* é origem e *n* é o índex do fluxo, contabilizado de acordo com número de fluxos de saída do estado *i*. Tendo essa definição sido feita na regra de refinamento mostrada anteriormente, faz-se o detalhamento de seu corpo a partir dos elementos (condições e ações) da transição, representado na Figura 3.34 por *contT*, que serão obtidos com os refinamentos seguintes.

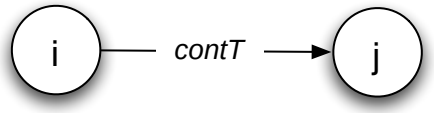


Figura 3.34: Transição de estado.

Transição de Estado pendente de *contT*

```
1  abstractions:
2    action flowFrom_i_1 is
3      require: ctl_state = i;
4      // contT
5    end
```

Uma transição pode ter o mesmo estado como origem e destino. Para o exemplo da Figura 3.35, obtém-se o código mostrado a seguir. Nota-se que a função *ctl_ state*, que representa o estado interno, é atualizada para o estado *i* na linha 7, que é tanto origem como destino. Por isso, este tipo de transição concorre com os demais fluxos de saída de um estado. Avaliação de condições ($C1$) será discutida mais adiante, assim como de ações ($A1$).

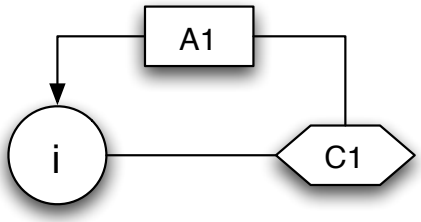


Figura 3.35: Transição com origem igual ao destino.

Origem Igual ao Destino

```
1  abstractions:
2    action flowFrom_i_1 is
3      require: ctl_state = i;
4
5      if C1 then
6        A1;
7        ctl_state := i;
8      end
9    end
```

Condições

As condições são responsáveis por controlar a passagem do fluxo em uma transição. Isso significa que, dependendo da avaliação da expressão, a transição pode executar ações diferentes ou mesmo não se completar. As condições são mapeadas para comandos *if-then-else* dentro de uma transição. Tomando como exemplo a Figura 3.36, pode-se ter o seguinte trecho de código. Avalia-se a condição C, caso verdadeira, o fluxo prossegue avaliando $contT$, caso contrário a transição é interrompida sem atualizar o estado interno. A atualização do estado interno não é mostrada pois depende de $contT$. Como esta é uma condição com um único fluxo de saída, a parte *else* é omitida.

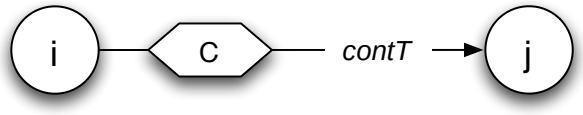


Figura 3.36: Condição com um fluxo de saída.

Condição *if-then*

```
1  abstractions:
2    action flowFrom_i_1 is
3      require: ctl_state = i;
4
5      if C then
6        // contT
7      end
8    end
```

Caso haja alternativas no fluxo, Figura 3.37, o código gerado corresponde ao mostrado a seguir, contendo a parte *else*. Caso a avaliação seja verdadeira, avalia-se *contT*, caso contrário, avalia-se *contF*.

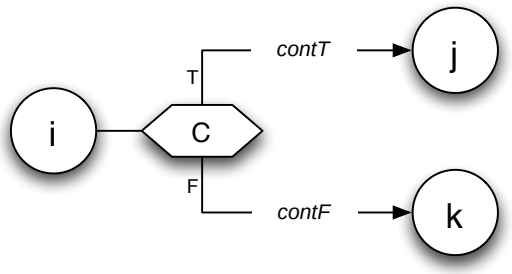


Figura 3.37: Condição com dois fluxos de saída.

Condição *if-then-else*

```
1  abstractions:
2    action flowFrom_i_1 is
3      require: ctl_state = i;
4
5
6      if C then
7        // contT
8      else
9        // contF
10     end
11   end
```

Em ambos os casos citados de condição, a atualização do estado interno do módulo depende das condições seguintes, *contT* ou *contF*. No caso de uma seqüência de condições em um mesmo fluxo, como mostra a Figura 3.38, o código correspondente seria um aninhamento de comandos *if-then-else*. A atualização do estado corrente é efetivada somente se o fluxo alcançar um estado de destino, ou seja, é realizada em um dos comandos *if-then-else* mais internos. O código correspondente à Figura 3.38 é apresentado a seguir.

Condições Aninhadas

```
1  action flowFrom_i_1 is
2    require: ctl_state = i;
3
4    if C1 then
5      A1;
6      if C2 then
7        A2;
8        ctl_state := j;
9      else
10       A3;
11       ctl_state := k;
12     end
13   end
14 end
```

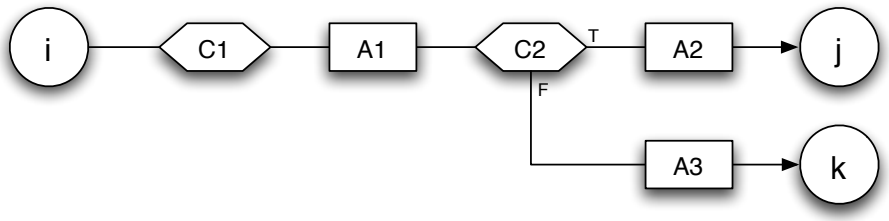


Figura 3.38: Seqüência de condições.

Transições com mesma origem e destinos diferentes podem causar inconsistência, de acordo com a propriedade de um diagrama estar somente em um único estado em um determinado instante. Isso pode ser mais bem esclarecido com o código a seguir, gerado a partir da Figura 3.8 da página 59, que por questões de legibilidade está reproduzida na Figura 3.39. A indexação dos fluxos é feita, neste caso, de cima para baixo.

Inconsistências em Condições

```
1  action flowFrom_i_1 is
2    require: ctl_state = i;
3
4    if C1 then
5      A1;
6      ctl_state := j1;
7    end
8  end
9
10
11
```

```
12  action flowFrom_i_2 is
13    require: ctl_state = i;
14
15    if C2 then
16      A2;
17      ctl_state := j2;
18    end
19
20  action flowFrom_i_3
21    require: ctl_state = i;
22
23    if C3 then
24      A3;
25      ctl_state := j2;
26    end
27  end
```

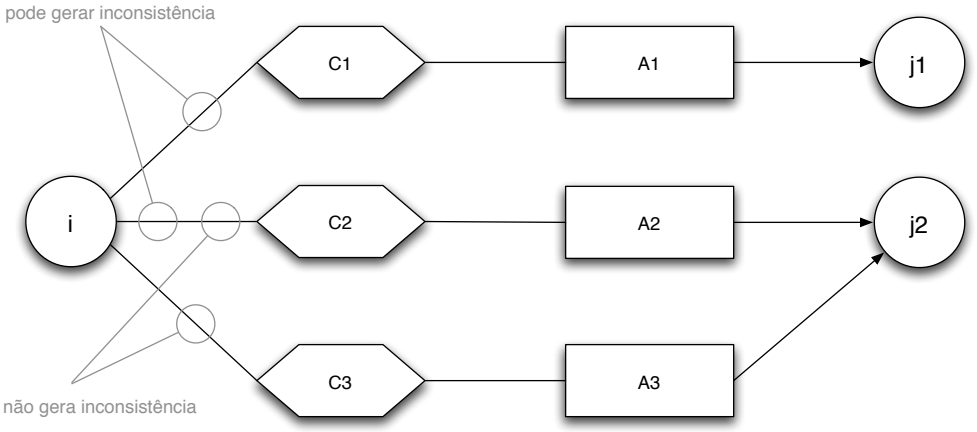


Figura 3.39: Situações de concorrência de fluxos.

As chamadas dessas ações são feitas no bloco **transition** de Machĭna e a execução depende do estado corrente. Como ambas possuem o mesmo estado de origem, sendo $C1$ verdadeira, se $C2$ ou $C3$ também forem, a atualização da linha 6 será inconsistente com a atualização 17 ou 25.

Pode-se perceber que as ações *flowFrom_i_2* e *flowFrom_i_3* podem conter as avaliações de $C2$ e $C3$ não disjuntas, pois as atualizações no estado interno nas linhas 17 e 25 associam o mesmo valor à *ctl_state*, não gerando inconsistência, de acordo com o modelo ASM [Gur95].

Caso exista um erro desta natureza, o método de verificação deve indicar onde se encontra o problema. Com isso, define-se a propriedade de exclusão mútua de condições da seguinte forma. Dado um DSE e considerando apenas as condições do

fluxo, identifica-se todos os caminhos possíveis $i \rightarrow jm$. A definição destes caminhos é feita pela conjunção cn das condições pertencentes ao fluxo. Caso não exista uma condição, assume-se cn como sendo verdadeiro. Isso pode ser representado pela tupla

$$(i,\ cn,\ jm)$$

Todas as tuplas que possuem mesmo estado de origem e de destino são agrupadas realizando a disjunção de suas condições. Por exemplo, as tuplas

$$(i,\ c2,\ j2)$$

$$(i,\ c3,\ j2)$$

são agrupadas como

$$(i,\ (c2\ \vee\ c3),\ j2)$$

Então, dado um estado i, as condições que levam aos diferentes estados seguintes devem ser mutuamente exclusivas e pode-se representar essa propriedade da seguinte forma

Exclusão Mútua

```
1   AG((ctl_state = i) -> (c1 xor (c2 ∨ c3) xor ··· cn ))
```

Assim, é possível verificar inconsistências na atualização do estado interno de um agente, e definindo a propriedade desta forma, torna-se possível identificar o momento em que o problema ocorre em um verificador de modelos, onde pode-se apresentar um exemplo de execução para que seja realizada a análise.

Com esta regra, as ações de transição de estado têm seu corpo definido de acordo com sua representação no Modelo Básico. As propriedades extraídas indicam as novas características que devem ser observadas. As chamadas às abstrações pertencentes ao fluxo são declaradas e ainda devem ser refinadas posteriormente.

Pré e pós condições de ações

No DSE, as ações podem estar associadas a pré e pós condições. Para toda ação presente em todos os DSEs, sua definição parcial em **abstractions** é acrescida da cláusula **require** e **ensure** de acordo com as definições no Modelo Básico. Assim, mais detalhes são inseridos na definição das ações, conforme previsto no tipo refinamento

de sub-rotinas. Por exemplo, a ação mostrada na Figura 3.40 é refinada para o código mostrado a seguir.

Figura 3.40: Pré e pós condições de ação.

Pré e pós condições de ação

```
1  abstractions:
2    action A is
3      require : RA;
4      ensure  : EA;
5    end
```

Atividade de estado

Uma atividade descreve a ação que será realizada durante a permanência de um agente em um estado. Ela é independente dos fluxos de saída, então sua chamada deve estar fora de qualquer comando condicional, como *if-then-else*. Não é possível atualizar o estado interno na atividade, portanto não existe inconsistências com os fluxos de saída. Seu efeito só é percebido na iteração seguinte, assim como qualquer outra ação do modelo. Tendo como exemplo a Figura 3.41, o código em Machĭna é representado a seguir. As ações são formalizadas na Seção 3.5.4.

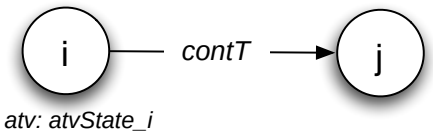


Figura 3.41: Atividade de um estado.

Atividade de um Estado

```
1  abstractions:
2    action flowFrom_i is
3      if ctl_state = i then
4        atvState_i;
5        // actions of contT
6      end
```

Esta regra faz parte do tipo refinamento de sub-rotinas e quando existir uma atividade associada a um estado *i*, a ação que a representa, se ainda não existir, é declarada parcialmente em **abstractions** e sua chamada é colocada como primeira regra da abstração *flowFrom_i*, logo após a verificação do valor de *ctl_state*. Isso garante que a ação de atividade do estado *i* será executada quando *ctl_state* for igual a *i* e antes de executar qualquer fluxo de saída ou operação de término da regra de transição do agente.

Propriedade de estado

A propriedade de estado *prop_i* representa uma condição que deve ser satisfeita quando o agente em execução permanecer em um determinado estado *i*, Figura 3.42. O refinamento de sub-rotina transforma a representação de propriedade de estado no Modelo Básico em uma propriedade CTL, da seguinte forma, para que seja possível realizar a verificação adequadamente.

Propriedade de estado

```
1  AG(ctl_state = i -> prop_i)
```

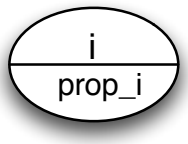


Figura 3.42: Propriedade de estado.

Dessa forma, pode-se verificar a execução do sistema de acordo com sua especificação, e no caso de falha, pode-se mostrar um exemplo de quando o erro ocorre.

Controle Global

O Controle Global atua sobre o fluxo de saída de todos os estados, exceto o estado inicial. Ele é opcional e, se existir, deve ser único para cada módulo. A representação em Machĭna dos diagramas reduzidos segue as mesmas regras do DTE principal, com exceção de que o estado de origem não é testado e de que o diagrama reduzido que não desvia o fluxo principal não atualiza o estado interno *ctl_state*.

Uma ação especial, denominada *GlobalControl*, é gerada para representar o comportamento do Controle Global. Um módulo que possui um Controle Global tem sua regra de transição, bloco **transition** de Machĭna, realizada em dois passos. O primeiro

chama a ação *GlobalControl* e o segundo consiste em executar o fluxo principal do DTE, como mostrado anteriormente.

Para exemplificar o Controle Global, utiliza-se o exemplo da Figura 3.14 na página 62, reproduzido na Figura 3.43 por questões de legibilidade. O código para a ação *GlobalControl* é mostrado a seguir.

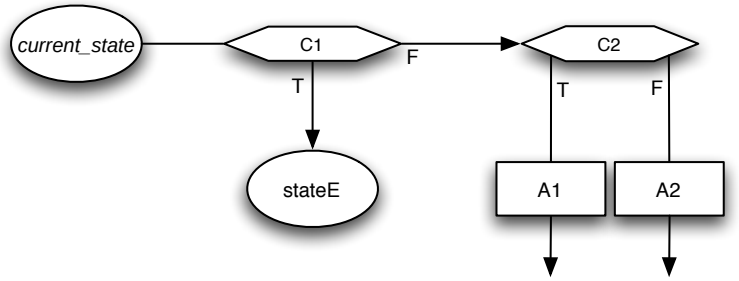


Figura 3.43: Controle Global.

Ação para Controle Global

```
1  abstractions:
2    action GlobalControl is
3      if C1 then
4        ctl_state := stateE;
5      else
6        if C2 then
7          A1;
8        else
9          A2;
10       end
11     end
12   end
```

Percebe-se que o estado corrente não é verificado, já que o Controle Global atua sobre todos os estados possíveis. A ação *GlobalControl*, primeiramente, verifica a condição $C1$ e, caso verdadeira, o fluxo principal é desviado para *stateE*, por isso a função *ctl_state* é atualizada na linha 4. Caso $C1$ seja falsa, avalia-se $C2$ para determinar qual ação será executada, $A1$ ou $A2$, e o fluxo é devolvido para o diagrama principal. Por isso não existe atualização do estado interno no bloco do comando *if-then* da linha 6 nem seu respectivo *else*, na linha 8;

Dada a regra de transição de um módulo descrita pelo DTE principal e o Controle Global da Figura 3.43, pode-se gerar o seguinte trecho de código para a regra de

transição, considerando *contFlow* como sendo todas as ações dos fluxos possíveis no diagrama.

Regra de Transição

```
1  transition:
2    step 1:
3      GlobalControl;
4    step 2:
5      // contFlow
6      next := 1;
```

Como o bloco **transition** é realizado em dois passos, as atualizações feitas no Controle Global são vistas nas ações de transição representadas por *contFlow*. Dessa forma, um desvio de fluxo é percebido pela atualização da função *ctl_state*, assim como as atualizações realizadas pelas ações $A1$ e $A2$. Para evitar a finalização inesperada do agente, o comando *next := 1* na linha 6 é realizado para que, na próxima iteração, o Controle Global seja executado novamente. Isso faz com que o primeiro passo, do Controle Global, seja novamente executado, preservando a semântica definida para o Diagrama de Transição de Estados.

Este refinamento é baseado no tipo de refinamento extensão conservativa, onde um novo comportamento, sob determinadas condições, é adicionado à execução normal. Em LMM, isso pode ser feito sem ter que alterar o comportamento já construído no DTE, contribuindo para melhor reaproveitamento do módulo. A regra de transição é alterada para preservar a semântica do Controle Global e suportar execução em dois passos.

Com as descrições apresentadas nesta seção, definem-se a função de estado interno com seus possíveis valores, a atividade de um estado e as transições de estado com as condições e ações. Também são definidos os comportamentos dos estados inicial e finais, assim como a ação para representar o Controle Global.

A estrutura de um módulo e o comportamento de seus agentes podem ser definidos pela descrição do Tipo Abstrato de Dados e pela regra de refinamento que atua sobre o DTE, mostrada nesta seção. Os blocos **initial** e **transition** já se apresentam completos neste momento, e os blocos **algebra** e **abstractions** parcialmente construídos. O Glossário e Mecanismo de Visibilidade completam estes blocos e definem os outros existentes em um módulo.

3.5.4 Glossário

O Glossário completa as definições dos blocos **algebra** e **abstractions**. Cada parte do Glossário é inserido em seu bloco correspondente na estrutura de Machĭna, de forma que a Álgebra completa **algebra** e Abstrações completa **abstractions**.

Tomando o exemplo de TAD da Figura 3.31 dado na Seção 3.5.1, página 93, supõem-se um DTE e o Controle Global, omitidos por questões de simplicidade, define-se um possível Glossário da seguinte forma. Podem existir outros elementos, não descritos no TAD, mas utilizados de alguma forma pelo módulo. Assim, estes também devem ser definidos na Álgebra, como, por exemplo, m, n e p na linha 11.

Glossário : Álgebra

```
1  // elementos da interface
2  public type typeT1 : Int;
3  public type typeT2 : Int;
4
5  // álgebra do TAD
6  x, y      : typeT1;
7  public z  : typeT2;
8  public w  : typeT2;
9
10 // elementos internos
11 m, n, p   : Int;
```

As abstrações disponibilizadas como serviços no TAD e as utilizadas no DTE devem ser definidas, assim como as demais que possam ser úteis dentro de outras ações, como mostra o exemplo a seguir.

Glossário : Abstrações

```
1  // elementos da interface
2  action service1(a : Int) is
3    . . . .
4  end
5
6  action service2(a : Int, b : Int) is
7    . . . .
8  end
9
10 action service3 is
11   incXY;
12   . . . .
13 end
```

```
14
15  // elementos internos
16  action incXY is
17    x := x + 2;
18    y := y + 1;
19  end
```

Essas definições completam os bloco **algebra** e **abstractions** anteriormente mostrados. Então, com a definição do Glossário apresentada e omitindo o DTE e o diagrama de Controle Global, um suposto código em Machĭna gerado seria:

Código Machĭna para *ModuleA*

```
1   module ModuleA
2     algebra:
3       // estrutura interna DTE
4       type InternalState is enum {initialState,
5                                   state1, . . .
6                                   stateN} default initialState;
7       ctl_state is InternalState;
8       public derived current_state : InternalState := clt_state;
9
10      // elementos da interface
11      public type typeT1 : Int;
12      public type typeT2 : Int;
13
14      // álgebra do TAD
15      x, y      : typeT1;
16      public z : typeT2;
17      public w : typeT2;
18
19      // elementos internos
20      m, n, p   : Int;
21
22    abstractions:
23      // controle global
24      action GlobalControl is
25        . . .
26      end
27
28      // transições de DSEs
29      action flowFrom_state1_1 is
30        require : ctl_state = state1;
31        . . .
32      end
```

```
33        . . .
34        // transições de um estado do DSEs
35        action flowFrom_state1 is
36          if ctl_state = state1 then
37            incXY;  // atividade de state1
38            . . .
39          end
40        end
41        . . .
42        // elementos da interface
43        public action service1(a : Int) is
44          . . . .
45        end
46
47        public action service2(a : Int, b : Int) is
48          . . . .
49        end
50
51        public action service3 is
52          incXY;
53          . . . .
54        end
55
56        // elementos internos
57        action incXY is
58          x := x + 2;
59          y := y + 1;
60        end
61
62      transition:
63        step 1:
64          GlobalControl;
65        step 2:
66          flowFrom_state1;
67          . . .
68          if ctl_state = final1 then
69            stop;
70          end
71          . . .
72          next := 1;
73
74      invariant:
75        // álgebra do TAD
76        x > y;      y > 0;      z < 0;
77    end ModuleA
```

Na definição do módulo, os primeiros elementos na álgebra foram obtidos a partir da estrutura interna do Diagrama de Transição de Estados, sendo a definição do tipo *InternalState*, nas linhas de 4 a 6, a função de estado interno *ctl_state* na linha 7 e a função derivada *current_state* na linha 8. As definições da linha 15 até 17 foram obtidas do detalhamento das declarações do TAD, e da linha 20 na Álgebra no Glossário.

No bloco de abstrações, na linha 24, define-se a ação para o Controle Global e a linha 29 corresponde ao início das declarações das ações de transições obtidas automaticamente a partir do Diagrama de Transição de Estados, assim como na linha 35, dá-se início às declarações correspondentes às ações da regra de transição. Da linha 43 até 60 encontram-se as ações que foram definidas no Glossário, na parte Abstrações, e o bloco de **invariant**, linha 74, foi obtido do TAD, como mostrado anteriormente.

A regra de transição, **transition** na linha 62, é gerada em dois passos, sendo o primeiro passo a execução do Controle Global, linha 64, e o segundo consiste nas chamadas das ações de transição de estado, que inicia na linha 66, e tratamento de estados finais, na linha 68. Para impedir a terminação não esperada da regra de transição do agente, faz-se na linha 72 o passo seguinte ser o primeiro novamente. Por fim, os invariantes são definidos na linha 74 a partir da declaração do TAD.

O Glossário complementa a definição de ações e funções presentes no DTE completo e no código Machĭna gerado até o momento. Caso existam ações com mais de uma definição para pré e pós condições, faz-se a conjunção das expressões booleanas, seja presente nos Diagrama de Saída de Estados ou na definição do Glossário. A interface possui uma correspondência de seus elementos com suas definições no Glossário. Sendo assim, pode-se completar a interface de *ModuleA* da seguinte forma.

Interface de *ModuleA*

```
1  interface ModuleA
2    public type typeT1 : Int;
3    public type typeT2 : Int;
4
5    public action service1(a : Int);
6    public action service2(a : Int, b : Int);
7    public action service3;
8
9  end ModuleA
```

Dessa forma, o Tipo Abstrato de Dados, o Diagrama de Transição de Estados e o Glossário definem a estrutura e o comportamento de um módulo em Machĭna tal como sua interface para prover serviços a agentes de outros módulos. Existem as construções básicas do modelo, como o estado interno e seu tipo, e as definidas pelo usuário, como

transições e elementos do TAD e Glossário. O Controle Global deve ser realizado antes de qualquer transição de estado, e para ter validade deve ser realizado no primeiro passo da regra de transição, deixando a execução normal no segundo passo. Esta é a penúltima regra do tipo refinamento de sub-rotinas.

3.5.5 Mecanismo de Visibilidade

O mecanismo de visibilidade determina o relacionamento entre dois módulos e a interação entre agentes de acordo com o diagrama TAD mostrado na Seção 3.1.5. O efeito é definir os blocos **import** e **include** de um módulo Machĭna. Por exemplo, dado o diagrama da Figura 3.44, têm-se os seguintes blocos para *ModuleA*.

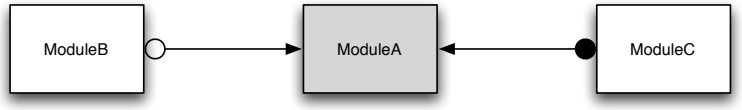


Figura 3.44: Diagrama TAD.

Definição de **include** e **import**

```
1  module ModuleA
2    import:  ModuleB;
3    include: ModuleC;
4    . . . . .
5  end ModuleA
```

No diagrama, o TAD que representa *ModuleA* está marcado como sendo a máquina do sistema. Isso significa criar uma máquina com um único agente de *ModuleA* e disparar sua regra de transição da seguinte forma:

Criação da máquina do sistema

```
1  machine ModuleA
2    agent of ModuleA;
3  end ModuleA
```

O exemplo mostra *ModuleA* importando os serviços de *ModuleB* e incorporando os elementos públicos de *ModuleC*. Além disso, cria-se a máquina que representa o sistema do Modelo Básico. Dessa forma, completa-se a especificação de um módulo definindo a relação entre módulos e interação entre agentes no sistema.

3.6 Conclusões

O Método de Refinamento Machĭna define a Linguagem de Modelagem Machĭna (LMM) para definição em alto nível do Modelo Básico. LMM é baseada em diagramas de fluxo de controle e os diferentes níveis de abstração do Modelo Básico ficam bem separados em seus componentes, contribuindo para obter uma linguagem intuitiva, maior poder de expressão e maior legibilidade nos domínios da Aplicação e dos Modelos.

A LMM possui Diagrama de Transição de Estados (DTE) e Diagramas de Saída de Estado (DSEs) como recursos gráficos. O Glossário define os elementos destes diagramas e permite expressar formalmente propriedades pertencentes ao Domínio da Aplicação. O alto nível de abstração presente em LMM permite validar a especificação sob a supervisão de clientes e desenvolvedores.

As regras definidas em MRM são baseadas nos tipos de refinamento identificados no Método de Refinamento ASM e geram uma especificação em Machĭna como código alvo. Herdando as características de Machĭna, LMM permite definir sistemas multiagentes com execuções síncronas e assíncronas.

Com essas características, MRM permite realizar facilmente descrições de sistemas de pequeno e grande porte com grande quantidade de informações, contribuindo para uma especificação completa, precisa e que captura as características mais relevantes do problema a ser solucionado.

Capítulo 4

Verificação Automática de Propriedades

O Método de Refinamento Machĭna suporta a construção do Modelo Básico e sua transformação para a linguagem Machĭna, conforme mostrado nas Seções 3.1 e 3.5. Porém, ainda é necessário verificar se as propriedades do sistema estão sendo respeitadas no modelo, ou seja, é preciso garantir que a implementação esteja de acordo com a especificação do problema. Em MRM, propõe-se a verificação automática e transparente, o que significa não contar com a participação humana nem exigir o pleno conhecimento das técnicas de verificação utilizadas.

Existem diferentes maneiras de se provar que as propriedades descritas no Modelo Básico são preservadas ao longo do refinamento. O Modelo Básico, por exemplo, poderia ser transformado em uma linguagem de entrada de um determinado provador de teoremas onde seria automaticamente verificado. O trabalho de Freek Wiedijk, *The Seventeen Provers of the World* [Wie06], relata e compara os 17 principais provadores de teorema existentes mostrando um exemplo de execução em cada um deles. Dentre os apresentados, estão:

HOL	Mizar	PVS	Coq
Otter/Ivy	Isabelle/Isar	Alfa/Agda	ACL2
PhoX	IMPS	Metamath	Theorema
Lego	Nuprl	Ω mega	B method
Minlog			

Esta lista contém 17 mecanismos de prova de teoremas e foram selecionados sob os critérios de serem desenvolvidos para formalização matemática, e possuírem uma especialidade, ou seja, cada um é melhor sob algum ponto de vista e são considerados líderes na sua área. Maiores detalhes sobre esta classificação encontram-se em [Wie06].

O exemplo utilizado para mostrar e comparar as ferramentas foi o problema da prova de que a raiz quadrada de dois é um número irracional.

De acordo com Börger em [BS03], as principais ferramentas utilizadas na verificação automática de modelos ASM são KIV [Rei92, Rei95, BRS+00a], PVS [ORSSC98, ORS92] e Isabelle [PN06], além de verificadores de modelos (*Model Checkers*), que serão descritos a seguir. Dentre estes provadores citados, apenas o KIV não apareceu na relação de [Wie06].

Estes provadores de teoremas são ferramentas poderosas que auxiliam a verificação, porém é necessário informar as estratégias em determinados pontos, exigindo conhecimento do modelo e da linguagem do provador. Embora os provadores possibilitem abordar uma grande variedade de problemas, essa característica impede a verificação completamente automática e transparente dentro do Método de Refinamento Machĭna.

Verificação de Modelos (*Model Checking*) é um método de verificação formal de sistemas onde é possível obter representações mais abstratas que provadores de teorema e são mais indicados para modelagem de controle de fluxo [CGP00]. Embora esse método não consiga abordar a mesma variedade de problemas que os provadores conseguem, é possível tratar, segundo Clarke, quase todos os problemas que são realmente desenvolvidos na prática [CGP00]. Além disso, verificadores de modelos, no caso de falha na verificação, podem fornecer exemplos de transições que levaram ao erro, possibilitando a depuração do sistema. A seguir são apresentados os trabalhos relacionados com *Model Checking* e seu uso em ASM.

4.1 Verificação de Modelos (*Model Checking*)

Verificação de Modelos (*Model Checking* [CGP00]) é um método que permite verificar automaticamente propriedades de um sistema de transição de estado. Um estado é dado pela situação de todas as variáveis do modelo em um determinado instante. Este método enumera todos os estados alcançáveis, dados os estados iniciais e a regra de transição, e verifica as propriedades de acordo com as especificações dadas. Isso fatalmente gera uma explosão de estados a serem verificados, porém a capacidade de lidar com tal fenômeno é uma característica notável de verificação de modelos.

Um verificador de modelos (*Model Checker*) implementa um método de verificação de modelos e é possível encontrar vários deles na literatura [RGA+96, CZS+97a, FSS+94, HHK96, Hol97, KNP02, Ltd97]. Neste trabalho foi utilizado um verificador baseado no *Symbolic Model Verifier* (SMV) [K.L92], que será mostrado a seguir. A descrição de outros verificadores e suas classificações (como PRISM [KNP02], VIS [RGA+96], SVE [FSS+94], MDG [CZS+97a], SPIN [Hol97], COSPAN [HHK96] e FDR

[Ltd97]) fogem do escopo deste trabalho.

Um verificador de modelos recebe como entrada uma especificação de um sistema de transição de estados, onde são definidas suas variáveis, como cada uma é inicializada e como são atualizadas ao longo das transições de estados. A representação do sistema é dada por meio da estrutura de *Kripke* [CGP00, Capítulo 2], definida pela tupla (S, S_0, R, L) onde:

- S é o conjunto de estados;
- $S_0 \subseteq S$ é o conjunto de estados iniciais, freqüentemente omitido por questões de simplicidade;
- $R \subseteq S \times S$ é a relação de transição;
- $L : S \rightarrow 2^{AP}$ é a função que rotula cada estado com um conjunto de propriedades verdadeiras, onde AP (*Atomic Prepositions*) são as variáveis de estado do sistema.

Internamente em um verificador de modelos, a estrutura de *Kripke* é representada por diagramas binários de decisão ordenados (OBDDS - *Ordered Binary Decision Diagrams*) [CGP00, Capítulo 5], que são derivados de árvores binárias de decisão. Uma árvore binária de decisão é uma árvore dirigida para representar funções booleanas. Cada nodo interno representa uma variável v do sistema, dada pela função *var(v)*. Cada nodo interno possui duas arestas, uma que representa a atribuição do valor 0 à v, chamada de *low(v)*, e a outra representa a atribuição de 1, chamada de *high(v)*. A aresta *low(v)* pode ser representada por uma aresta pontilhada. Os nodos externos, terminais, são representados pelos valores 0 ou 1, obtidos por *value(v)*. Cada caminho na árvore representa uma atribuição a cada *var(v)*, e o resultado é dado pelo valor do vértice terminal, *value(v)* = 0 ou 1.

Árvores binárias de decisão não possuem uma representação concisa de uma fórmula booleana. Elas possuem o mesmo tamanho que uma tabela verdade. Porém, existem certas características de redundâncias que podem ser exploradas a fim de se obter melhores representações, como por exemplo junção de sub-árvores idênticas.

Ao juntar sub-árvores iguais, obtém-se um grafo orientado acíclico (*Direct Acyclic Graph* - DAG) com uma raiz única, caracterizando-se em um BDD (*Binary Decision Diagram*). Um BDD é dito ordenado, OBDD (*Ordered Binary Decision Diagram*), se todos os caminhos a partir da raiz respeitam a mesma ordem das variáveis. Um OBDD é dito reduzido, ROBDD (*Reduced Ordered Binary Decision Diagram*) sob as seguintes condições:

- quando não existem dois nodos distintos u e v que representem a mesma variável e possuam os mesmos filhos *low* e *high*, ou seja

$$(var(v) = var(u)\ \&\ low(v) = low(u)\ \&\ high(v) = high(u)) \rightarrow\ v \neq\ u$$

- uma variável v não contém os mesmos filhos, ou seja

$$low(v)\ \neq\ high(v)$$

Considerando três variáveis, *x1*, *x2* e *x3*, pode-se construir a árvore binária de decisão mostrada na Figura 4.1(a) de acordo com uma função *f(x1,x2,x3)*, na qual tem a tabela verdade representada na Figura 4.1(b). Pode-se notar que todos os possíveis caminhos sempre terão a mesma ordem de variáveis. Em seguida, Figura 4.1(c) mostra o ROBDD para esta mesma função. Como pode ser visto, é um diagrama com menos estados e somente dois nodos terminais, representados pelas caixas contendo 0 ou 1.

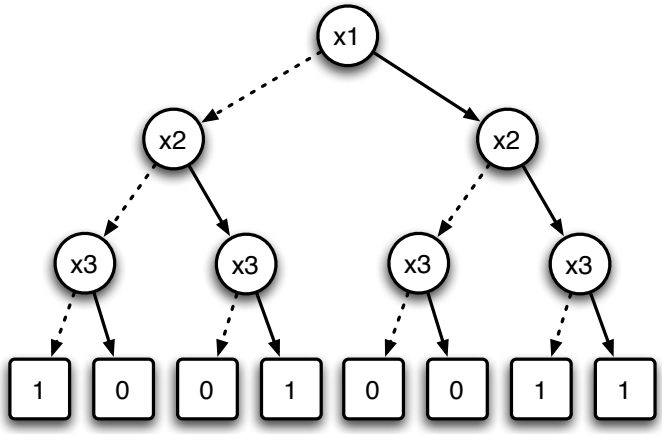
Considerando uma estrutura *Kripke* $M = (S, R, L)$, para representá-la em um OBDD é necessário descrever o conjunto de estados S e a relação R. Por questões práticas, o rótulo L foi omitido. Dado que v é uma variável booleana pertencente ao conjunto de variáveis do sistema V e $\overline{v}$ é um vetor de variáveis v cada uma com seu valor, um estado pode ser descrito por uma representação de $\overline{v}$, que aplicada a uma função booleana f obtém-se o valor 1. Por exemplo, para o exemplo da Figura 4.1(c), existiriam os três estados $(\neg\, x1, \neg\, x2, \neg\, x3), (\neg\, x1, x2, x3)$ e $(x1, x2)$, obtidos a partir dos caminhos do OBDD que resultam no nodo terminal 1.

Para a representação de R utiliza-se a mesma codificação de S, considerando v' uma variável pertencente ao conjunto V', que significa o valor de v no estado seguinte, e $\overline{v}'$ um vetor de v'. Então, a relação R é definida como $(\overline{v}, \overline{v}')$, onde um vetor de variáveis, cada uma com determinado valor, é mapeado em outro vetor com seus respectivos valores no estado seguinte. Assim, pode-se representar uma relação de transição por uma conjunção dos valroes atuais com os valores do próximo estado. Supondo um conjunto de transições para o exemplo da Figura 4.1(c), pode-se descrevê-lo em fórmulas lógicas conforme a seguir. A Figura 4.2 mostra a representação gráfica destas transições. A disjunção de todas as transições é a representação de relação de transição R completa.

$$(\neg\, x1\ \wedge\ \neg\, x2\ \wedge\ \neg\, x3\ \wedge\ \neg\, x1'\ \wedge\ x2'\ \wedge\ x3')$$

$$(\neg\, x1\ \wedge\ \neg\, x2\ \wedge\ \neg\, x3\ \wedge\ x1'\ \wedge\ x2')$$

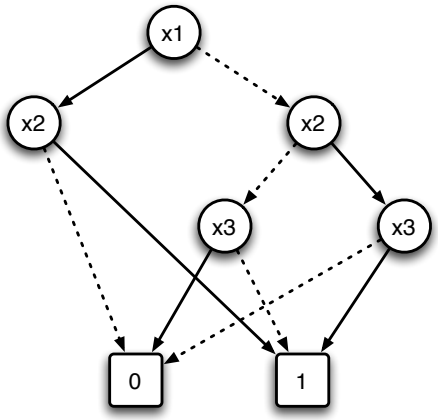
$$(\neg\, x1\ \wedge\ x2\ \wedge\ x3\ \wedge\ \neg\, x1'\ \wedge\ x2'\ \wedge\ x3')$$



(a) Árvore binária de decisão.

x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(b) Tabela verdade para função *f*.



(c) ROBDD

Figura 4.1: Diagrama Binário de Decisão.

$$(x1 \ \wedge \ x2 \ \wedge \ \neg \, x1' \ \wedge \ \neg \, x2' \ \wedge \ \neg \, x3')$$

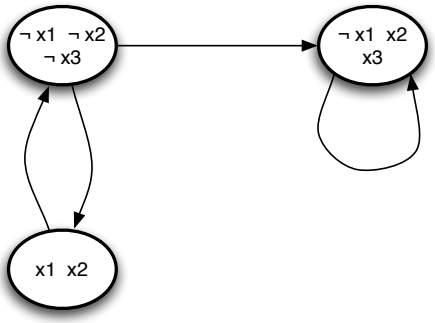


Figura 4.2: Representação de uma estrutura *Kripke* em OBDD.

A verificação é realizada não pelo acompanhamento dos estados, mas sim por perguntas que são feitas ao modelo, as quais são descritas normalmente em fórmulas tem-

porais CTL e LTL. Tais fórmulas expressam as propriedades que devem ser garantidas pelo sistema e são descritas em função da ordem temporal dos estados ou caminhos para se chegar a determinados estados.

A análise dos estados, realizada pelas perguntas feitas ao sistema, proporciona uma verificação de forma simples, sem a necessidade de conferir cada estado existente. É possível expressar diversas formas de propriedades e um verificador de modelos responde verdadeiro ou falso, e no caso de ser falso, é possível que seja mostrado um exemplo de tal situação. Este é um importante recurso para depuração do sistema modelado.

4.1.1 *Symbolic Model Verifier* (SMV)

Symbolic Model Verifier (SMV) [K.L92, McM93] é um verificador de modelos com linguagem própria para especificação de sistemas de transição de estados onde os estados são finitos, ou máquina de estados finitos (*Finite State Machine*). O SMV utiliza algoritmos simbólicos (*Symbolic Model Checking* [ISEJ05]) baseados em OBDD (*Ordered Binary Decision Diagram*), um melhoramento de BDD (*Binary Decision Diagram*) [CGP00, Capítulo 5]. Com isso, é possível verificar sistemas com grande número de estados, e.g., na ordem de 10^{120}, como demonstrado nos exemplos [JED91, JED$^+$94].

Uma execução de SMV verifica cada propriedade, definida em CTL (*Computational Tree Logic*) ou LTL (*Logical Temporal Logic*), utilizando algoritmos sobre o ROBDD. É retornado um resultado verdadeiro ou falso para cada uma delas. No caso de um resultado falso, e sempre que possível, é dado um contra-exemplo onde mostra-se uma execução, a partir do estado inicial, que viola a propriedade. Este artifício é muito útil para depuração do sistema submetido à verificação.

O SMV foi inicialmente desenvolvido na *Carnegie Mellon University* (CMU), e ficou conhecido como CMU SMV. Novas propostas para SMV foram feitas com o objetivo de obter maior desempenho e prover funcionalidades mais poderosas. Destacam-se os trabalhos Cadence SMV [McM99] e NuSMV [CCO$^+$06]. Ambos são reimplementações do CMU SMV, Cadence SMV foi desenvolvido pelo Doutor K. L. MacMillan no *Cadence Berkeley Labs*. NuSMV foi desenvolvido no ITC-IRST. Estes dois trabalhos apresentam melhorias consideráveis sobre o CMU SMV e são muito semelhantes, embora apresentem sintaxe diferente. Para o Método de Refinamento Machĭna, escolheu-se trabalhar com NuSMV pela similaridade com a sintaxe de CMU SMV e facilidade em obter manual de consultas e exemplos. Na Seção 4.1.2, são apresentadas as principais características de NuSMV.

4.1.2 NuSMV

O verificador de modelos NuSMV [CCO+06, CCO+05b, CCO+05a] é resultado da reimplementação e extensão do CMU SMV [McM93]. NuSMV faz parte de um projeto estruturado e bem documentado com o objetivo de ser amplamente utilizado em escalas industriais. Por isso, sua versão atual, NuSMV 2.4, é distribuída sob licença *OpenSource*, que permite qualquer interessado livremente utilizar e participar do desenvolvimento.

NuSMV permite representar sistemas de estados finitos síncronos e assíncronos, e especificar propriedades utilizando *Computation Tree Logic* (CTL) e *Linear Temporal Logic* (LTL) [CGP00, Capítulo 3]. A seguir, é dada uma breve descrição de como se especifica um sistema utilizando NuSMV. Como exemplo, será mostrada a especificação de um contador binário de três *bits*. Cada *bit* é modelado por uma célula de contagem e o sistema tem como objetivo somar constantemente o valor 1 ao valor anteriormente obtido.

MODULE

Módulo é um encapsulamento de declarações de variáveis de estado. Cada módulo pode conter a regra de transição de estados e especificações de propriedades, estes e outros elementos serão explicados a seguir. É possível declarar um módulo com parâmetros, no qual a passagem é por referência. Por exemplo, cria-se a seguir o módulo para uma célula de um contador binário que recebe como parâmetro o "vai-um" da soma.

Contador Binário

```
1  MODULE counter_cell(carry_in)
```

VAR

As declarações de variáveis de estado são realizadas no bloco VAR. Podem ser do tipo enumerado, intervalo, booleano e instâncias de outros módulos. No exemplo do contador binário declara-se a variável *value* como booleana.

Contador Binário

```
1  MODULE counter_cell(carry_in)
2    VAR
3      value :  boolean;
```

ASSIGN

A regra que determina a inicialização e transições de uma variável de estado são definidas por atribuições no bloco ASSIGN. A atribuição direta estabelece o valor da variável, *init(variable)* indica qual o valor inicial da variável de estado, e *next(variable)* fornece o valor no estado seguinte a partir do estado corrente. A atribuição em *next(variable)* pode ser feita utilizando a regra *case*, onde é possível determinar o próximo valor de *variable* em função de uma determinada condição. Assim, *next(variable)* pode facilmente determinar a regra de transição sob a perspectiva de uma variável. A seguir, definem-se o valor inicial e a regra de transição de *value* no exemplo do contador binário.

Contador Binário

```
1  MODULE counter_cell(carry_in)
2    VAR
3      value :  boolean;
4
5    ASSIGN
6      init(value) := 0;
7      next(value) := value + carry_in mod 2;
```

O valor inicial para *value* é 0, e a transição é feita calculando *value + carry_in*, onde *carry_in* é obtido pela soma de dígitos de ordem menor (menos significativos). Faz-se *mod 2* da soma para evitar acesso a um valor inválido, por exemplo 2, que não é aceito em uma variável booleana.

Pode-se utilizar a regra *case* para definir *next(variable)*, conforme o código a seguir, supondo um módulo com uma variável *ctl_state* enumerada com os valores *state1*, *state2* ou *state3*.

Exemplo de Case

```
1  MODULE case_example
2    VAR  ctl_state :  {state1, state2, state3};
3
4    ASSIGN
5      init(ctl_state) := state1;
6      next(ctl_state) :=
7        case
8          ctl_state = state1 :  state2
9          ctl_state = state2 :  state3
10         ctl_state = state3 :  state1
11         1 :  state1
12       esac;
```

A atribuição ao próximo valor de *ctl_state* é condicionada ao seu valor corrente. Se for *state1*, o próximo valor será *state2*. Se for *state2*, será *state3* e se for *state3*, volta-se ao valor *state1*. A avaliação é feita na ordem que as cláusulas aparecem e a atribuição é feita quando se encontra a primeira cláusula verdadeira, independente das seguintes.

Caso não seja indicado o próximo valor de uma variável ou nenhuma das cláusulas de uma regra *case* for verdadeira, atribui-se um valor não determinístico dentre os possíveis valores que a variável pode assumir. Em NuSMV, isso significa que existirão tantos estados seguintes possíveis quantos forem os valores possíveis para aquela variável. Por isso, na linha 11 do código *Exemplo de Case*, a condição é 1 e atribui-se o valor *state1*. Isso significa que, se nenhuma das cláusulas anteriores for verdadeira, o próximo estado da variável *ctl_state* será *state1*.

DEFINE

Para tornar a especificação NuSMV mais concisa, pode-se associar uma variável a uma expressão por meio do bloco DEFINE. Uma variável em DEFINE é sempre substituída pela sua definição quando é encontrada na especificação. Pode-se notar que não é criada uma nova variável no OBDD que é gerado. No exemplo do contador binário, cria-se a variável *carry_out*, que indica se "vai-um" ou não.

Contador Binário

```
1  MODULE counter_cell(carry_in)
2    VAR
3      value :  boolean;
4
5    ASSIGN
6      init(value) := 0;
7      next(value) := value + carry_in mod 2;
8
9    DEFINE
10     carry_out := value & carry_in;
```

Módulo principal (*main*)

Toda especificação NuSMV deve conter um módulo principal, sem parâmetros, que deve representar o sistema. Para o contador binário, definem-se as três células de contagem onde o *carry_in* da primeira é sempre um, indicando que sempre é somado um, pois este é o valor a ser somado a cada transição de estado. Para as demais, o *carry_in* é o *carry_out* da anterior, simbolizando a comunicação entre as células, como

ocorre em um *hardware* que representa este sistema. Note que todas as variáveis, neste caso, são instâncias do módulo *counter_ cell* criado.

Módulo principal

```
1  MODULE main
2    VAR
3      bit0 :  counter_cell(1);
4      bit1 :  counter_cell(bit0.carry_out);
5      bit2 :  counter_cell(bit1.carry_out);
```

Cada *bit* executará paralelamente e de forma síncrona a sua regra de transição que foi especificada no bloco ASSIGN.

SPEC

As propriedades CTLs são especificadas precedidas da palavra chave SPEC. Caso sejam fórmulas LTL, deve-se utilizar LTLSPEC. As demais formas de especificações não fazem parte deste trabalho e podem ser encontradas no manual de NuSMV [CCO+05b].

As propriedades em NuSMV são na verdade perguntas realizadas ao sistema modelado pelo OBDD. Assim, no exemplo do contador binário, pode-se perguntar se é sempre verdade que em algum estado no futuro *carry_ out* do bit2 será verdadeiro. Essa propriedade é definida na linha 8 do código *Propriedades* a seguir.

Como verificação do contador, pode-se perguntar se, obtendo o valor 000, o próximo será 001, considerando *bit0* como o menos significativo. Esta propriedade é mostrada pela fórmula LTL na linha 11 do código *Propriedades* a seguir.

Propriedades

```
1  MODULE main
2    VAR
3      bit0 :  counter_cell(1);
4      bit1 :  counter_cell(bit0.carry_out);
5      bit2 :  counter_cell(bit1.carry_out);
6
7    SPEC
8      AG AF bit2.carry_out
9
10   LTLSPEC
11     G((!bit0.value & !bit1.value & !bit2.value) -> X bit0.value)
```

Assim, completa-se a especificação NuSMV de um contador binário de três *bits*. A execução deste modelo retorna verdadeira para ambas as perguntas realizadas, formalizadas pelas fórmulas CTL e LTL mostradas. A seguir, são apresentadas outras construções importantes para a verificação em Método de Refinamento Machĭna.

INVAR

É possível definir invariantes do sistema via o bloco INVAR. Este bloco restringe o domínio no qual uma variável de estado pertence. Seu impacto na verificação é uma redução do OBDD quanto aos possíveis valores. Embora o tamanho do OBDD não tenha relação direta com desempenho, os cortes podem eliminar caminhos que não são desejáveis em um determinado ponto.

ISA

Em alguns casos, pode ser necessário que módulos compartilhem algumas de suas partes com outros módulos. Em NuSMV, isso pode ser feito separando as partes comuns em módulos e utilizar declarações ISA para importá-las para a declaração de outros módulos. ISA funciona como uma macro que expande o corpo do módulo importado no local da declaração, copiando as definições. Por exemplo, o módulo *moduleA* a seguir contém uma variável e uma propriedade.

Exemplo de ISA

```
1 MODULE moduleA
2   VAR
3     va :  boolean;
4
5   SPEC AG(va | !va);
```

Um outro módulo *moduleB* pode ser definido da seguinte forma, com a declaração ISA *moduleA*.

Exemplo de ISA

```
1 MODULE moduleB
2   VAR
3     vb :  boolean;
4
5   ISA moduleA
6
7   SPEC AG(vb | !vb);
```

Isso é equivalente a especificar o módulo *moduleB* da seguinte forma, onde *moduleA.va* é diferente de *moduleB.va*. Pode-se notar que até mesmo as propriedade são importadas.

Exemplo de ISA

```
1 MODULE moduleB
2   VAR
3     va :  boolean;
4     vb :  boolean;
5
6   SPEC AG(va | !va);
7   SPEC AG(vb | !vb);
```

No caso de um módulo *moduleC* importar o módulo *moduleB* utilizando ISA, este também importa os elementos do *moduleA*. Por exemplo, o módulo

Exemplo de ISA

```
1 MODULE moduleC
2   VAR
3     vc :  boolean;
4
5   ISA moduleB
6
7   SPEC AG(vc | !vc);
```

é equivalente à seguinte especificação.

Exemplo de ISA

```
1 MODULE moduleC
2   VAR
3     va :  boolean;
4     vb :  boolean;
5     vc :  boolean;
6
7   SPEC AG(va | !va);
8   SPEC AG(vb | !vb);
9   SPEC AG(vc | !vb);
```

process

Duas instâncias de módulos executam a regra de transição em paralelo e de forma síncrona. Caso sejam declaradas como processo (*process*), elas simulam uma execução assíncrona, onde a cada transição de estado, escolhe-se não deterministicamente qual processo a ser executado. Os demais processos, que não foram escolhidos para execução, não alteram o valor de suas variáveis. O exemplo a seguir mostra três portões lógicos que invertem a entrada recebida. A saída de uma portão é a entrada do seguinte, formando um anel. A declaração dos três portões como processo simula o atraso no circuito, pois cada portão tem sua vez de executar não deterministicamente. A declaração FAIRNESS *running*, na linha 19, garante que o processo que simboliza o portão lógico será executado pelo menos uma vez.

Portão inverte

```
1  MODULE main
2    VAR
3      gate1 :  process inverter(gate3.output);
4      gate2 :  process inverter(gate1.output);
5      gate3 :  process inverter(gate2.output);
6
7    SPEC
8      (AG AF gate1.output) & (AG AF !gate1.output)
9
10
11 MODULE inverter(input)
12   VAR
13     output :  boolean;
14
15   ASSIGN
16     init(output) := 0;
17     next(output) := !input;
18
19   FAIRNESS running
```

Com estes elementos básicos de NuSMV, pode-se mostrar como uma especificação Machĭna pode ser transformada em uma especificação NuSMV, e com isso, será possível obter um método de verificação para Método de Refinamento Machĭna.

4.1.3 Verificador de Modelos para ASM

Existem vários trabalhos que propõem verificação a diversos modelos de especificação de sistemas. Por exemplo, existem extensões ligadas à UML, como apresentado por Johan Lilius e Iván Paltor em [LP99b, LP99a], que realizam a verificação automática do modelo, neste caso utilizam-se os verificadores de modelos SPIN e PROMELA. A abordagem encontrada em [BBK+04] também realiza a verificação de diagramas de estados, porém realiza-se prova por execução simbólica de premissas utilizando o provador de teoremas KIV [Rei92].

Outra abordagem muito utilizada, ainda sobre UML, é a apresentação de uma semântica formal, principalmente de diagramas de transição de estados e atividades, que possam ser posteriormente verificados [Obe03, KM02, BCR00b, BCR00a]. Como MRM está imerso no modelo ASM e LMM é baseada em diagramas, são mostrados trabalhos que relacionam UML, linguagem gráfica, com o modelo ASM. Por exemplo, Ileana Ober [Obe03] define uma semântica para UML por meio de um meta-modelo utilizando ASM, enquanto Alexander Knapp e Stephan Merz [KM02] propõem uma geração de código ASM a partir de máquinas de estados UML. Börger, Cavarra e Riccobene também descrevem uma semântica formal para UML em termos de ASM, possibilitando utilizar-se do rigor matemático deste modelo para eliminação de ambigüidades e verificação do sistema [BCR00b, BCR00a].

Para o modelo ASM, são apresentadas algumas maneiras de verificação automática, como os trabalhos de Spielmann [Spi99, Spi00] e Kirsten Winter [Win01], utilizando verificação de modelos via transformações de uma especificação ASM para uma entrada de um verificador de modelos.

A principal pesquisa que aplica verificador de modelos a ASM é realizada por Kirsten Winter e Del Castillo [Win97, CW00, Win01], onde foi desenvolvido um módulo dentro de *The ASM Workbench* [Del99], um *framework* para desenvolvimento ASM, para verificação automática. Nesse trabalho, são apresentadas regras de refinamento sobre uma especificação ASM, feita na linguagem ASM-SL, com o objetivo de obter uma linguagem intermediária, chamada de ASM-IL. E a partir daí pode ser feito um segundo processo de transformações para obter entradas de diferentes verificadores de modelos. No caso, foram apresentados SMV e o pacote MDG (*Multiway Decision Graphs*) [CZS+97b].

Durante o refinamento de uma linguagem ASM para SMV, devem ser resolvidos quatro principais problemas, que são explicados a seguir:

- estados de ASM devem ser finitos (FSM);
- expansão de dinâmica de vocabulário deve ser tratada (regra ASM *import*);

- funções devem ser todas sem argumentos;
- regra de transição deve ser planificada.

Verificadores de modelos representam apenas sistemas de transição de estados finitos, portanto uma especificação ASM deve ser sempre uma máquina de estados finita (FSM - *Finite State Machine*, Seção 2.1.6) ou ser convertida para tal. Uma proposta de conversão é apresentada em [GGSV02]. Em Método de Refinamento Machĭna, o Modelo Básico é sempre descrito em termos de um FSM, representado pelo Diagrama de Transição de Estados. Com isso, essa transformação não é necessária em MRM.

As variáveis presentes na especificação de um verificador de modelos devem ser conhecidas antes de sua execução, sendo assim, regras que expandem dinamicamente o vocabulário da máquina devem ser previamente tratadas, assim como funções com aridade maior que zero. No caso do trabalho de Kirsten, a regra *import* de ASM [Gur95] não é considerada e foi descrito como trabalhos futuros.

No caso de funções com argumentos, foi gerada por Kirsten uma variável para cada valor possível do argumento passado no momento de seu uso, considerando a regra de ajuste (*Fitness Constraints*) determinada naquele ponto. Regra de ajuste determina quais os possíveis valores de uma variável em um determinado ponto do programa, sendo assim, considere o seguinte trecho de ASM.

ASM

```
1  if a < 3 then
2    f(a) := 2*a;
3  end
```

Pode-se determinar a regra de ajuste para a variável a, que pode assumir os valores 1, 2, ou 3, no trecho apresentado da seguinte forma.

$$a : \{1, 2, 3\}$$

Assim, cria-se uma variável para cada valor de a possível no ponto de uso da função f e obtém-se o seguinte trecho de ASM-IL, onde $a = 3$ não atualiza f.

ASM-IL

```
1  if a = 1 then
2    f1 := 2*1;
3  else if a = 2 then
4    f2 := 2*2;
5  end
```

Esta abordagem fatalmente gera uma explosão de estados no OBDD de SMV, porém o tamanho do OBDD não tem relação com desempenho da execução [CGP00]. Assim, este tratamento torna-se viável para eliminar argumentos das funções.

Em SMV, a regra de transição é descrita em termos de cada variável, enquanto que em ASM a transição é descrita em termos das atualizações. Por exemplo, considere o trecho ASM a seguir.

ASM

```
1  transition:
2    if C1 then
3      a := 1;
4      if C2 then
5        c := 3;
6      end
7    end
```

O equivalente em SMV seria, no bloco ASSIGN, dado para cada variável.

SMV

```
1  ASSIGN
2    next(a) :=
3      case
4        C1 : 1;
5        1  : a;
6      esac;
7
8    next(c) :=
9      case
10       C1 & C2 : 3;
11       1 : c;
12     esac;
```

Pode-se perceber que o aninhamento de regras *if-then* em ASM deve ser refinada para uma representação equivalente em SMV. Isso significa que a regra de transição ASM deve ser planificada, ou seja, estruturas complexas (aninhadas) devem ser representadas de forma a conter somente regras do tipo:

$$if\ C\ then\ simple\_update$$

O processo descrito por Kirsten [Win01] realiza a transformação ASM-SL (*ASM-Source Language*) para SMV em duas etapas. Na primeira, estes quatro principais

problemas são resolvidos para obter uma linguagem intermediária, chamada de ASM-IL (*ASM-Itermediate Language*), com as seguintes características:

- simples: estruturas complexas, como funções com argumentos, são desdobradas;
- planificada: regra de transição representada apenas por regras *if-then* sem aninhamento.

São apresentados, em [Win01], os seguintes passos para se obter ASM-IL a partir de ASM-SL:

- desdobramento de funções (*Functions Unfolding*);
- desdobramento de regras (*Rules Unfolding*);
- simplificação de termos (*Simplifying Terms*);
- simplificação de regras (*Simplifying Rules*).

Pela simplicidade de ASM-IL, ela funciona como uma interface entre ASM e verificadores de modelos. Então, o segundo passo é realizado transformando ASM-IL em uma entrada de um verificador de modelos escolhido.

A verificação no Método de Refinamento Machĭna é baseada nos trabalhos de Kirsten e Castillo [Win97, CW00, Win01] e as regras foram aplicadas sob as mesmas perspectivas, porém realizando as adequações à linguagem Machĭna, conforme é mostrado na Seção 4.2.

4.2 Verificação Método de Refinamento Machĭna

A verificação de sistema proposta dentro do Método de Refinamento Machĭna é realizada sobre o código Machĭna gerado e utilizando o verificador de modelos NuSMV, descrito anteriormente. Deve-se converter a especificação Machĭna em uma especificação NuSMV correspondente de forma automática. Considerando as diferenças entre uma linguagem ASM e de NuSMV, dado um sistema em Machĭna, MRM o transforma em uma linguagem intermediária, denominada C-Machĭna (Core-Machĭna), Seção 4.3.

A Figura 4.3 representa as etapas de refinamento que convertem o código Machĭna em uma representação equivalente em NuSMV. Com essa abordagem de geração de uma linguagem intermediária, é possível obter uma interface para verificadores de modelos que sejam baseados em sistemas de transição e utilizam abordagem lógica. Não faz parte do escopo deste trabalho validar essa interface com outras ferramentas além de NuSMV, deixando em aberto possíveis trabalhos futuros.

Figura 4.3: Etapas de refinamento de Machĭna para NuSMV.

Primeiramente, é realizada a simplificação dos termos. Para isso, tipos elaborados são refinados, à medida do possível, para construções mais simples e as funções com aridade maior que zero são transformadas em representações equivalentes com aridade igual a zero. Posteriormente, trata-se da formação dos módulos a partir da inclusão de elementos pertencentes a outros módulos.

Então, a planificação é realizada substituindo-se regras do tipo *choose*, *with* e regras em múltiplos passos por definições simples *if-then-else*. Por fim, as regras condicionais são simplificadas para serem apenas *if-then* sem aninhamento.

4.3 C-Machĭna

C-Machĭna (Core-Machĭna) é um subconjunto da linguagem Machĭna onde existem construções e regras mais simples. Suas características são:

- todas as regras são planificadas, ou seja, comandos aninhados são transformados em regras simples do tipo *if g then R; end*, inclusive sequenciadores de regras e chamadas a abstrações simples (abstrações que não possuem parâmetros, declarações locais, *loops* e nem seqüência de passos de execução (**step**));
- funções com aridade maior que zero são transformadas em uma representação equivalente que seja declarada e avaliada antes da execução, ou seja, não existe avaliação dependente de estado, todas as funções devem ter aridade zero;
- apenas existem os tipos básicos **Bool**, **Int**, **Char**, intervalo, enumeração e o tipo agente;
- modularização, mecanismo de visibilidade e cláusulas *include* e *import* são mantidos em C-Machĭna com a mesma semântica que em Machĭna;
- abreviatura de termos e abstrações regras não são permitidas em C-Machĭna;
- as regras de administração de agentes são realizadas por meio de funções explícitas, e não com regras e funções implícitas como em Machĭna;
- definições de regras em múltiplos passos não são permitidas;

- não são permitidas as regras *choose*, *forall*, *case* ou *with*;
- expressões elaboradas, como *all* e *exist*, não são permitidas;
- agentes possuem a mesma semântica de declaração e execução que em Machĭna;
- a definição da máquina de execução é realizada da mesma forma que em Machĭna.

C-Machĭna é um importante recurso na Verificação Método de Refinamento Machĭna, pois sua simplicidade permite transformar mais facilmente um programa Machĭna em uma especificação de verificadores de modelo. Neste trabalho, apenas é abordada a ferramenta NuSMV, mas trabalhos futuros podem utilizar C-Machĭna como interface para outros verificadores de modelos, permitindo o uso de outras ferramentas.

4.4 Mapeamento de Machĭna a C-Machĭna

O refinamento de Machĭna para C-Machĭna considera problemas sintáticos, semânticos, verificação de tipos e controle de visibilidade já resolvidos.

4.4.1 Refinamento da álgebra: tipos

Os tipos em Machĭna devem ser transformados em estruturas equivalentes mais simples, principalmente os tipos estruturados. De acordo com o refinamento de dados, não somente a estrutura é alterada, mas também as operações devem ser refinadas de forma a preservar o efeito sobre o dado.

Básicos

Os tipos básicos de Machĭna são **Bool**, **Char**, **Int**, **Real**, **String** e intervalos. No refinamento de Machĭna para C-Machĭna suas representações são mantidas. Porém, abstrações de operações, como *abs*, *max* e *min*, *succ* e *pred* devem ser substituídas por operações mais básicas. Demais operações existentes são mantidas.

Para o tipo **Char**, existem as operações abstratas:

- *ord*(c : **Char**):**Int**, retorna o valor ASCII do caractere c;
- *chr*(i : **Int**):**Char**, retorna o caractere com o código ASCII i;
- *succ*(c : **Char**):**Char**, retorna o sucessor de c ou **undef**;
- *pred*(c : **Char**):**Char**, retorna o predecessor de c ou **undef**.

As operações *ord* e *chr* são mantidas em C-Machĭna. Já as operações *succ* e *pred* são refinadas respectivamente nas seguintes expressões. Primeiro avalia-se se estão no intervalo de caracteres válidos e retornam o sucessor, ou predecessor, e caso contrário retornam o caractere com código ASCII zero, que é equivalente ao **undef**. Vale lembrar que código ASCII do intervalo válido de caracteres é de 32 a 126, então 32 não tem predecessor e 126 não tem sucessor.

succ

```
1  if ord(c) >= 32 and ord(c) <  126 then
2    chr(ord(c)+1);
3  else
4    chr(0);
5  end
```

pred

```
1  if ord(c) > 32 and ord(c) <=  126 then
2    chr(ord(c)-1);
3  else
4    chr(0);
5  end
```

Para o tipo **Int** existem as operações abstratas:

- *abs*(i : **Int**):**Int**, retorna o valor absoluto de i;
- *max*(i : **Int**, j : **Int**):**Int**, retorna o maior valor entre dois números inteiros, i ou j;
- *max*(i : **Int**, j : **Int**):**Int**, retorna o menor valor entre dois números inteiros, i ou j;
- *sqr*(i : **Int**):**Int**, retorna o quadrado de i.

Cada uma delas são respectivamente refinadas em expressões conforme a seguir.

abs

```
1  if i >= 0 then a else (-1 * a) end;
```

max

```
1  if i >= j then i else j end;
```

min

```
1  if i <= j then i else j end;
```

sqr

```
1  i * i;
```

O mesmo ocorre com as operações abstratas do tipo **Real**, preservando as conversões *integer*, que transforma um real em um inteiro, e *real*, que converte um inteiro em um real.

Compostos

Os tipos compostos são enumerações, união disjunta, tuplas, conjuntos, lista, nodo de árvore, funcional, agente, abstração de regras, arquivo e tipos definidos pelo programador. Os tipos enumeração, lista, nodo árvore, funcional, agente e abstração de regras são mantidos inalterados. A seguir são apresentados os refinamentos dos demais.

O tipo **?** é definido como a união disjunta de todos os tipos possíveis, então esse refinamento é realizado considerando os tipos visíveis dentro do módulo.

A união disjunta é decomposta criando uma função para cada tipo que forma a união. Em Machĭna, cada função tem um atributo interno *type* que indica qual o seu tipo em termos de um inteiro. No caso de uma união disjunta, este atributo assume um valor de acordo com a atribuição que é feita à função. Para este controle, define-se uma função adicional que deve sempre refletir o comportamento da união disjunta. Assim, a união disjunta *a : T1 | T2* é refinada para os seguintes termos.

União disjunta

```
1  aT1 : T1;          // função para T1
2  aT2 : T2;          // função para T2
3  aType : Int;       // controla o tipo de a, T1 ou T2
```

A obtenção do valor de *a* é dado pela regra *with*, a ser discutida mais adiante. Quando houver uma atribuição *a := b*, o seguinte refinamento é realizado. Pode-se perceber que é analisado qual o tipo de *b*, então a atribuição é corretamente realizada e o controlador do tipo de *a* é atualizado.

Atribuição em união disjunta

```
1  if b.type = T1 then
2    aT1 := b;
3    aType := aT1.type;
4  elseif b.type = T2 then
5    aT2 := b;
6    aType := aT2.type;
7  end
```

De forma semelhante à união disjunta, o tipo tupla é decomposto de acordo com seus elementos e as operações são refinadas preservando a semântica. Então, considerando as definições das tuplas p e q e as operações a seguir.

Tuplas

```
1  p, q : tuple (x : Int, y : Int);
2  . . .
3  q := p;
4  q.x := p.y;
5  p := (1,2);
```

O refinamento é dado da seguinte forma.

Refinamento de tuplas

```
1  // declaração da tupla p
2  tuple_p_x : Int;
3  tuple_p_y : Int;
4
5  // declaração da tupla q
6  tuple_q_x : Int;
7  tuple_q_y : Int;
8
9  // refinamento da operação da linha 3
10 tuple_q_x := tuple_p_x;
11 tuple_q_y := tuple_p_y;
12   // refinamento da operação da linha 4
13 tuple_q_x := tuple_p_y;
14
15 // refinamento da operação da linha 5
16 tuple_p_x := 1;
17 tuple_p_y := 2;
```

O tipo conjunto denota uma coleção de elementos do mesmo tipo. Sendo assim,

pode-se interpretar conjuntos como uma relação, e o refinamento é realizado para uma função que mapeia o tipo do conjunto em um valor booleano, indicando se o elemento pertence ou não à coleção.

Conjunto

```
1  s = set of T;
```

A declaração do conjunto apresentada é então transformada em uma função conforme a seguir.

Rrefinamento de conjunto

```
1  s : T -> Bool;
```

As operações sobre conjuntos são:

- s1 + s2, união;
- s1 - s2, diferença;
- s1 * s2, interseção;
- x **in** s, pertinência;
- s(x), pertinência.

A seguir, apresenta-se, respectivamente, o refinamento de cada uma delas, considerando os conjuntos *s1*, *s2* e *s3* do tipo T. A pertinência dada por *x **in** s* é tratada com a sintaxe de *s(x)*, que é preservada.

A união *s1 + s2* seleciona todos os elementos pertencentes ao domínio T que satisfazem a relação *s1* ou *s2* e atribui a *s3*.

União: s1 + s2

```
1  forall x:T
2  satisfying s1(x) or s2(x)
3  do s3(x) := true;
4  end
```

A diferença *s1 - s2* seleciona todos os elementos pertencentes ao domínio T que satisfazem a relação *s1* e não satisfazem *s2*, então as atribui a *s3*.

Diferença: s1 - s2

```
1  forall x:T
2  satisfying s1(x) and not s2(x)
3  do s3(x) := true;
4  end
```

A interseção *s1 * s2* é refinada de forma semelhante, porém inclui em *s3* os elementos que satisfazem tanto a relação *s1* quanto *s2*.

Interseção: s1 * s2

```
1  forall x:T
2  satisfying s1(x) and s2(x)
3  do s3(x) := true;
4  end
```

No caso de uma atribuição de expressão *s3 := {a, b, c}* faz-se a expansão relacionando cada elemento.

Atribuição

```
1  s3(a) := true;
2  s3(b) := true;
3  s3(c) := true;
```

Funções definidas como arquivos têm seu uso, lado direito da expressão, substituído pela regra não determinística *choose*, onde o domínio corresponde ao tipo do arquivo. Este refinamento considera arquivos como sendo uma interação com o ambiente externo e o aborda como um valor não determinístico. No trecho de código a seguir, considera-se *readT* a função que lê um elemento do arquivo *f*, que possui o tipo *T*, e o atribui a *x*.

Arquivo

```
1  f : file of T;
2  x : T;
3  . . .
4  readT(f,x);
```

O refinamento então é dado pela seguinte regra, percebe-se que não existe mais a declaração do arquivo. Preservando a declaração de *x*, apenas refina-se a obtenção do valor, *readT* na linha 4.

Refinamento de arquivo

```
1  x : T;
2  . . .
3  choose f : T
4  do x := f;
5  end
```

Alguns tipos podem ser definidos pelo programador a fim de melhorar a legibilidade do código. Embora Machĭna permita definição de tipos com parâmetros, isto não é tratado neste trabalho, assim como definição de tipos mutuamente recursivos. O refinamento, então, apenas substitui o tipo criado pela sua definição.

Novo tipo

```
1  type T = Int;
2  a : T;
```

Refinamento

```
1  a : Int;
```

4.4.2 Refinamento da álgebra: funções

As funções são refinadas aplicando o método proposto por Kirsten e Castillo [Win97, CW00, Win01], conforme a descrição na Seção 4.1.3, onde são criadas, para cada função com argumentos, todas as possibilidades de acordo com o domínio do parâmetro. Neste caso, a regra de ajuste *Fitness Constraint* para cada ponto de chamada de função deve ser informada pelo desenvolvedor. Com isso, o processo de transformação exige a intervenção humana neste ponto, impedindo que a verificação no Método de Refinamento Machĭna seja uma tarefa completamente automática.

Foge do escopo deste trabalho o tratamento da aplicação da regra de ajuste. Embora seja necessário a intervenção do desenvolvedor, os ganhos com as demais regras de refinamento para obter a entrada de um verificador de modelos são consideráveis. Futuros trabalhos podem ser realizados para tornar a aplicação da regra de ajuste um processo automático.

4.4.3 Refinamento da álgebra: *import*

A cláusula *import* tem papel fundamental no controle de visibilidade de tipos e abstrações exportadas por um módulo, não havendo necessidade de ser refinada em C-

Machĭna, considerando que qualquer problema relacionado a acesso inválido já tenha sido resolvido em Machĭna.

4.4.4 Refinamento da álgebra: *include*

Elementos públicos

Para todo módulo A, faz-se a expansão da cláusula *include* da seguinte forma. Para cada módulo incluído em A, verifica-se sua cláusula *include* e adiciona, se ainda não existir, os módulos por este incluído na cláusula *include* de A. Esse processo deve ser repetido até que se elimine toda a dependência de inclusões existentes. Embora a semântica da cláusula *include* em C-Machĭna seja a mesma que em Machĭna, essa expansão é fundamental para o refinamento de Machĭna em NuSMV, como descreve a Seção 4.5.2.

Posteriormente, cria-se um segundo módulo, chamado *publicA*, que contém os elementos públicos de A, com exceção dos elementos compartilhados. Então, para todo módulo Bi presente na cláusula *include* de A, realiza-se a troca para incluir *publicBi*, e também faz-se incluir *publicA*.

Considerando o seguinte trecho de código

Machĭna

```
1  module A
2    include B;
3  end A
4
5  module B
6    include C;
7  end B
```

O resultado da expansão é apresentado a seguir.

C-Machĭna: expansão de *include*

```
1  module A
2    include B, C;
3  end A
4
5  module B
6    include C;
7  end B
```

Enfim, o resultado do refinamento é apresentado no seguinte código.

C-Machĭna

```
1  module A
2    include publicA, publicB, publicC;
3  end A
4
5  module B
6    include publicB, publicC;
7  end B
8
9  module publicA
10   // elementos públicos de A
11 end publicA
12
13 module publicB
14   // elementos públicos de B
15 end publicB
16
17 module publicC
18   // elementos públicos de C
19 end publicC
```

Este refinamento simplifica o tratamento da cláusula *include*. Ao gerar uma área comum, no caso *publicA*, para os módulos que incluem A e o próprio A, é possível tratar todos eles da mesma forma. A vantagem fica evidente no refinamento de C-Machĭna para NuSMV, Seção 4.5.2.

Elementos compartilhados

Considerando a expansão realizada anteriormente, para todo módulo A, as funções compartilhadas (*shared*) são transferidas para um módulo a parte, denominado *sharedA*, e as funções compartilhadas públicas (*public shared*) são transferidas para outro módulo, denominado *publicSharedA*. Então, o módulo A passa a incluir tanto *sharedA* quanto *publicSharedA*, e para os demais módulos *publicBi* presentes na cláusula *include* de A, adiciona-se o módulo *publicSharedBi*.

Portanto, considerando o seguinte trecho de código, onde o módulo C possui elementos compartilhados.

C-Machĭna: módulo C possui elementos compartilhados.

```
1  module A
2    include publicA, publicB, publicC;
3  end A
4
5  module B
6    include publicB, publicC;
7  end B
8
9  module C
10   include publicC;
11 end C
12 . . .
```

O resultado do refinamento é apresentado a seguir.

C-Machĭna

```
1  module A
2    include publicA, publicB, publicC, publicSharedC;
3  end A
4
5  module B
6    include publicC, publicSharedC;
7  end B
8
9  module C
10   include publicC, sharedC, publicSharedC;
11 end C
12
13 module publicA
14   // elementos públicos de A
15 end publicA
16
17 module publicB
18   // elementos públicos de B
19 end publicB
20
21 module publicC
22   // elementos públicos de C
23 end publicC
24
25 module sharedC
26   // elementos compartilhados de C
27 end sharedC
```

```
28
29    module publicSharedC
30      // elementos públicos e compartilhados de C
31    end publicSharedC
```

A criação de módulos separados contendo os elementos da álgebra preserva a mesma semântica de Machĭna quanto aos elementos públicos, compartilhados e públicos-compartilhados. O objetivo destes refinamentos é tratar estes três grupos de funções adequadamente. Assim, a formação do módulo pela cláusula *include* é realizada igualmente para todos os módulos e cada grupo receberá os cuidados necessários quando for feito o refinamento para NuSMV, apresentado na Seção 4.5.

4.4.5 Refinamento de expressões

As expressões **case**, **if-then-else** e **with**, quando aparecem como lado direito de uma atribuição, são refinadas para sua regra correspondente, realizando a atribuição no corpo da regra. Por exemplo, considerando o código a seguir.

Expresões

```
1    a := if x then 1 else 2 end;
```

Transforma-se esta atribuição na seguinte regra *if-then-else*. As demais expressões citadas são tratadas da mesma forma.

Refinamento

```
1    if x then
2      a := 1;
3    else
4      a := 2;
5    end
```

Caso as expressões apareçam como guardas, ou seja, são avaliadas como expressão booleana, faz-se a conjunção conforme o exemplo a seguir.

Expressões booleanas

```
1    if if g then p else q end then
2      R;
3    end
```

Refinamento

```
1  if (g and p) or (not g and q) then
2    R;
3  end
```

Este segundo tratamento somente é possível pois p e q são expressões booleanas, e considera-se este problema já resolvido na especificação em Machĭna.

As expressões **exists** e **all** são, por definição, expressões booleanas. Ambas são refinadas realizando a expansão de seus elementos, construindo-se uma fórmula lógica a partir da expressão contida em **satisfying**. Dada a seguinte expressão **exists** e considerando o domínio do tipo T como sendo $\{1, 2, 3\}$.

Expressões

```
1  a := exists x : T satisfying x > 0;
```

O refinamento realiza a expansão via disjunção, ou conjunção no caso da expressão **all**, de todas as expressões possíveis formadas pela substituição de x, conforme a seguir.

Refinamento

```
1    a := (1 > 0) or (2 > 0) or (3 > 0);
```

A seguir, é mostrado um exemplo onde a expressão é constituída de mais de um elemento, x e y.

Expressões

```
1  b := exists x : T, y : T satisfying x > 0 and y < 5;
```

Refinamento

```
1  b := (1 > 0 and 1 < 5) or (1 > 0 and 2 < 5) or (1 > 0 and 3 < 5) or
2       (2 > 0 and 1 < 5) or (2 > 0 and 2 < 5) or (2 > 0 and 3 < 5) or
3       (3 > 0 and 1 < 5) or (3 > 0 and 2 < 5) or (3 > 0 and 3 < 5);
```

4.4.6 Refinamento de regras de transição

Abreviatura de termos

Dada uma regra conforme a seguir, toda ocorrência de v é trocada pela sua definição *exp* considerando o escopo correspondente.

let

```
1  let v := exp;
2  a := v;
```

Refinamento

```
1  a := exp;
```

Estado de execução

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

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

Adicionalmente, todo agente tem uma função **self** que retorna a sua identificação. No refinamento de Machĭna para C-Machĭna, a função **self** fica inalterada.

O atributo implícito **state** é refinado para a função enumerada *state* onde os valores possíveis são *anew*, *active*, *stopped*, *blocked* e *destroyed* e o valor padrão é *anew*. Assim, a regra de transição *RT* é envolvida pela condição *state = active*. A seguir, são mostradas a definição da função *state* e a alteração realizada na regra de transição em C-Machĭna.

```
1  algebra:
2    state : public enum {anew, active,
3                         stopped, blocked,
4                         destroyed}  default anew;
5
6  transition:
7    if state = active then RT; end
```

Administração de agentes

Para administrar um agente em Machĭna, existem as regras **create**, **dispatch**, **stop**, **destroy** e **return**. Todas elas podem ser compreendidas como açúcar sintático para atribuições de valores à função **state**, que representa o estado de execução de um agente, como mostrado a seguir.

create agnt

```
1  agnt.state := anew;
```

dispatch agnt

```
1  if agnt.state != blocked and agnt.state != destroyed then
2    agnt.state := active;
3  end
```

stop

```
1  if self.state != anew and self.state != destroyed then
2    self.state := stopped;
3  end
```

destroy agnt

```
1  agnt.state := destroyed;
```

A criação de um agente é realizada somente uma vez e antes de qualquer outra operação sobre *state*. Estas condições são resolvidas durante a especificação em Machĭna. O disparo de um agente somente pode ser realizado se este já estiver criado, e ainda não executando, ou se estiver parado. Se estiver em execução, o disparo não tem efeito, pois o valor de *state* não se altera. Um agente destruído não pode voltar a execução. No caso de estar bloqueado, o agente se encontra em execução aguardando uma resposta, por isso não se pode alterar o valor de *state* para active neste momento.

Somente faz sentido parar a execução de um agente se o mesmo estiver em execução, ativo ou bloqueado. Por fim, a destruição de um agente pode ser realizada em qualquer circunstância.

Então, estas regas são assim refinadas. No caso de *agnt* ser uma lista de agentes, deve-se gerar uma regra para cada $a \in agnt$. A regra **return** é tratada durante o

refinamento das abstrações. É importante ressaltar que **return** somente é utilizado dentro do corpo de abstrações que contém *loop*.

Abstrações simples

O refinamento de Machĭna em C-Machĭna deve substituir as chamadas de abstrações simples, pertencentes ao agente, pelo seu corpo de execução. Uma abstração é simples quando não possui parâmetros, declarações locais, *loop* e nem seqüência de passos de execução (**step**). Assim, seu código pode ser introduzido no local de sua chamada.

Caso a abstração possua uma pré-condição, deve-se gerar um marcador indicando que essa pré-condição deve ser satisfeita naquele momento. Como a pré-condição de abstrações são formuladas com funções do módulo, não existe problema de escopo e pode-se colocar o termo ***require:*** *expression* antes do bloco que representa o corpo da ação. O mesmo ocorre para a pós-condição, e ambas se tornam asserções durante as transições. Por exemplo, a ação *actionA* tem a regra R como corpo, P como pré-condição e Q como pós-condição.

Ações Simples

```
1  action actionA
2    require P;
3    ensure  Q;
4    R;
5  end
```

Quando houver uma chamada à *actionA*, esta será substituída pelo código a seguir.

Refinamento

```
1    require P;
2    R;
3    ensure  Q;
```

Abstrações

As abstrações que possuem parâmetros, declarações locais ou seqüência de regras devem ser refinadas em uma especificação NuSMV separada, e as demais regras de refinamento são aplicadas normalmente. Caso haja alguma função declarada no módulo, esta deve ser reproduzida na nova especificação NuSMV.

A pós-condição da ação deve estar bem descrita e indicar o efeito na álgebra do módulo. Assim, as alterações realizadas pela ação são preservadas em seu ponto de

chamada. Por exemplo, a ação *inc* demonstrada a seguir tem o efeito de incrementar o valor da função passada como parâmetro via o argumento x. A função a é iniciada com o valor 0 (zero) e a regra de transição incrementa constantemente o seu valor.

Ações

```
1  algebra:
2    a : Int := 0;
3
4  abstractions:
5    action inc (out x : Int) is
6      ensure x = old x + 1;
7
8      x := x + 1;
9    end
10
11 transition:
12   inc(a);
```

Pode-se notar que a cláusula **ensure**, que representa a pós-condição de *inc*, indica o efeito na álgebra. No refinamento para C-Machĭna, ela substitui o corpo da ação, onde o parâmetro x é trocado pela função a. Com isso, a utilização de **old** não se faz mais necessária. Além disso, a cláusula **ensure** é preservada, da mesma forma que foi feito para ações simples, conforme mostrado a seguir.

Refinamento

```
1  transition:
2     a := a + 1;
3     ensure a := old a + 1;
```

Ações com *loop* não serão tratadas neste trabalho e podem ser realizados trabalhos futuros para obter seu correto refinamento.

Regras com seqüência de passos de execução (step)

Quando a regra de transição for executada em mais de um passo (cláusula **step**), deve-se realizar o refinamento para uma regra *if-then* equivalente. Em Machĭna não é permitido aninhamento das seqüências, ou seja, na presença de uma cláusula **step**, é certo de que somente existirá uma seqüência. O mesmo ocorre em regras com mais de um passo dentro de ações, e por serem ações não simples terão um tratamento adequado no refinamento de forma a não violar essa restrição.

Então, para a transformação de Machĭna em C-Machĭna, cria-se uma função *step* do tipo intervalo, que tem seus limites definidos de acordo com o número de passos possíveis e o valor inicial é 1. Assim, o conjunto de regras que eram executadas em um determinado passo i, passa a ser executado sob a condição de *step* possuir o valor i.

Em Machĭna, **next** é utilizado para atualizar o valor de **step** indiretamente, auxiliando no controle de visibilidade. Em C-Machĭna isso não é mais necessário, então sempre que aparecer **next** como regra em um passo, ela é substituída pela atualização explícita da função *step*.

O limite inferior do intervalo *step* é, por definição, 1. Considere o limite superior como sendo n, inicialmente 1, o conjunto X de passos definidos e a função *max(x,y,n)* que retorna o maior valor entre x, y e z. Então, para toda cláusula **step**

Machĭna

```
1  step x : Rx; next := y;
```

gera-se o trecho em C-Machĭna

C-Machĭna

```
1  if step = x then Rx; step := y; end
```

O refinamento deve ainda realizar os seguintes cálculos, escritas em notações matemática, onde o limite superior n é definido tomando-se o maior valor entre o passo x, a atribuição y e o próprio n. O conjunto X é acrescido do passo x avaliado.

$$n \leftarrow max(x, y, n)$$

$$X \leftarrow X + \{x\}$$

Na ausência de atualização **next**

Machĭna

```
1  step x : Rx;
```

gera-se o seguinte trecho em C-Machĭna

C-Machĭna

```
1  if step = x then Rx; end
```

As operações realizadas são as mesmas com a diferença que não existe y, então n recebe o maior valor entre o passo x e o próprio n.

$$n \leftarrow max(x, 0, n)$$

$$X \leftarrow X + \{x\}$$

Com isso, todas as regras de atualização (Rx) dos passos definidos são transformadas em execuções sob a condição de *step*. Caso não exista atualização via **next**, nada é feito por enquanto. Ao final desta etapa, o limite superior do intervalo *step* é dado por n e o conjunto X contém todos os passos definidos.

É possível omitir passos, por exemplo define-se o passo 1 e depois o 3, omitindo-se assim o passo 2. Dado o intervalo de 1 a n, o conjunto O é formado pelos passos omitidos, aqueles não presentes em X.

$$O \leftarrow \{1..n\} - X$$

Então, para todo $o \in O$, cria-se uma regra condicional da mesma forma que foi feito para os passos definidos. Porém, a única regra presente é a atualização de *step* para o valor seguinte, como mostra o código a seguir, onde o é um elemento de O.

C-Machĭna

```
1  if step = o then step := step + 1; end
```

Por fim, todas as regras condicionais geradas que não contém **next** devem ser acrescidas da regra de atualização *step := step + 1;*. Porém, é possível que a regra de transição seja repetida, isso ocorre quando ela estiver marcada com **transition** ou **loop**. Neste caso, o último passo, passo n, deve atualizar *step* para 1, mas isso somente é feito se não houver outra atualização já definida.

Essa última etapa permite que *step* atinja o valor *n+1* quando a regra de transição não é demarcada como repetição, pois no passo n terá a atualização *step := step + 1;*. Portanto, o limite superior do intervalo de *step* em C-Machĭna deve ser alterado para *n+1*.

Dessa forma, a correspondência com a semântica de **step** em Machĭna é preservada. A seqüência se inicia no passo 1 e toda transição atualiza o valor de **step** com o valor de **next**, e este é sempre incrementado na ausência de uma atualização explícita. O exemplo a seguir mostra a aplicação destas regras de refinamento. Dados os passos

Sequenciamento de Regra

```
1  . . . . .
2  transition:
3    step 1: R1;
4    step 3: R3; next := 4;
5    step 4: R4; next := 8;
6    step 5: R5; next := 3;
7
8  . . . . .
```

A primeira etapa gera o seguinte trecho de código

Refinamento: etapa 1

```
1  algebra
2    step : 1 .. n;
3
4  initial
5    step := 1;
6  . . . . .
7  if step = 1 then R1;              end
8  if step = 3 then R3; step := 4; end
9  if step = 4 then R4; step := 8; end
10 if step = 5 then R5; step := 3; end
```

A partir disso, o refinamento calcula os seguintes valores para n e X

$$n = 8$$

$$X = \{1, 3, 4, 5\}$$

Então, determina-se o conjunto O de passos omitidos

$$O = \{2, 6, 7, 8\}$$

Em seguida, o código é acrescido das regras condicionais correspondentes aos passos omitidos com a atualização de *step* para o passo seguinte. Note que no passo 8, é possível atingir o valor 9, então, por segurança, o intervalo de *step* deve ser uma unidade maior que o n definido. Por fim, os passos que não contém atualizações de *step* são acrescidos da regra *step := step + 1*, como é o caso do primeiro passo.

Refinamento: etapa 2

```
1  algebra
2    step : 1 .. 9;
3  . . . . .
4  if step = 1 then R1; step := 2; end
5  if step = 2 then     step := 3; end
6  if step = 3 then R3; step := 4; end
7  if step = 4 then R4; step := 8; end
8  if step = 5 then R5; step := 3; end
9  if step = 6 then     step := 7; end
10 if step = 7 then     step := 8; end
11 if step = 8 then     step := 9; end
```

Se a regra de transição em Machĭna não for repetida, este é o resultado final. Caso contrário, deve-se alterar o passo 8 para reinicializar *step* e recomeçar a executar as regras. Assim tem-se

Refinamento: etapa 3

```
1  algebra
2    step : 1 .. 9;
3  . . . . .
4  if step = 1 then R1; step := 2; end
5  if step = 2 then     step := 3; end
6  if step = 3 then R3; step := 4; end
7  if step = 4 then R4; step := 8; end
8  if step = 5 then R5; step := 6; end
9  if step = 6 then     step := 7; end
10 if step = 7 then     step := 8; end
11 if step = 8 then     step := 1; end
```

Esse refinamento preserva a semântica de regras em múltiplos passos definida em Machĭna. É possível realizar um tratamento no código final gerado para fins de otimizações. Por exemplo, os passos 2, 6 e 7 nada fazem e poderiam ser excluídos e as atualizações para eles alterada para o passo seguinte. Porém, isso não é parte deste trabalho, podendo ser futuramente realizado. Da forma apresentada, é preciso apenas garantir que nos passos omissos apenas seja atualizada a função *step* e nenhuma outra mais do vocabulário do módulo.

Regra *choose*

Para elementos com tipos básicos, como inteiro e booleano, a regra **choose** pode ser refinada para uma função externa, onde o tipo é o mesmo do elemento, e a expressão

em **satisfying** se transforma em um invariante do sistema.

choose

```
1  choose i : Int
2  satisfying i > 0 and i < 10
3  do f(i) := i * 2;
```

O exemplo acima é refinado da seguinte forma, considerando x como índice da regra **choose**, definido pela ordem em que foram encontradas.

Refinamento

```
1  algebra:
2      // função que representa a regra choose
3      external iChoose_x : Int;
4      . . .
5  transition:
6      // regra de transição de choose
7      f(iChoose_x) := iChoose_x * 2;
8      . . .
9  invariant:
10     // invariante a partir de satisfying
11     iChoose_x > 0 and iChoose_x < 10;
```

Regra *with*

A regra **with** pode ser utilizada em dois momentos, para elementos de um valor composto, como lista, tupla ou nodo de árvore, ou então para inspecionar o tipo de uma união disjunta. Considerando apenas o segundo caso, refina-se **with** para uma regra condicional, onde cada *if-then* corresponde a uma cláusula. O elemento a ser comparado é o controlador do tipo da união disjunta e a regra a ser realizada tem o padrão substituído pela função correspondente, de acordo com o tipo.

with

```
1  a : T1 | T2;
2  . . .
3  with a
4    as a1 : T1 => f := a1;
5    as a2 : T2 => g := a2;
6    otherwise  => f := 0; g := 0;
7  end
```

O exemplo acima é refinado para a seguinte regra condicional.

Refinamento

```
1  // refinamento da união disjunta
2  aT1 : T1;
3  aT2 : T2;
4  aType : Int;
5
6  // refinamento da regra with
7  if aType = T1.type then
8    f:= aT1;
9  elseif aType = T2.type then
10   g:= aT2;
11 else
12   f := 0; g := 0;
13 end
```

Regra *forall*

A regra **forall** é expandida criando-se uma regra condicional para cada possível valor do domínio de seus elementos e condicionada à expressão contida em **satisfying**. Tanto a regra quanto a expressão tem os elementos substituídos pelo valor obtido na expansão.

forall

```
1  forall x : 1..3, y : 4..6
2  satisfying x%2 = 1
3  do f(x,y) := x + y;
```

No exemplo acima, o elemento x pertence ao domínio $\{1, 2, 3\}$ e y ao $\{4, 5, 6\}$. A expressão de **satisfying** restringe apenas os valores de x. A seguir, apresenta-se um refinamento para esta regra.

Refinamento

```
1  // x = 1 e valores de y
2  if 1%2 = 1 then f(1,4) := 1 + 4; end
3  if 1%2 = 1 then f(1,5) := 1 + 5; end
4  if 1%2 = 1 then f(1,6) := 1 + 6; end
5  // x = 2 e valores de y
6  if 2%2 = 1 then f(2,4) := 2 + 4; end
7  if 2%2 = 1 then f(2,5) := 2 + 5; end
8  if 2%2 = 1 then f(2,6) := 2 + 6; end
```

```
9   // x = 3 e valores de y
10  if 3%2 = 1 then f(3,4) := 3 + 4; end
11  if 3%2 = 1 then f(3,5) := 3 + 5; end
12  if 3%2 = 1 then f(3,6) := 3 + 6; end
```

Regra *case*

A regra *case* é diretamente transformada em uma regra *if-then-else* equivalente, onde compara-se a expressão *e*, conforme o código a seguir, com as expressões de cada cláusula, e executa-se a regra de acordo com esta avaliação. A cláusula **otherwise**, se existir, corresponde ao *else* do último comando *if-then*.

case

```
1  case e
2    of e1 => R1;
3    of e2 => R2;
4    . . .
5    otherwise => Rk;
6  end
```

O código de exemplo de regra *case* mostrado é refinado na seguinte regra condicional.

if-then-else

```
1  if e = e1 then R1;
2  else  if e = e2 then R2;
3  . . .
4  else Rk;
```

Regra *if-then-else*

As regras do tipo *if-then-else* devem ser refinadas para regras condicionais do tipo *if-then* (sem *else*). Portanto, dada uma regra

if-then-else

```
1  if C then RLT; else RLF; end
```

onde *RLT* e *RLF* são listas de regras simples ou outra regra condicional, deve-se separar as duas partes gerando duas regras condicionais da seguinte forma:

if-then lista de regras

```
1  if     C then RLT; end
2  if not C then RLF; end
```

Posteriormente, cria-se uma regra *if-then* para cada $RT \in RLT$ e $RF \in RLF$. Supondo RLT com n regras e RLF com m, tem-se

if-then simples

```
1  if     C then RT_1; end
2  . . . . .
3  if     C then RT_n; end
4
5  if not C then RF_1; end
6  . . . . .
7  if not C then RF_m; end
```

Regras *if-then* aninhadas

Se existir alguma regra condicional em RT ou RF, deve-se eliminar o aninhamento fazendo a conjunção das condições, como mostra o exemplo

if-then aninhado

```
1  if C1 then
2    if C2 then R; end
3  end
```

if-then sem aninhamento

```
1  if C1 and C2 then
2    R;
3  end
```

4.5 Mapeamento de C-Machĭna a NuSMV

Devido à simplicidade de C-Machĭna, a transformação para NuSMV é quase direta. São necessários o refinamento dos tipos, apresentando uma nova estrutura e preservando as operações, e do refinamento da regra de transição, conforme apresentados a seguir.

4.5.1 Refinamento de tipos

Os tipos em Machĭna: real, *string*, lista, nodo árvore e abstração de regras, não foram refinados para C-Machĭna em nenhum dos tipos aqui apresentados. Este é um trabalho ainda a ser realizado. Outra possibilidade é preservá-los em C-Machĭna e apresentar um refinamento para NuSMV.

Os tipos a serem refinados são:

- **Int**;
- **Char**;
- **Bool**;
- **Intervalo**;
- **Enumeração**;
- **Agente**.

Alguns tipos são refinados em estruturas contendo seu valor e uma variável que indica o tipo, possibilitando a sua inspeção. Outros, que existem em NuSMV, são apenas representados na sintaxe correta. A criação desta estrutura é realizada por módulos que apenas possuem declarações de variáveis, portanto não executam nenhuma regra de transição e são compreendidos apenas como abstração de estruturas. O tipo agente, devido suas peculiaridades de formação a partir de outros módulos, é apresentado separadamente na Seção 4.5.2.

Os tipos **Int** e **Char** são refinados em termos de intervalos conforme a seguir. O intervalo de **Char** corresponde aos possíveis valores da tabela ASCII. Para **Int**, o intervalo é definido entre -2.147.483.648 e 2.147.483.647, conforme a definição de Machĭna [BTIB05, Capítulo 2].

Int

```
1  MODULE Int
2    VAR
3      value :  -2147483648 ..  2147483647;
4      type :  1;
```

Char

```
1  MODULE Char
2    VAR
3      value :  0 ..  255;
4      type :  2;
```

Os tipos booleano, intervalo e enumeração existem da mesma forma em NuSMV. O tipo **Bool** é representado pela estrutura da mesma forma que **Int** e **Char**. Os demais são representados com o mesmo tipo em NuSMV, sendo necessário apenas colocá-los na sintaxe correta.

Bool

```
1  MODULE Bool
2    VAR
3      value :  boolean;
4      type :  2;
```

Intervalo

```
1  variable :  inf ..  sup;
```

Enumeração

```
1  variable :  {e1, e2 ...  en};
```

4.5.2 Refinamento de agentes

O tipo agente é refinado em uma instância de módulo em NuSMV, porém a formação da álgebra do módulo deve ser considerada neste momento. Assim, cada módulo de C-Machĭna é transformado em um módulo em NuSMV.

Para a formação do vocabulário, é necessário tratar a inclusão de elementos públicos e compartilhados. No caso de elementos públicos, todo módulo que inclui *publicA* é acrescido da cláusula *ISA publicA*. O comando ISA de NuSMV copia as variáveis de um módulo em outro.

No caso dos elementos compartilhados, os módulos que incluem *sharedA* ou *publicSharedA* são criados com um parâmetro para cada função compartilhada existente nestes dois módulos, e isso se repete para todas as demais inclusões realizadas. Assim, os agentes dos módulos que incluem A, conforme modelado em Machĭna, compartilham

as funções *public shared* e apenas os agentes de A irão compartilhar as funções *shared* e *public shared*.

As variáveis criadas como parâmetro de módulos, representando as funções compartilhadas, terão um comportamento não determinístico a cada transição de estado. Assim, é possível simular o compartilhamento de informações. Deve-se ressaltar que, com essa abordagem, a condição de corrida não é verificada em NuSVM, devendo esta ser garantida previamente pelo desenvolvedor.

Considerando o exemplo em C-Machĭna dos módulos A e B a seguir, ambos incluem *publicA* e *publicSharedA*, e somente A inclui *sharedA*. Inicialmente em Machĭna, o módulo A tem as funções a pública, b compartilhada e pública, e c como compartilhada.

Módulos em C-Machĭna

```
1  module A
2    include publicA, publicSharedA, sharedA;
3  end A
4
5  module B
6    include publicA, publicSharedA;
7  end B
8
9  module publicA
10   a : Int;
11 end publicA
12
13 module publicSharedA
14   b : Int;
15 end publicSharedA
16
17 module sharedA
18   c : Int;
19 end sharedA
```

Pode-se, então, realizar o seguinte refinamento para NuSMV, sendo c uma variável compartilhada entre agentes de A e b compartilhada entre agentes de A e de B.

Módulos em NuSMV

```
1 MODULE A(b, c)
2   ISA publicA
3
4 MODULE B(b)
5   ISA publicA
6
```

```
7  MODULE publicA
8    VAR a :  Int;
9
10 MODULE publicSharedA
11   VAR b :  Int;
12
13 MODULE sharedA
14   VAR c :  Int;
```

Por fim, o módulo principal é criado com base na máquina de definição de C-Machĭna, que é a mesma de Machĭna. Então, para o código a seguir

Máquina de definição

```
1  machine mX
2    create agent of mX;
3  end mX
```

o módulo *main* em NuSMV é definido da seguinte forma. Criam-se os agentes que terão suas regras, bloco ASSIGN, disparados em execução.

Módulo principal

```
1  MODULE main
2    agent :  mX;
```

4.5.3 Refinamento da regra de transição

Primeiramente, inicializa-se cada variável de acordo com o seu estado inicial definido em C-Machĭna. Caso uma função não tenha sido inicializada, atribui-se em NuSMV seu valor padrão de acordo com o domínio. A exceção é no caso de uma função externa, onde nada é feito, assim pode-se obter o comportamento não determinístico. Então, para cada variável declarada em VAR, cria-se em ASSIGN o seguinte código.

Inicialização em NuSMV

```
1  ASSIGN
2    init(variable) := initialValue;
```

A regra de transição em C-Machĭna é formada de um conjunto de regras do tipo

$$if\ g\ then\ R$$

onde R é uma regra de atribuição simples, ou seja, apenas uma variável é atualizada de cada vez. Agrupam-se então todas as regras condicionais onde R possui o mesmo lado esquerdo e cria-se uma regra *case* em NuSVM com as respectivas guardas. Para evitar o comportamento não determinístico, que não é desejado, a última cláusula do *case* é construída com a guarda 1 (verdadeira) e atribui-se o mesmo valor. Assim, se nenhuma outra condição anterior for satisfeita, a variável não se altera. Às funções externas não são atribuídos valores em NuSMV para poder ter o efeito não determinístico. O exemplo a seguir mostra o refinamento de um suposto conjunto de regras onde aparecem as funções a e **b**.

C-Machĭna

```
1  . . .
2  transition:
3    if g1 then a := x; end
4    if g2 then a := y; end
5    if g3 then b := w; end
6    if g4 then b := z; end
```

Refinamento par NuSMV

```
1  . . .
2  ASSIGN
3    init(a) := 0; - valor padrão
4    init(b) := 2; - valor em initial state
5
6    next(a) :=
7      case
8        g1 :  x;
9        g2 :  y;
10        1 :  a; - preserva valor de a caso g1 e g2 sejam falsos
11     esac;
12
13   next(b) :=
14     case
15       g3 :  w;
16       g4 :  z;
17        1 :  b; - preserva valor de b caso g3 e g4 sejam falsos
18     esac;
```

Se existir alguma regra que indica pré-condição de uma ação (**require**), cria-se uma propriedade que verifica se sempre que a condição C ocorrer, a pré-condição deverá ser satisfeita. Assim, será possível verificar, via NuSMV, se a pré-condição de uma ação

é satisfeita no momento de sua chamada. Após a criação da propriedade, a regra é descartada e não vai para o bloco ASSIGN. Assim, dada a regra

Pré-condição de Ação

```
1  if C then require P; end
```

será criada a propriedade para NuSMV

Propriedade

```
1  SPEC AG(C -> P);
```

Para a cláusula de pós condição de uma ação, deve-se gerar a seguinte propriedade

Propriedade

```
1  LTLSPEC G(C -> X Q);
```

que significa sempre que a condição C for satisfeita, ou seja sempre que uma ação retornar de sua execução, em qualquer estado imediatamente seguinte, a condição Q deve ser satisfeita.

Para se provar que esta propriedade é válida, deve-se mostrar que Q será verdade no próximo estado. Caso a regra de transição contenha múltiplos passos e o passo seguinte seja um omisso, ou seja, passo não declarado explicitamente na regra de transição, nada é feito na álgebra além de alterar o valor de *step*. Então se Q for verdade no passo seguinte (o omisso), também será verdade no seguinte do omisso. Lembrando-se que a semântica de **step** com passos omissos é não fazer nada nestes passos e o usuário deve ter consciência disso.

Além disso, deve-se mostrar que uma transição de estado em Machĭna tem correspondência um para um em C-Machĭna, isso significa que não existem estados intermediários. Isso é fácil de ser percebido, uma vez que a planificação de regras apenas realiza a conjunção das condições existentes em regras *if-then-else* aninhadas e regras *case*. A simplificação de termos apenas substitui a regra de atualização por um conjunto de atualizações condicionadas correspondente. Em ambos os casos, não são criados estados intermediários, preservando a validade da propriedade acima.

4.5.4 Propriedades e invariantes

As propriedades e invariantes definidos no Modelo Básico e as propriedades extraídas das regras de refinamento são diretamente mapeados para os blocos correspondentes

em NuSMV. Para propriedades, é preciso analisar sua natureza para definir em qual definição será incluída, SPEC, LTLSPEC ou PSLSPEC. Para os invariantes, o mapeamento é direto para cada expressão, precedida de INVAR. Completa-se assim a especificação NuSMV do Modelo Básico.

4.6 Conclusões

Foi mostrado o refinamento de Machĭna para NuSMV, obtendo a entrada a ser verificada, onde conferem-se as propriedades definidas de acordo com o Modelo Básico, representado pela especificação NuSMV obtida. O resultado final pode apresentar um sistema correto, ou então, falha em uma ou mais propriedades. Neste segundo caso, se for possível, mostra-se um exemplo de execução para cada propriedade não garantida. Assim, a localização de erros se torna uma tarefa mais fácil.

As correções são realizadas no Modelo Básico e deve ser realizado todo o processo de refinamento até ser novamente possível realizar a verificação. Como o processo é automático e todas as informações necessárias estão no Modelo Básico, o código executável e a verificação podem ser sempre obtidos a partir do Método de Refinamento Machĭna. Além disso, os dados fornecidos pela execução NuSMV fazem parte da documentação, servindo como comprovante da correta especificação e podem ser utilizados para futuras análises do sistema.

Verificação de Modelos (*Model Checking*) é um método que permite verificar automaticamente propriedades de um sistema. O verificador NuSMV utiliza algoritmos simbólicos baseados em OBDD (*Ordered Binary Decision Diagram*) e descreve uma linguagem para especificação de sistemas de transição de estados onde os estados são finitos. Uma característica notável do método Verificação de Modelos é a facilidade de lidar com o grande número de estados possíveis que um sistema pode alcançar.

Em NuSVM, as propriedades de um sistema podem ser definidas em fórmulas CTL (*Computational Tree Logic*) ou LTL (*Logical Temporal Logic*). A vantagem é poder realizar a verificação não pelo acompanhamento dos estados, mas sim por perguntas que são feitas ao modelo. Caso uma das propriedades falhe, NuSMV pode fornecer um contra-exemplo, provendo assim uma possibilidade de depuração.

Para todas as construções de Machĭna, com exceção de ações com *loop*, foi possível obter uma representação equivalente em NuSMV, incluindo funções de controles implícitas à linguagem, como a função *state* e *step*. Foi possível também preservar a modularidade e comunicação entre os agentes, assim como a execução assíncrona. Porém, para alguns tipos de Machĭna, como listas e tipo abstração de regras, não conseguiu-se encontrar uma representação em NuSMV. Com a abordagem adotada

para funções compartilhadas, não é possível verificar a condição de corrida, ficando a cargo do desenvolvedor.

A primeira dificuldade no refinamento para NuSMV foi solucionar o problema de funções com aridade maior que zero, onde procurou-se algumas alternativas diferentes das apresentadas por Kirsten [Win01], porém nenhuma atingiu um resultado satisfatório. Então, prevaleceu a mesma solução apresentada por ela, com o mesmo problema de gerar explosão de estados no BDD ao utilizar a regra de restrição de ajuste. Embora um verificador de modelos seja capaz de tratar explosão de estados de forma eficiente, acredita-se que alguns dos problemas encontrados durante a avaliação, Capítulo 5, podem estar ligados à explosão de estados gerada.

Outra dificuldade foi modelar a comunicação e execução entre os agentes, principalmente quanto a variáveis compartilhadas. Machĭna define a execução de agentes como sendo assíncrona e paralela, enquanto em NuSMV estas duas são possíveis mas não ao mesmo tempo. A melhor alternativa foi a execução assíncrona por dois motivos: primeiro, de acordo com Börger [BS03], nenhum ambiente ASM apresenta execução assíncrona, sendo assim, Machĭna é o único ambiente de execução ASM síncrono e assíncrono, e pode ser verificado; segundo, o prejuízo seria minimizado, pois a única perda seria a verificação de acessos concorrentes a variáveis compartilhadas, já que as demais comunicações são realizadas via trocas de mensagens e Machĭna é definida com características modulares. Ao realizar a avaliação, mostrada no Capítulo 5, percebeu-se que somente seria possível obter uma resposta da verificação se os módulos fossem verificados separadamente, simulando a comunicação. E essa simulação somente foi possível devido à escolha de execução assíncrona em NuSMV.

O refinamento de Machĭna para NuSMV tem como benefício prover verificação semi-automática ao Método de Refinamento Machĭna. Como as regras são aplicadas automaticamente, o processo de verificação em MRM tende a ser transparente para o desenvolvedor. Outro benefício foi a definição de uma linguagem intermediária, Core-Machĭna(C-Machĭna), que é mais simples que Machĭna. Assim, trabalhos de desenvolvimento de compiladores e expansão da linguagem podem ser mais facilmente realizados.

O resultado obtido com a execução de NuSMV provê um recurso importante para depurar a especificação descrita em MRM. Um possível contra-exemplo pode ser utilizado como entrada de dados para simular uma execução no Modelo Básico e facilitar a identificação do problema. Este ainda é um processo manual, mas trabalhos futuros podem ser realizados para tornar esta tarefa mais interativa e automática.

Capítulo 5

Avaliação do Método Proposto

Para fins de avaliação do Método de Refinamento Machĭna, apresenta-se neste capítulo a especificação de um sistema para controle de uma caldeira a vapor. O objetivo é avaliar a Linguagem de Modelagem Machĭna quanto ao poder de expressão, legibilidade e separação dos níveis de abstração. São modeladas características como inclusão de novos comportamentos pelo Controle Global, comunicação entre agentes, definição de propriedades do sistema e não determinismo. Quanto à aplicação de regras e verificação automática, será possível avaliar a obtenção automática de propriedades, transparência para o desenvolvedor e resultados da verificação.

5.1 O Problema da Caldeira a Vapor

O problema *Steam Boiler* [Abr96b], tratado neste capítulo, foi descrito por Jean-Raymond Abrial em agosto de 1994 como sugestão de trabalho para os participantes do *Dagstuhl Meeting Methods for Semantics and Specification*, que foi realizado em Junho de 1995. Foi especificado o funcionamento de um programa de controle de nível de água dentro de uma caldeira a vapor. Os participantes deveriam apresentar uma definição em alto nível e o correto refinamento para um código executável.

Uma caldeira deve trabalhar com o nível de água dentro de limites considerados normais de funcionamento e nunca ultrapassar os níveis de risco definidos. Um problema com relação a esses fatos pode causar sérios danos à caldeira. Abrial descreve inicialmente o ambiente físico existente no qual o programa deve interagir. Cada unidade física é apresentada, assim como suas restrições e condições de funcionamento. São definidas algumas constantes e variáveis que devem ser verificadas e controladas ao longo do funcionamento do sistema.

Em seguida, Abrial apresenta, de forma geral, como o programa se comunica com as unidades físicas e quais são as fases de comunicação. Posteriormente, são detalhados

os cinco modos de operação que o programa pode assumir. A especificação determina os tipos de mensagens enviadas e recebidas pelo programa e, finalmente, descreve a detecção de erros nos equipamentos.

Em [ABL96, ABL95], foram apresentadas 23 soluções abordando diferentes métodos de refinamento. Uma delas foi proposta por Börger [BBD+96], onde foi apresentado um modelo em alto nível (Modelo Básico) fundamentado na Álgebra Evolutiva. Como linguagem para o Modelo Básico foi utilizado o modelo ASM e apresentou-se o refinamento até a obtenção do código executável em C++.

Este capítulo apresenta a definição do Modelo Básico, escrito em Linguagem de Modelagem Machĭna, para o problema da caldeira a vapor. Então, aplica-se o Método de Refinamento Machĭna obtendo o código Machĭna correspondente e realizando a verificação do sistema. A especificação de Abrial permite, em alguns momentos, divergências nas interpretações. Para melhor validação e apresentação do Método de Refinamento Machĭna, muitas das decisões foram tomadas seguindo a solução apresentada em [BBD+96], já que ambos são derivados do modelo ASM de refinamento.

5.2 Unidades Físicas

O programa de controle da caldeira a vapor deve se relacionar com algumas unidades físicas, que foram listadas como sendo:

- caldeira a vapor;
- unidade medidora de água;
- unidade medidora de vapor;
- quatro bombas de água para a caldeira;
- quatro controladores de bomba (um para cada bomba);
- operador de mesa;
- sistema de mensagem.

O operador de mesa não faz parte dessa modelagem, mas são consideradas funções provindas do ambiente externo que são acionadas pelo operador de mesa. O sistema de mensagem, como será visto, é modelado com a comunicação entre agentes realizadas por serviços disponibilizados. Portanto, erros de mensagens não são considerados nesta definição.

A caldeira a vapor é constituída de uma válvula para liberação de água, que é utilizada para esvaziar a caldeira apenas na fase inicial. A capacidade total de água C é indicada em litros, e M_1 e M_2 são os limites inferiores e superiores, respectivamente, também em litros. Se o nível de água estiver abaixo de M_1 com liberação de vapor sem o devido suprimento de água, ou se o nível estiver acima de M_2 com as bombas fornecendo água e pouca liberação de vapor, a caldeira pode estar em perigo após cinco segundos. São estipulados os níveis inferior N_1 e superior N_2 para o funcionamento normal, que devem ser mantidos durante o funcionamento. Devem-se preservar as condições $M_1 < N_1$ e $M_2 > N_2$. A quantidade máxima W de vapor saindo da caldeira é medida em litros/seg e o gradiente de aumento U_1 e diminuição U_2 de vapor é dado em litros/seg/seg. A Tabela 5.1 resume as variáveis e constantes do sistema de controle da caldeira a vapor.

	Unidade	**Comentários**
		Quantidade de água na caldeira
C	litros	Capacidade máxima
M1	litros	Limite mínimo de risco
M2	litros	Limite máximo de risco
N1	litros	Limite mínimo normal
N2	litros	Limite máximo normal
		Liberação de vapor
W	litros/seg	Quantidade máxima
U1	litros/seg/seg	Gradiente máximo de aumento
U2	litros/seg/seg	Gradiente máximo de diminuição
		Valores correntes
q	litros	Quantidade de água na caldeira
v	litros/seg	Quantidade de vapor dissipado

Tabela 5.1: Resumo de variáveis e constantes.

A caldeira a vapor é formada por estas unidades físicas que possuem comunicação direta com o sistema de controle. Portanto, para fins de simplicidade, a modelagem da caldeira não é explícita, e quando for necessário a mesma é representada por uma função no sistema de controle da caldeira a vapor. A seguir, são apresentados cada unidade física, o controlador para unidades físicas e o programa controlador da caldeira a vapor.

5.2.1 Unidades de Medida

A unidade medidora de água (*water level measuring unit - wm*) mede a quantidade q em litros de água na caldeira em um determinado instante, assim como a unidade medidora de vapor (*steam level measuring unit - sm*) mede a quantidade u de vapor

que sai da caldeira em litros/seg. Estas duas unidades podem ser modeladas de forma semelhante. Assim, o Tipo Abstrato de Dados apresentado em Figura 5.1 mostra o identificador *id* para diferenciar as unidades de medidas e o valor lido *measure*.

Na interface, apresentam-se os serviços *SetId*, para atribuir o correto identificador, e *SetSBC*, para indicar o sistema de controle da caldeira. *GetError* provê em *e* a situação de erro da unidade. Por fim, o pedido para a unidade parar seu funcionamento pode ser realizado com *Stop*.

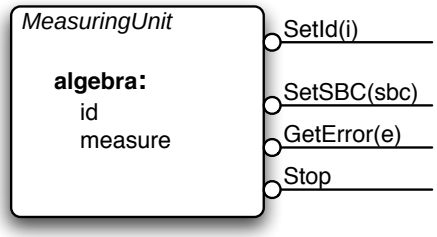
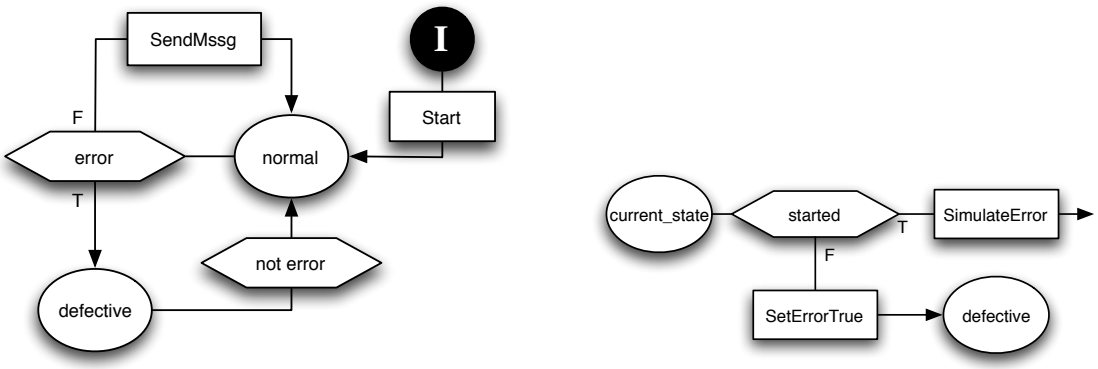


Figura 5.1: TAD *Measuring Unit*.

Cada unidade, em um funcionamento normal, manda constantemente uma mensagem para o programa de controle contendo o valor q, ou v, lido. Na presença de algum erro, a unidade fica aguardando o reparo. Este comportamento pode ser conferido na Figura 5.2(a)

Por questões de simplicidade, erros nestas unidades são simulados. Como um erro pode acontecer a qualquer momento, esta simulação é realizada no Controle Global, conforme Figura 5.2(b). Além disso, é realizada a verificação se todas as funções de *Measuring Unit* foram corretamente inicializadas, caso falso, a unidade permanece no estado de defeito.



(a) DTE *Measuring Unit*. (b) Controle Global *Measuring Unit*.

Figura 5.2: *Measuring Unit*

Além das duas funções que aparecem no TAD de *Measuring Unit*, existem outras

duas, *error*, para controle de erro, e *system* que referencia o sistema de controle da caldeira. Por fim, define-se a função que verifica se as demais foram corretamente inicializadas. Assim, a álgebra pode ser definida conforme a seguir.

Glossário : Álgebra

```
1  id      : SBC.MeasuringId;
2  measure : Int;
3  error   : Bool;
4  system  : SBC;
5  derived started : Bool := system != undef and id > 0;
```

A função *started* poderia descrever um invariante no módulo *MeasuringUnit*, mas isso poderia provocar a interrupção do programa em caso de *started* ser falso. Tratada no Controle Global, a verificação de *system* e *id* é realizada com o objetivo de contornar o problema durante a execução do sistema.

A ação *Start* tem o objetivo de inicializar as funções *measure* e *error* da unidade medidora. As funções *id* e *system* devem ser configuradas antes da execução da regra de transição, utilizando os serviços *SetId* e *SetSBC*. Como foi mostrado no Controle Global, a inicialização incorreta destas funções coloca a unidade medidora em estado de defeito. A ação *SendMssg* envia para o programa de controle da caldeira, *system*, a informação lida e qual é a unidade, passando o seu *id*. A simulação de erro é feita atribuindo-se um valor, verdadeiro ou falso, não deterministicamente à função *error*. *GetError* é um serviço que provê a informação em *e* da função *error*. Por fim, *Stop* interrompe a execução da unidade. O Glossário a seguir apresenta estas definições.

Glossário : Abstrações

```
1  action Start is
2    measure := 0;
3    error   := false;
4  end
5
6  action SendMssg is
7    system.RecieveMeasuring(id, measure);
8  end
9
10 action SimulateError is
11   choose e : Bool
12   do error := e;
13   end
14 end
15
```

```
16    action SetErrorTrue is
17      error := true;
18    end
19
20    public action SetId(in i : SBC.MeasuringId) is
21      id := i;
22    end
23
24    public action SetSBC(in sbc : SBC) is
25      system := sbc;
26    end
27
28    public action GetError(out e : Bool) is
29      e := error;
30    end
31
32    public action Stop is
33      stop;
34    end
```

5.2.2 Válvula

A válvula, módulo *Valve*, da caldeira a vapor tem comportamento semelhante das unidades de medidas. O TAD para a válvula é apresentado na Figura 5.3 e mostra a função *opened*, que indica se a válvula está ou não aberta, e os serviços que disponibiliza, sendo os pedidos para abrir e fechar, a referência para o sistema de controle da caldeira, obtenção do estado de erro e o pedido de interrupção da execução da válvula. Além disso, define os serviços para abrir e fechar a válvula, correspondendo às mensagens enviadas pelo programa de controle.

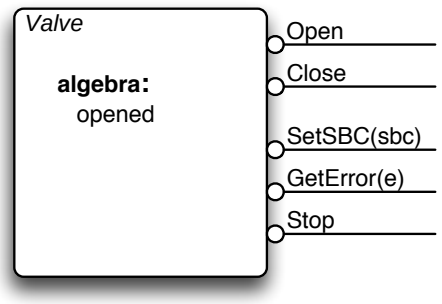
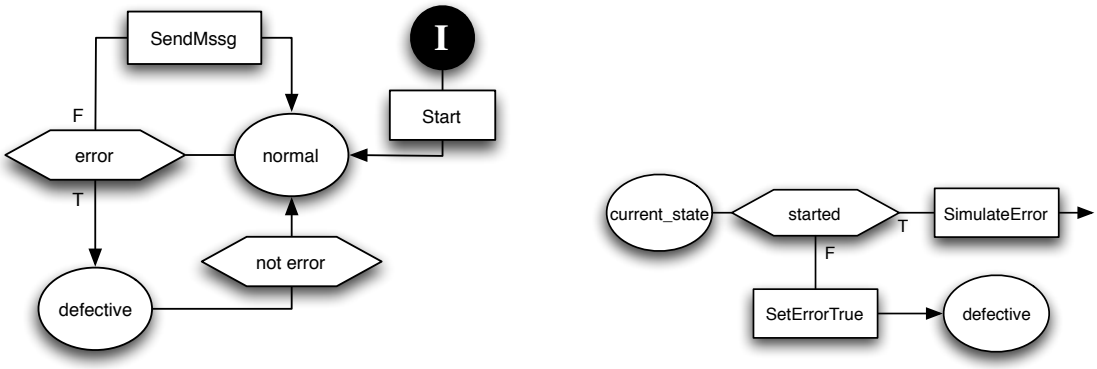


Figura 5.3: TAD *Valve*.

Como pode-se verificar nas Figuras 5.4(a) e 5.4(b), a válvula tem o comportamento semelhante às unidades de medida. A diferença está na álgebra e na mensagem enviada

ao sistema, que no caso da válvula, envia a informação se está aberta ou fechada. Assim, são definidas a Álgebra e Abstrações para *Valve*.



(a) DTE *Valve*. (b) Controle Global *Valve*.

Figura 5.4: *Valve*

Glossário : Álgebra

```
1  opened : Bool;
2  error  : Bool;
3  system : SBC;
4  derived started : Bool := system != undef;
```

Glossário : Abstrações

```
1  action Start is
2    opened := false;
3    error  := false;
4  end
5
6  action SendMssg is
7    system.RecieveValve(opened);
8  end
9
10 action SimulateError is
11   choose e : Bool
12   do error := e;
13   end
14 end
15
16 action SetErrorTrue is
17   error := true;
18 end
```

```
19
20    public action Open is
21      opened := true;
22    end
23
24    public action Close is
25      opened := false;
26    end
27
28    public action SetSBC(in sbc : SBC) is
29      system := sbc;
30    end
31
32    public action GetError(out e : Bool) is
33      e := error;
34    end
35
36    public action Stop is
37      stop;
38    end
```

5.2.3 Bombas

Cada uma das quatro bombas de água da caldeira possui sua capacidade P medida em litros/seg e a vazão corrente p também em litros/seg. O controlador *ctrl* de cada bomba indica se existe ou não circulação de água no sentido bomba-caldeira. Além dos tradicionais serviços de identificação *SetSBC* do sistema, obtenção do estado de erro *GerError* e requisição de parada *Stop*, são disponibilizados os serviços de ligar e desligar a bomba, *TurnOn* e *TurnOff*, respectivamente. Todas essas informações podem ser conferidas no TAD de *Pump* representado na Figura 5.5.

Uma bomba contém os modos de funcionamento ligado *on* e desligado *off*. Caso exista algum erro, a bomba entra em estado defeituoso, *defevtive*. Quando inicializada, a bomba gasta em torno de cinco segundos para começar a prover água à caldeira, e quando ela é desligada, o efeito é instantâneo. Este efeito é representado no DTE-pump, mostrado na Figura 5.6, pela verificação do tempo, *fiveSeconds*, antes de simular a passagem de água da bomba para a caldeira, reconhecimento realizado pelo controlador *ctrl*. Enquanto a bomba está ligada, estado *on*, ela realiza a leitura da vazão. Isso é modelado pela atividade deste estado *ReadP*. Os estados desligado, *off*, e defeituoso, *defectvie*, possuem a propriedade *pumpOff*, que será definida posteriormente.

De forma semelhante às outras unidades físicas, deve-se verificar se a bomba foi

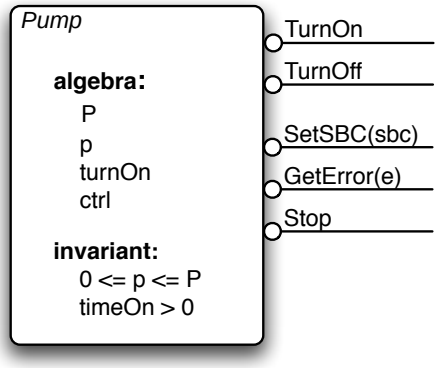


Figura 5.5: TAD *Pump*

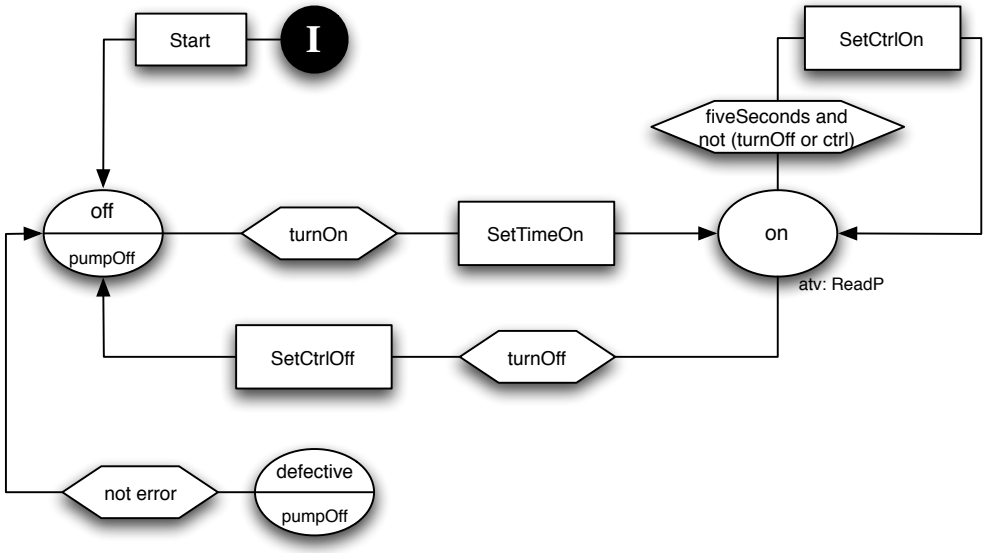


Figura 5.6: DTE *Pump*

corretamente inicializada. No funcionamento normal, a bomba envia constantemente seus dados para o sistema de controle da caldeira a vapor. Caso exista algum erro, o funcionamento da bomba é imediatamente interrompido e ela é desligada. Estas condições e operações são mostradas no Controle Global, Figura 5.7, pois devem ser sempre conferidas e executadas.

A seguir, define-se a álgebra para *Pump* considerando as funções do TAD e *timeOn* como indicador do momento que a bomba foi ligada. A condição *fiveSeconds*, presente no DTE, verifica se já se passaram cinco segundos desde que a bomba foi ligada. A função *turnOn* é um indicativo de que houve uma requisição para que se ligasse a bomba e *turnOff* para que desligasse.

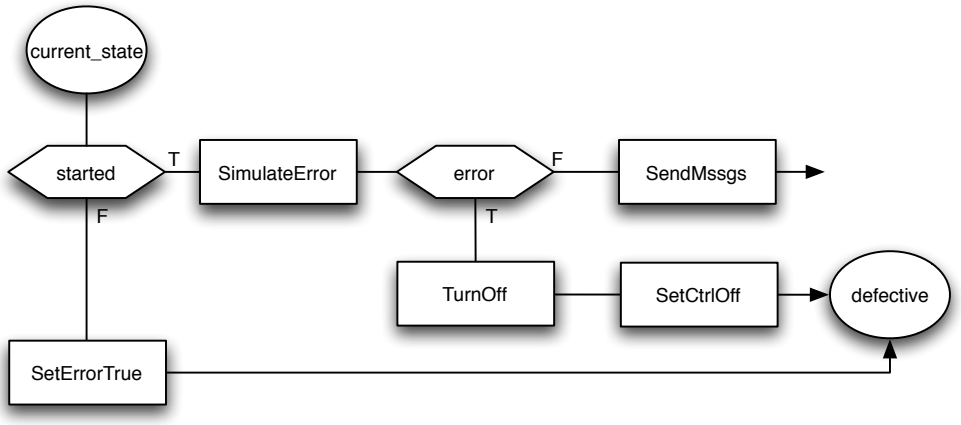


Figura 5.7: Controle Global *Pump*

Glossário : Álgebra

```
1  P : Int;      // capacidade total
2  p : Int;      // vazão corrente
3  ctrl : Bool; // controlador
4
5  derived pumpOff : Bool := p = 0 and not ctrl;
6
7  turnOn : Int;
8  derived turnOff : Bool := not turnOn;
9
10 timeOn : Bool;
11 derived fiveSeconds : Int := (Timer.time - timeOn) > 5;
12
13 error  : Bool;
14 system : SBC;
15 derived started : Bool := system != undef and P > 0;
```

Além das definição dos elementos do TAD e DTE é possível notar a definição da propriedade *pumpOff* na linha 5. Esta condição, definida como propriedade nos estados *off* e *defective*, indica que se a bomba não está ligada, a vazão corrente deve ser zero e o controlador falso, indicando a não passagem de água na bomba.

A ação *Start* inicializa as funções de *Pump*, considerando *system* previamente configurada como referência para o sistema de controle da caldeira. *SetTimeOn* marca o tempo em que a solicitação para ligar a bomba ocorreu, possibilitando simular o atraso de cinco segundos para a bomba começar a jogar água na caldeira.

Após os cinco segundos, a bomba inicia o despejo de água e quando for solicitado o desligamento, o processo deve ser inverso. Para isso existem as ações *SetCtrlOn* e *SetCtrlOff*. Estando ligada, a atividade *ReadP* simula a leitura da vazão corrente,

desde que exista água passando pela bomba. Duas informações são enviadas para o sistema, uma é a vazão corrente e a outra é a situação do controlador, indicando se existe ou não fluxo de água. Então, *SendMssgs* envia as mensagens *SendPumpState* e *SendPumpCtrl*. Assim como nos casos anteriores, o erro é simulado em *SimulateError*. A seguir, apresentam-se abstrações contendo estas definições e as definições dos serviços disponibilizados por *Pump*.

Glossário : Abstrações

```
1  action Start is
2    p      := 0;
3    ctrl   := false;
4    turnOn := false;
5    error  := false;
6    timeOn := Timer.time;
7  end
8
9  action SetTimeOn is
10   timeOn := Timer.time;
11 end
12
13 action ReadP is
14   choose i : Int
15   satisfying ctrl and i > 0 and i < P;
16   do p := i;
17 end
18
19 action SetErrorTrue is
20   error := true;
21 end
22
23 action SendMssgs is
24   SendPumpState;
25   SendPumpCtrl;
26 end
27
28 action SendPumpState is
29   if current_state = on then
30     system.RecievePumpState(self,true);
31   else
32     system.RecievePumpState(self,false);
33   end
34 end
35
36
```

```
37  action SendPumpCtrl is
38    system.RecievePumpCtrl(self,ctrl);
39  end
40
41
42  public action TurnOn is
43    turnOn := true;
44  end
45
46  public action TurnOff is
47    turnOn := false;
48  end
49
50  public action SetSBC(in sbc : SBC) is
51    system := sbc;
52  end
53
54  public action GetError(out e : Bool) is
55    e := error;
56  end
57
58  public action Stop is
59    stop;
60  end
```

Como propriedades de uma bomba, podem ser definidas as seguintes:

- se o controlador indicar que não existe fluxo de água na bomba, a vazão corrente deve ser zero;
- se um erro for encontrado, o estado interno será, em algum momento no futuro, *defective*.

Glossário : Propriedades

```
1  AG not ctrl -> p = 0;
2  AG error -> AF current_state = defective;
```

5.3 Controlador de unidade física

Embora não tenha sido mencionado na especificação de Abrial, foi criado um controlador para cada unidade física, de forma a lidar com envios e reconhecimentos das

mensagens de erro. Assim, é possível associar a cada unidade física, além de seus estados naturais, os estados de funcionamento *regular*, *defective*, e *aknowledged*, conforme foi originalmente descrito por Abrial.

O TAD da Figura 5.8 mostra que cada controlador tem uma unidade *unit* associada e provê os serviços de recebimento do modo de execução do sistema de controle da caldeira, *RecieveMode*, e da mensagem informando que este está pronto, *RecieveProgReady*, estando de acordo com a especificação de Abrial. Além disso, os serviços *SetUnit* e *SetSBC* configuram qual a unidade e o sistema correspondentes.

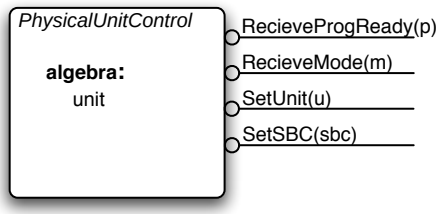


Figura 5.8: TAD *PhisicalUnitControl*

Considerando a associação do sistema e da unidade nas respectivas referências já realizadas, o controlador aguarda pela mensagem informando que o programa está pronto para execução. Assim feito, inicia-se o funcionamento da unidade e retorna a mensagem de que a unidade está ponta. No estado *regular*, verifica-se a todo instante se existe algum erro em *unit*, caso exista, envia-se continuamente uma mensagem reportando a detecção do erro até que se obtenha uma resposta de reconhecimento do problema pelo programa. O estado então é de erro reconhecido, *aknowledged*, e aguarda-se até o reparo da unidade. Quando isso ocorrer, é enviada uma mensagem ao programa indicando o reparo e a unidade volta ao estado normal de funcionamento. Tudo isso é mostrado no DTE da Figura 5.9

O Controle Global na Figura 5.10 modela a verificação que avalia se o sistema de controle da caldeira está pronto, desde que o estado não seja *started*, onde aguarda-se a mensagem do sistema. Caso o programa não esteja pronto, a unidade é interrompida.

A Álgebra a seguir mostra a definição de *unit* em TAD e demais elementos presentes no DTE e Controle Global. A função *startedState* corresponde a verificar se o estado interno do DTE é *started*.

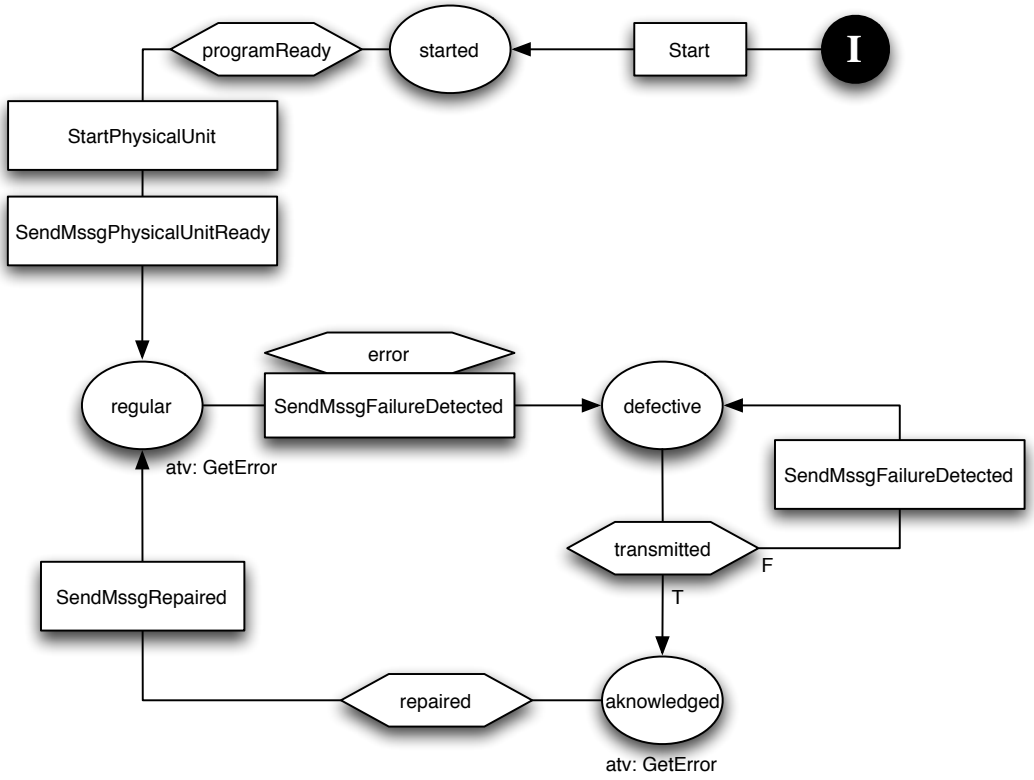


Figura 5.9: DTE *PhysicalUnitControl*.

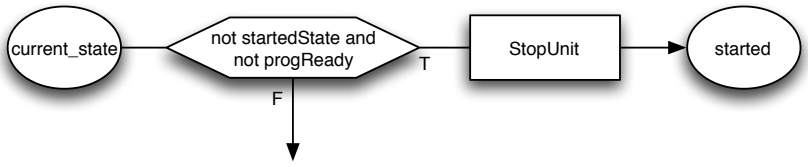


Figura 5.10: Controle Global *PhisicalUnitControl*.

Glossário : Álgebra

```
1  unit : PhysicalUnit;
2
3  transmitted  : Bool;
4  programReady : Bool;
5
6  error  : Bool;
7  system : SBC;
8
9  derived repaired : Bool := not error;
10 systemMode       : SBC.InternalState;
11
12 derived startedState : Bool := current_state = started;
```

As ações *SendMssgPhysicalUnitReady* e *SendMssgRepaired* indicam para o sistema que a determinada unidade física está pronta e que foi reparada, respectivamente, enquanto a ação *SendMssgFailureDetected* avisa que a unidade tem um erro e aguarda a confirmação de envio. O serviço *RecieveMssgProgReady* recebe a informação de que o programa está pronto e *RecieveMssgMode* o modo de operação do sistema.

Toda vez que o controlador consulta a função *error* de uma unidade física, deve-se assinalar que esta nova consulta ainda não foi enviada ao sistema, conforme mostrado na ação *GerError*, na linha 15. Estas e as demais ações são definidas a seguir em Abstrações.

Glossário : Abstrações

```
1  action Start is
2    error        := false;
3    transmitted  := false;
4    programReady := false;
5  end
6
7  action StartUnit is
8    dispatch unit;
9  end
10
11 action StopUnit is
12   unit.Stop;
13 end
14
15 action GetError is
16   unit.GetError(error);
17   transmitted := false;
18 end
```

```
19  action SendMssgPhysicalUnitReady is
20    system.RecievePhysicalUnitReady(unit);
21  end
22
23  action SendMssgFailureDetected is
24    system.RecieveFailureDetected(unit,transmitted);
25  end
26
27  action SendMssgRepaired is
28    system.RecieveRepaired(unit);
29  end
30
31  public action RecieveProgReady(in p : Bool) is
32    progReady := p;
33  end
34
35  public action RecieveMode(in m : SBC.InternalState) is
36    systemMode := m;
37  end
38
39  public action SetUnit(in u : PhysicalUnit) is
40    unit := u;
41  end
42
43  public action SetSBC(in sbc : SBC) is
44    system := sbc;
45  end
```

Uma vez que o programa de controle não está pronto, o controlador deve interromper o funcionamento da unidade física e se manter no estado *started*. Esta é uma propriedade que pode ser definida da seguinte forma.

Glossário : Propriedades

```
1  AG(!progReady -> AF (current_state = started));
```

5.4 Sistema de Controle da Caldeira a Vapor (SBC)

O sistema de controle da caldeira (*Steam Boiler Control - SBC*) contém as variáveis conforme listadas na Tabela 5.1, página 169 e recebe as mensagens como listado na interface do TAD de *SBC*, Figura 5.11. Em seguida, são apresentadas as definições das funções do TAD de *SBC* e serviços de sua interface.

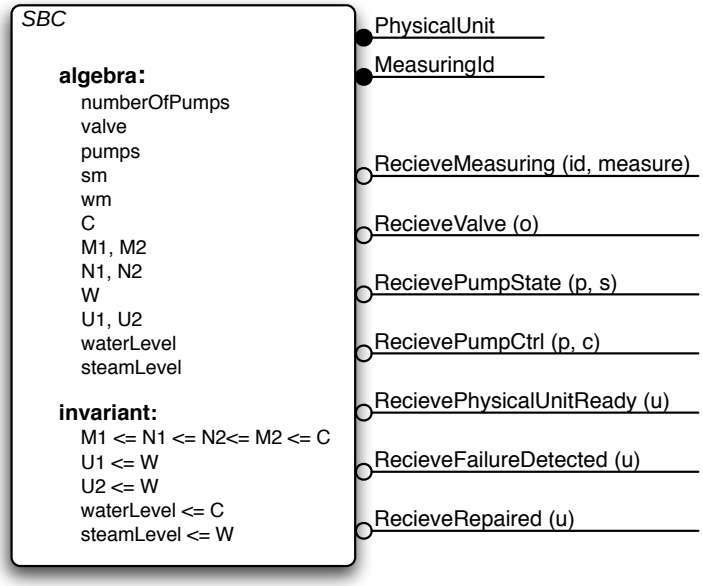


Figura 5.11: TAD *SBC*.

Glossário : Álgebra

```
1  public type PhysicalUnit : MeasuringUnit | Valve | Pump;
2  public type MeasuringId  : public enum {waterM, steamM};
3  numberOfPumps     : Int
4
5  derived numberOfUnits    : Int := (numberOfPumps + 3)
6
7   // Unidades físicas
8  valve : agent of Valve;           // válvula
9  pumps : agent of Pump;            // bombas na caldeira
10 sm    : agent of MeasuringUnit;   // steam measuring
11 wm    : agent of MeasuringUnit;   // water measuring
12
13  // Controladores de unidades física
14 physicalUnitControl : agent of PhisicalUnitControl;
15
16  // Constantes
17 C            : Int;        // capacidade total
18 M1, M2       : Int;        // níveis críticos
19 N1, N2       : Int;        // níveis normais
20 W            : Int;        // quantidade máxima de vapor
21 U1, U2       : Int;        // gradientes de vapor
22 waterLevel   : Int;        // nível de água (q)
23 steamLevel   : Int;        // nível de vapor (v)
```

Glossário : Abstrações

```
1  public action RecieveMeasuring(in id : MeasuringId,
2                                 in measure : Int) is
3    if     id = waterM then waterLevel := measure;
4    elseif id = steamM then steamLevel := measure;
5    end
6  end
7
8  public action RecieveValve(in o : Bool) is
9    valveOpened := o;
10 end
11
12 public action RecievePumpState(in p : Pump, in s : Bool) is
13   isPumpOn(p) := s;
14 end
15
16 public action RecievePumpCtrl(in p : Pump, in c : Bool) is
17   pumpCtrl(p) := c;
18 end
19
20 public action RecievePhysicalUnitReady(in u : PhysicalUnit) is
21   ready(u) := true;
22 end
23
24 public action RecieveFailureDetected(in u : PhysicalUnit) is
25   failure(u) := true;
26 end
27
28 public action RecieveRepaired(in u : PhysicalUnit) is
29   failure(u) := false;
30 end
```

Como pode ser visto, os serviços são responsáveis por alterar os valores de algumas funções de *SBC*, que são definidas a seguir.

Glossário : Álgebra

```
1  ready       : PhysicalUnit -> Bool;
2  failure     : PhysicalUnit -> Bool;
3  valveOpened : Bool;
4
5  pumpCtrl    : Pump -> Bool;
6  isPumpOn    : Pump -> Bool;
7  derived isPumpOff(p : Pump) : Bool := not isPumpOn(p);
```

O sistema *SBC* opera em cinco diferentes modos de operação: inicialização, normal, degradado, recuperação e emergência. A seguir, apresenta-se cada um deles em termos de seus Diagramas de Saída de Estado e o Controle Global que é realizado. Ao final, mostram-se as propriedades para o módulo *SBC*. O modo de operação *stopped* não possui DSE por ser um estado final e é apresentado com sua atividade de finalização no DTE completo.

5.4.1 Controle Global

Constantemente, verifica-se se o nível de água na caldeira está dentro dos limites normais. Uma vez que essa condição seja desrespeitada, o programa tenta ajustar o nível, caso contrário, tomam-se as providências para manter o nível da água. Deve-se ressaltar que mensagens à válvula somente devem ser enviadas no estado inicial e esta deve permanecer fechada nos demais modos de operação. Além disso, a todo momento o sistema de controle envia às unidades físicas qual é o estado corrente. Neste modelo, a mensagem é enviada para o controlador de cada unidade.

Quando se identifica um erro grave, por meio da função *emergencyStop*, o sistema deve ser imediatamente transferido para o estado de emergência *stopped*, e aguardar que o ambiente externo tome as providências necessárias. Dessa forma, descreve-se o diagrama, Figura 5.12, para Controle Global de *SBC*. Abstrações no Glossário a seguir definem as ações conforme a descrição apresentada, a saber:

- *SendMssgMode*: envia a todos os controladores de unidade física, *physicalUnitCtrl*, o estado corrente;
- *RetainWaterLevel*: desliga todas as bombas e, se estiver no estado inicial, fecha a válvula;
- *AdjustWaterLevel*: se o nível de água, *waterLevel*, estiver abaixo no nível inferior normal, *N1*, é dado ordem para aumentar o nível de água, caso contrário, diminui-se o nível;
- *StopProgram*: toma as providências necessárias para finalizar corretamente o programa: envia mensagens a todas os controladores de unidade física.

Glossário : Abstrações

```
1  action SendMssgMode is
2    forall c : physicalUnitControl
3    do c.RecieveMode(current_state);
4    end
5  end
6
7  action RetainWaterLevel is
8    StopPumps;
9    if current_state = initialMode then CloseValve; end
10 end
11
12 action AdjustWaterLevel is
13   if waterLevel < N1
14   then RaiseWaterLevel;
15   else ReduceWaterLevel;
16   end
17 end
18
19 action StopProgram is
20   forall c : physicalUnitControl
21   do c.RecieveProgReady(false);
22   end
23 end
```

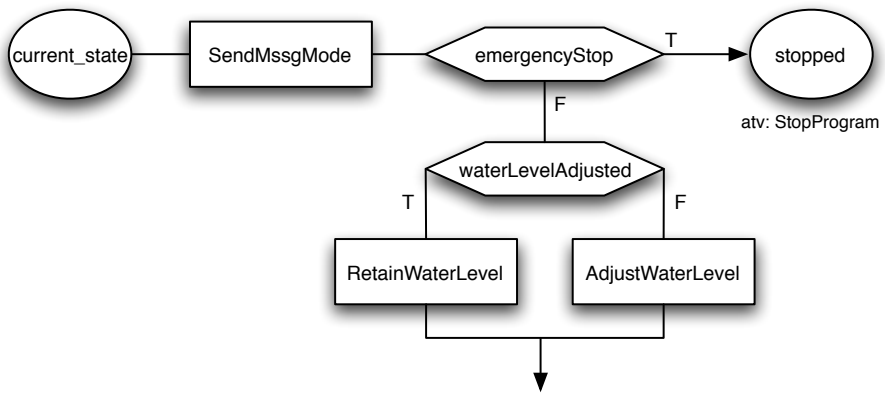


Figura 5.12: Controle Global *SBC*

As definições de ações do Controle Global apresentadas sugerem novas abstrações, que são então definidas a seguir, a saber:

- *StopPumps*: envia mensagem para todas as bombas, *pumps*, solicitando seu desligamento;

- *CloseValve*: mensagem enviada à válvula *valve*, no estado inicial, para que esta seja fechada;
- *OpenValve*: menagem enviada à válvula *valve*, no estado inicial, para que esta seja aberta;
- *RaiseWaterLevel*: toma as devidas providências para que o nível de água suba, ativando algumas bombas e, se estiver no estado inicial, fecha a válvula;
- *ReduceWaterLevel*: toma as devidas providências para que o nível de água seja reduzido, fechando algumas bombas e, se estiver no estado inicial, abre a válvula.

Glossário : Abstrações

```
1  action StopPumps is
2    forall p : pumps
3    do p.TurnOff;
4    end
5  end
6
7  action CloseValve is
8    valve.Close;
9  end
10
11 action OpenValve is
12   valve.Open;
13 end
14
15 action RaiseWaterLevel is
16   ActiveSomePumps;
17   if current_state = initialMode then CloseValve; end
18 end
19
20 action ReduceWaterLevel is
21   StopSomePumps;
22   if current_state = initialMode then OpenValve; end
23 end
```

Duas outras novas ações aparecem. São referentes ao envio de mensagem para algumas bombas com o objetivo de que sejam ligadas ou desligadas. Nesta modelagem, escolhe-se não deterministicamente uma bomba que esteja desligada, ou ligada, para executar a operação correspondente. A função *isPumpOn* mapeia uma bomba em seu estado ligado e *isPumpOff* no estado desligado.

Glossário : Abstrações

```
1  action ActiveSomePumps is
2    choose p : pumps
3    satisfying isPumpOff(p)
4    do p.TurnOn;
5    end
6  end
7
8  action StopSomePumps is
9    choose p : pumps
10   satisfying isPumpOn(p)
11   do p.TurnOff;
12   end
13 end
```

A seguir, são apresentadas algumas definições na álgebra para melhor compreensão do Controle Global.

Glossário : Álgebra

```
1  derived emergencyStop        : Bool := reachingLimitLevel;
2  derived waterLevelAdjusted   : Bool := N1 <= waterLevel and
3                                         N2 >= waterLevel;
4  external reachingLimitLevel : Bool;
```

Não fica claro na especificação de Abrial a definição de *reachingLimitLevel*. Por isso, neste modelo, esta condição é modelada como uma função externa.

5.4.2 DSE-initialMode

No modo de operação *initialMode*, o sistema de controle da caldeira aguarda o sinal de que a caldeira a vapor está pronta. Nesta modelagem, isso é simulado e assume que *steamBoilerWaiting* recebe o sinal enviado pela caldeira. Então, deve-se aguardar que o nível de água esteja correto. Feito isso, o programa envia constantemente mensagens para todas as unidades físicas indicando que está pronto até que receba a confirmação de que todas elas também estão. Se houver alguma falha em uma das unidades vitais, medidora de nível de água, *wm*, e vapor, *sm*, o programa entra no estado de emergência, chamado de *stopped*. Se for encontrado algum erro nas demais unidades físicas, o sistema entra no estado degradado, chamado de *degraded*. Caso contrário, ele entra no estado normal de execução, como pode ser conferido na Figura 5.13.

Pode-se perceber que do estado inicial até o estado *initialMode* existem duas ações e um outro estado, *starting*. Este é um estado secundário utilizado apenas para auxílio

às inicializações. Por não ser descrito por Abrial, não existe foco especial destinado a ele.

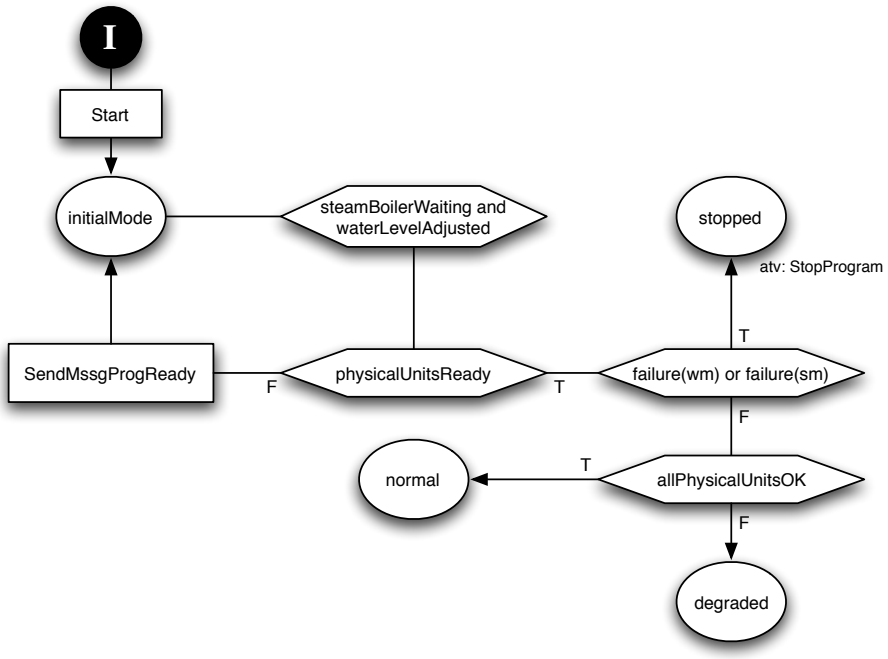


Figura 5.13: DSE-initialMode.

A ação *SendMssgProgReady*, presente no DSE-initialMode, envia para todos os controladores de unidades físicas a mensagem de que o sistema de controle da caldeira está pronto. Dessa forma, o controlador pode iniciar a atividade da unidade associada a ele.

Glossário : Abstrações

```
1  action SendMssgProgReady is
2    forall c : physicalUnitControl
3    do c.RecieveProgReady(true);
4    end
5  end
```

A seguir, define-se a Álgebra para o DSE-initialMode apresentado.

Glossário : Álgebra

```
1  steamBoilerWaiting : Bool;
2
3  derived physicalUnitsReady : Bool := all u : PhysicalUnit
4                                       satisfying ready(u);
5
```

```
6   derived allPhysicalUnistOK : Bool := all u : PhysicalUnit
7                                        satisfying not failure(u);
```

Por fim, define-se a ação *Start*, onde as funções deste módulo são inicializadas, as associações das unidades físicas com seus controladores e estes são disparados.

Glossário : Abstrações

```
1   action Start is
2     step 1:
3       numberOfPumps := 4;
4
5       create valve;
6       create pumps(numberOfPumps);
7       create sm;
8       create wm;
9
10      create physicalUnitControl(numberOfUnits);
11
12      C  := 255;
13      M1 := 10; M2 := 245;
14      N1 := 15; N2 := 235;
15      W  := 200;
16      U1 := 100; U2 := 100;
17
18      waterLevel := 0;
19      steamLevel := 0;
20
21      forall u : PhysicalUnit
22      do ready(u)   := false;
23         failure(u) := false;
24      end
25
26      valveOpened := false;
27
28      forall p : pumps
29      do pumpCtrl(p) := false;
30         isPumpOn    := false;
31      end
32
33      steamBoilerWaiting := true;
34
35    step 2:
36      physicalUnitControl(1).setUnit(valve);
37      physicalUnitControl(2).setUnit(sm);
```

```
38        physicalUnitControl(3).setUnit(sw);
39        physicalUnitControl(4).setUnit(pump(1));
40        physicalUnitControl(5).setUnit(pump(2));
41        physicalUnitControl(6).setUnit(pump(3));
42        physicalUnitControl(7).setUnit(pump(4));
43
44      step 3:
45        forall p : physicalUnitControl do dispatch p;
46    end
```

5.4.3 DSE-normal

O modo normal representa o funcionamento padrão da caldeira a vapor com todas as unidades físicas funcionando corretamente. Na presença de algum erro na unidade medidora de nível de água, o programa passa para o modo de recuperação, estado *rescue*. Qualquer erro em alguma outra unidade física leva o programa para o modo degradado, estado *degraded*. O DSE-normal é mostrado na Figura 5.14.

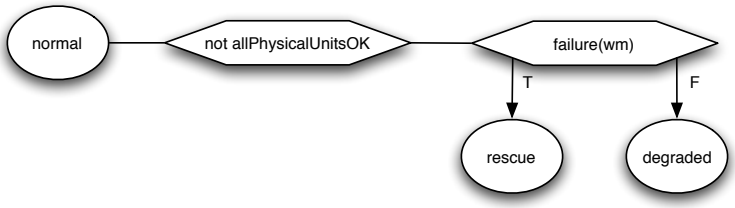


Figura 5.14: DSE-normal.

5.4.4 DSE-degraded

Neste modo de operação, o programa tenta manter o nível normal de água mesmo na presença de falhas de algumas unidades físicas. Assume-se que a unidade medidora de nível de água esteja funcionando corretamente. Caso seja detectado uma falha na unidade medidora de nível de água, o programa passa para o modo de recuperação, estado *rescue*. Uma vez que todas as unidades tenham sido recuperadas, o funcionamento volta a operar normalmente. Estas considerações são mostradas no DSE-degraded na Figura 5.15.

5.4.5 DSE-rescue

O modo de recuperação, estado *rescue*, tem o objetivo de manter o nível de água quando a unidade medidora de nível de água estiver com problemas. Então, para manter o

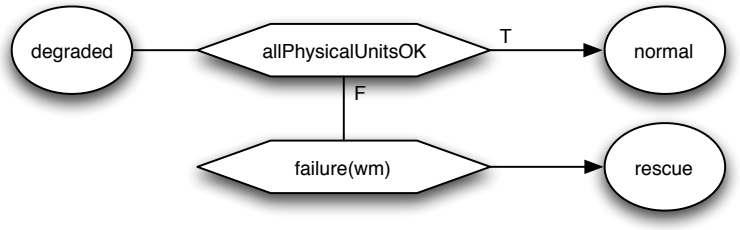


Figura 5.15: DSE-degraded.

correto nível de água na caldeira, é necessário realizar um cálculo utilizando a medição do vapor e as informações da vazão de cada bomba. Logo, a unidade medidora de vapor e cada bomba não podem apresentar problemas. Caso uma dessas duas condições falhe, considera-se um erro grave e o programa é levado ao estado de emergência, *stopped*.

Assim que a unidade medidora de nível de água for reparada, o programa tenta retornar para o modo normal. Então verifica-se se todas as unidades físicas estão funcionando. Caso exista alguma com problemas, retorna-se para o modo degradado. Figura 5.16 mostra o DSE-rescue.

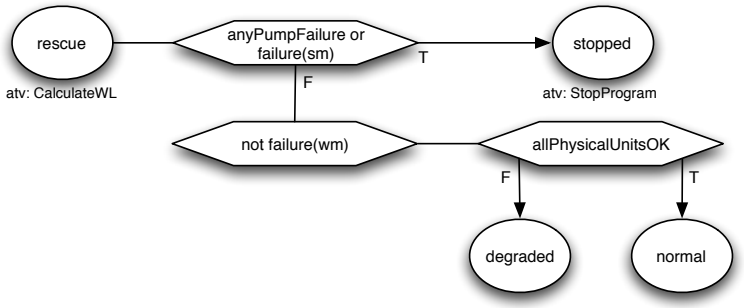


Figura 5.16: DSE-rescue.

A especificação de Abrial não deixa claro como é realizado o cálculo para dedução do nível de água, então, nessa modelagem, este cálculo é simulado obtendo o valor não deterministicamente. Deve-se obter um valor que seja possível infringir os limites inferior e superior M1 e M2, podendo assim simular valores de risco. O cálculo é realizado sempre que se permanece no estado *rescue*, modelado então com a atividade *CalculateWL*, definida a seguir.

Glossário : Abstrações

```
1  action CalculateWL is
2    choose i : 0..(M2 + 1)
3    do waterLevel := i; end
4  end
```

A seguir, define-se a Álgebra para DSE-rescue.

Glossário : Álgebra

```
derived anyPumpFailure : Bool := exists p : pumps
                                 satisfying failure(p);
```

5.4.6 Propriedades

Durante a especificação de Abrial, é possível identificar algumas propriedades para o sistema da caldeira a vapor. A seguir, apresentam-se essas propriedades e depois elas são formalizadas em fórmulas CTL. A troca de mensagens entre as unidades físicas foram tratadas utilizando os serviços disponibilizados na interface, portanto as propriedades que dizem respeito a falhas de mensagens não se aplicam a esta modelagem.

P 1. A válvula é aberta somente no estado inicial, e deve permanecer fechada nos demais estados de funcionamento, excluindo o estado *stopped*;

P 2. a válvula deve ser fechada antes de abrir qualquer uma das bombas;

P 3. uma vez fora do estado *initialMode*, o programa jamais retorna a este estado;

P 4. se o programa recebeu as mensagens STEAM-BOILER-WAITING e PHYSICAL-UNITS-READY, o processo de inicialização termina;

P 5. se o nível de água estiver entre N1 eN2, então todas as bombas devem ser fechadas.

A seguir, a definição em LMM.

Glossário : Propriedades

```
// P 1
AG(curent_state != initialMode ∧ curent_state != stopped -> ValveClosed);

// P 2
AG(not ValveClosed -> allPumpsClosed);

// P 3
AG(current_state != initialMode -> AF current_state != initialMode);


```

```
11  // P 4
12  AG (current_state = initialMode ∧
13                       steamBoilerWaiting ∧
14                       physicalUnitsReady -> AF current_state != initialMode);
15
16  // P 5
17  AG (waterLevelAdjusted -> AF allPumpsClosed);
```

Mesmo em Propriedades, algumas novas funções aparecem e são definidas da seguinte forma:

Glossário : Álgebra

```
1  derived valveClosed    : Bool := not valveOpened;
2  derived allPumpsClosed : Bool := all p : pumps
3                                   satisfying isPumpOff(p);
```

5.5 Diagrama TAD

O diagrama TAD da Figura 5.18 mostra como os módulos estão relacionados. Assume-se *Timer* como sendo um módulo que contabiliza o tempo do ambiente externo.

5.6 Resultados

O refinamento do Modelo Básico para o Sistema de Controle da Caldeira a Vapor, apresentado neste capítulo, gerou o código Machĭna e a especificação NuSMV correspondente. Então, foi realizada a verificação das propriedades perante o Modelo Básico.

A especificação NuSMV gerada consiste na representação de um único sistema que aborda todos os módulos especificados, seus respectivos agentes e suas interações, inclusive trocas de mensagens. Porém, não foi possível obter uma resposta do ambiente de verificação. Em alguns casos isso é possível, e a verificação não retorna se as propriedades foram, ou não, garantidas. Rodando o sistema NuSMV em uma máquina Apple iBook 1.42Ghz, no sistema operacional MacOSX e com 1Gb de memória RAM, esperou-se por mais de quatro horas sem respostas.

A execução NuSMV de um agente de um módulo gera um determinado número de estados no BDD interno. Quando o número de agentes e módulos aumenta, a quantidade de estados cresce de forma exponencial. Isso se deve à forma como as execuções podem se relacionar. No caso de Machĭna, a relação é determinada pela comunicação entre os agentes. Acredita-se que esta comunicação gera um BDD complexo a ponto de

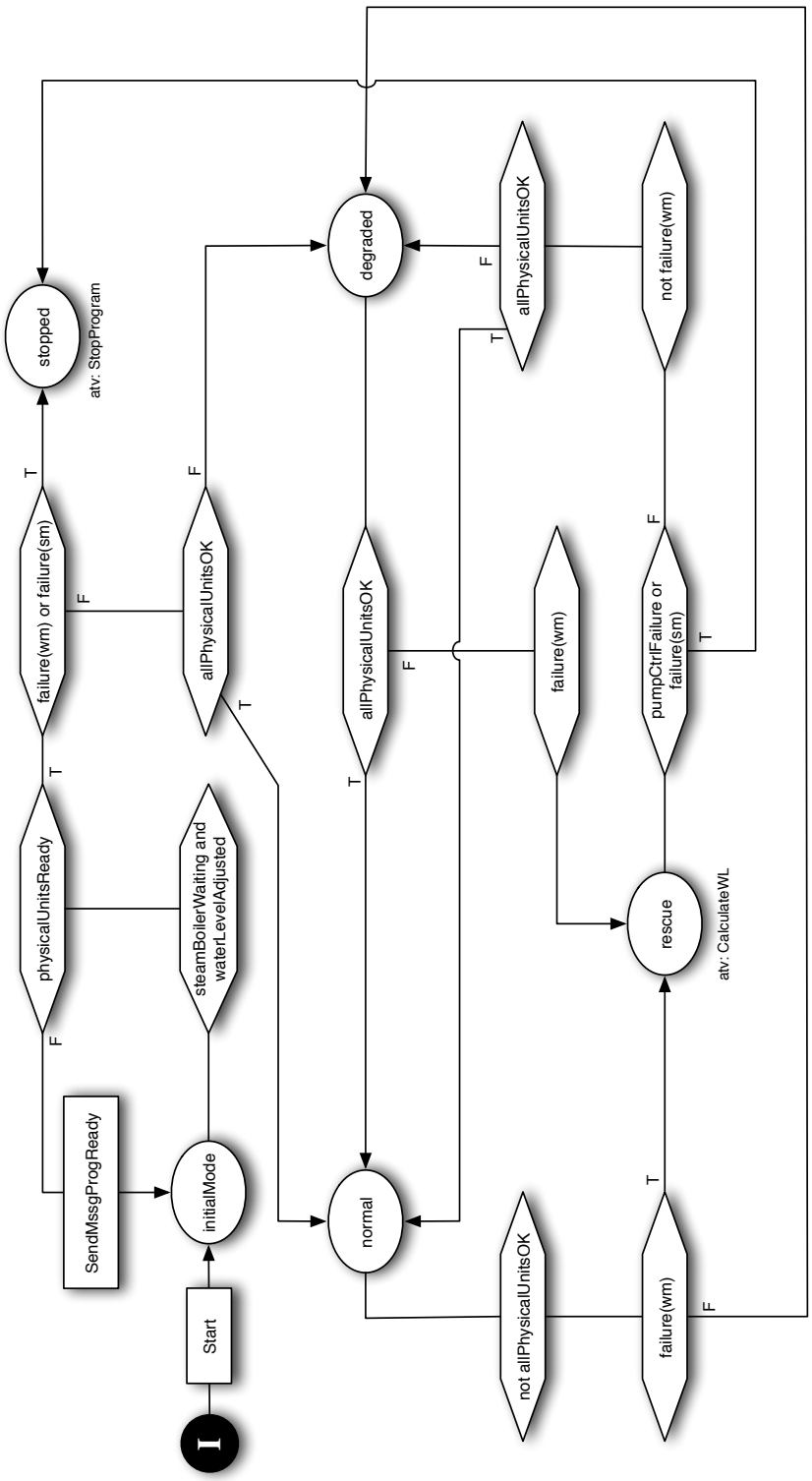


Figura 5.17: DTE *SteamBoilerControl* completo.

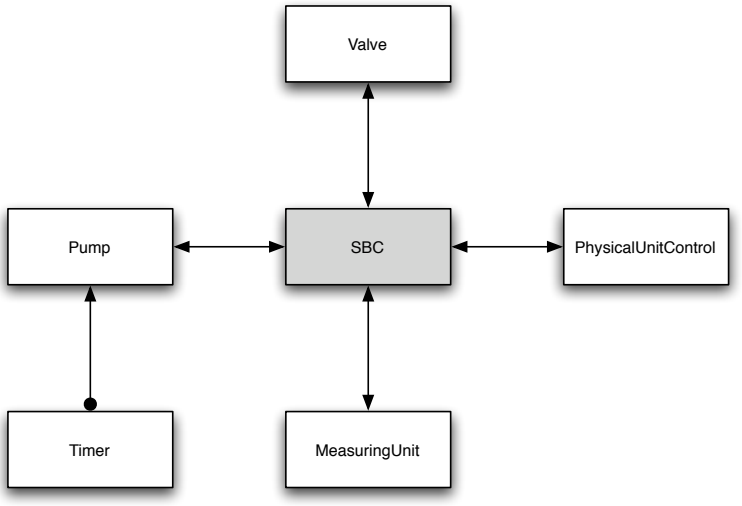


Figura 5.18: Diagrama TAD.

não ser possível obter uma resposta do modelo em tempo hábil. Embora verificadores de modelos tenha grande capacidade de lidar com explosão de estados, não foi possível mostrar este fato com os testes realizados, criando especulações sobre a real causa da falta de resposta na execução NuSMV. Uma justificativa pode ser o tamanho do BDD, que pode ter utilizado uma quantidade excessiva de memória, sendo necessário realizar operações caras no ponto de vista de hierarquia de memória. Outra possibilidade seria a complexidade da estrutura do BDD, tornando a verificação ineficiente.

Uma alternativa para completar a verificação foi alterar o código NuSMV gerado para que cada módulo fosse conferido separadamente, assim as trocas de mensagens foram simuladas por funções externas, que possuem comportamento não determinístico no verificador de modelos utilizado. Este método preserva o mesmo efeito de antes, uma vez que cada agente compreende outro que com ele relaciona via troca de mensagens, sem a necessidade de conhecer sua estrutura interna nem suas propriedades, que se referem apenas aos elementos internos ao módulo. Assim, foram geradas cincos especificações NuSMV, uma para cada módulo, a partir do código originalmente obtido no refinamento, e todas elas retornaram uma resposta para as propriedades. A seguir, Tabela 5.2 mostra os estados alcançáveis dentre os possíveis, de acordo com a especificação NuSMV de cada módulo, e o tempo gasto para verificar as propriedades. Pode-se notar que, conforme as características de verificação de modelos, a quantidade de estados não está diretamente relacionada com o tempo. Estes dados foram coletados com a verificação de um único agente de cada módulo. Testes mostraram que a execução de um número maior de agentes apresenta valores mais elevados de estados e tempo, como é o caso de *pump.smv*, que, quando verificado com quatro bombas, gastou-se um tempo superior a 20 minutos.

Arquivo	Estados alcançáveis	Total de Estados	Tempo
measuring_unit.smv	6.144	36.864	$\cong$ 1s
valve.smv	44	360	$\cong$ 1s
pump.smv	$\cong 2^{21}$	$\cong 2^{52}$	$>$ 3min
puc.smv	1.440	3.600	$\cong$ 1s
sbc.smv	730.432	$\cong 2^{21}$	$\cong$ 1s

Tabela 5.2: Especificações NuSMV.

Para a análise das propriedades, foi considerado apenas o módulo principal *sbc.smv*, mas as conclusões foram as mesmas propriedades para as demais de todos os módulos. Considerando a propriedade P2, que diz que a válvula deve ser fechada antes de abrir qualquer bomba, ou seja, se a válvula estiver aberta, todas as bombas devem estar fechadas.

Propriedade 2

```
1  // P 2
2  AG(not valveClosed -> allPumpsClosed);
```

Esta propriedade foi verificada como falsa, e o seguinte contra-exemplo foi apresentado.

Contra exemplo para propriedade P2

```
1  - specification AG (valveOpened -> allPumpsClosed) IN sbc is false
2  - as demonstrated by the following execution sequence
3  Trace Description:  CTL Counterexample
4  Trace Type:  Counterexample
5  -> State:  1.1 <-
6  sbc.ctl_state = initialMode
7  sbc.step = 1
8  sbc.valveOpened = 0
9  sbc.waterLevel = 0
10 sbc.emergencyStop = 0
11 sbc.P1on = 0
12 sbc.P2on = 0
13 sbc.P3on = 0
14 sbc.P4on = 0
15 sbc.ready = 0
16 sbc.failurePump = 0
17 sbc.failureWL = 0
18 sbc.failureSL = 0
19 sbc.allPumpsClosed = 1
20 sbc.allPhysicalUnitsOk = 1
```

```
21 sbc.waterLevelAdjusted = 0
22 sbc.M2 = 90
23 sbc.M1 = 10
24 sbc.N2 = 80
25 sbc.N1 = 20
26 sbc.C = 100
27 -> State:  1.2 <-
28 sbc.step = 2
29 sbc.P1on = 1
30 sbc.allPumpsClosed = 0
31 -> State:  1.3 <-
32 sbc.step = 1
33 sbc.waterLevel = 4
34 -> State:  1.4 <-
35 sbc.step = 2
36 sbc.waterLevel = 0
37 sbc.P2on = 1
38 -> State:  1.5 <-
39 sbc.step = 1
40 sbc.waterLevel = 85
41 -> State:  1.6 <-
42 sbc.step = 2
43 sbc.valveOpened = 1
44 sbc.waterLevel = 0
45 sbc.P1on = 0
```

As duas primeiras linhas mostram a propriedade verificada e o resultado, no caso falso, e as duas linhas seguintes indicam que há um contra-exemplo a ser mostrado conforme uma propriedade CTL. Cada estado *State* é indicado por um índex, por exemplo *State 1.1* na linha 5 indica o primeiro estado da execução 1, pois em uma inspeção mais detalhada, pode-se conferir mais de uma execução. No exemplo apresentado, este é o estado inicial, onde as variáveis são apenas inicializadas, portanto todas as variáveis são mostradas. Na transição de estados, apenas aparecem as variáveis que foram alteradas, assim, no estado 1.2, linha 27, aparecem apenas três varáveis, indicando que as demais permanecem com o mesmo valor. A listagem é finalizada com o estado em que a propriedade não é satisfeita.

Pode-se notar, que no estado 1.4, a bomba 2 é ligada, linha 37. Quando a válvula é aberta no estado 1.6, linha 43, a bomba 2 ainda permanece ligada. Conforme o contra-exemplo, seu valor não se altera entre os estados 1.4 e 1.6. Analisando este contra-exemplo no Modelo Básico, foi possível identificar o erro. Na ação *ReduceWaterLevel*, é feita a solicitação de desligar algumas bombas e, se estiver no estado *initialMode*, a válvula deve ser aberta.

ReduceWaterLevel

```
1  action ReduceWaterLevel is
2    StopSomePumps;
3    if current_state = initialMode then OpenValve; end
4  end
```

De acordo com a especificação de Abrial, todas as bombas deveriam ser fechadas, caso o estado do controlador da caldeira fosse *initialMode*, então esta ação foi modificada para a seguinte.

ReduceWaterLevel

```
1  action ReduceWaterLevel is
2    if current_state = initialMode then
3      OpenValve;
4      StopPumps;
5    else
6      StopSomePumps;
7    end
8  end
```

Dessa forma, a verificação da propriedade P2 pôde ser garantida, e isto é apresentado no ambiente NuSMV conforme a seguir.

Propriedade P2 garantida

```
1  - specification AG (valveOpened -> allPumpsClosed) IN sbc is true
```

Com o auxílio de contra-exemplos fornecidos pelo modelo NuSMV, foi possível inspecionar facilmente o Modelo Básico e localizar e corrigir o problema apontado. Demais propriedades que falharam na verificação também foram facilmente corrigidas, avaliando a eficácia da integração do Método de Refinamento Machĭna com verificação de modelos.

5.7 Conclusões

Com o exemplo do problema da caldeira a vapor, descrito por Abrial, foi possível mostrar como utilizar a Linguagem de Modelagem Machĭna em um problema real, assim como aplicar o Método de Refinamento Machĭna obtendo os resultados de implementação, documentação e verificação. Foi possível avaliar o alto grau de abstração do diagramas DTE e DSEs, além disso, a descrição textual presente no Modelo Básico au-

xilia na compreensão do problema apresentado e principalmente da solução proposta. Foi possível utilizar tanto o DTE diretamente, por exemplo para unidades físicas, como sua construção a partir de DSEs, como na especificação do sistema de controle da caldeira (SBC). Assim, o volume de informação nestes diagramas pode ser dado de forma legível e pode ser dada a devida concentração em cada estado dos módulos, que são os pontos de observação no MRM. Os níveis de abstração bem definidos na LMM foram fundamentais para obter soluções adequadas em cada domínio e cada etapa do desenvolvimento de sistemas.

Foi constatado, na maioria dos testes, que é necessário realizar uma preparação inicial antes de ativar a execução do sistema. Pode-se citar como exemplo a necessidade do agente *Host* no problema do Jantar dos Filósofos, Seção 3.2.3, e, no problema da Caldeira a Vapor, a necessidade da ação *Start* ser definida em múltiplos passos para a construção correta das relações entre as unidades físicas, Seção 5.4.2. Este é um comportamento que não está diretamente relacionado com solução do problema, mas está presente na especificação em um nível mais elevado de abstração. Por exemplo, no caso do Jantar dos Filósofos, o módulo *Host* é destacado por meio de TAD e DTE. Estas são construções de LMM voltadas ao Domínio da Aplicação, ou seja, para a compreensão de clientes. E neste caso, estão modelando requisitos de *software*, que são aspectos de níveis pertencentes ao Domínio dos Modelos. Portanto, trabalhos futuros podem incorporar um tratamento mais adequado a este tratamento de inicialização do sistema, preservando a distinção dos níveis de abstração, que é uma característica importante no Modelo Básico.

Durante a modelagem de sistemas onde existe grande número de troca de mensagens entre agentes, como o problema da Caldeira a Vapor, percebeu-se a necessidade de uma melhor representação deste fenômeno nos níveis mais elevados da LMM, no caso, os diagramas DTE e DSEs. Muito embora as ações que realizam as trocas de mensagens tenham nomes sugestivos, não fica evidente como a comunicação é realizada. No exemplo do Jantar dos Filósofos, a ação *tryGetForks* é quem realiza a comunicação com o agente *host* para solicitar os garfos, e, aparentemente, isso não fica muito claro no DTE de *Philosophers*. Poderia-se solucionar este problema utilizando o corpo da função *tryGetForks*, *host.getForks(leftFork, rightFork)*, no DTE. Mas podem haver situações onde a comunicação é uma das regras a ser executada, além disso, acredita-se que um estudo sobre representações gráficas de comunicação entre agentes pode apresentar um resultado melhor.

O Controle Global modela comportamentos transversais ao sistema, porém somente é permitido um diagrama reduzido. Para representações de vários comportamentos, pode ocorrer uma mistura de informações, prejudicando a legibilidade. Por exemplo, o Controle Global do Sistema de Controle da Caldeira a Vapor, Seção 5.4.1, trata de três

situações. A primeira, diz respeito ao envio do modo de execução do sistema a todas as unidades físicas, indicado pela ação *SendMssgMode*. Posteriormente, verifica-se se existe uma situação de erro grave, podendo desviar o fluxo do DTE principal. Por fim, o nível da água é analisando e recebe o tratamento adequado, de acordo com o resultado. Em alguns casos, pode ser conveniente apresentar as situações modeladas no Controle Global de forma distinta e independente, como pode-se notar, o envio da mensagem *SendMssgMode* é independente dos demais elementos, que são integrados. Um possível melhoramento seria permitir, sem causar inconsistências no modelo, adição isolada de novos comportamentos, ou seja, cada novo comportamento teria um diagrama reduzido específico. Neste caso, seria necessário definir um método de composição dos trechos do Controle Global. Dessa forma, este recurso apresentaria maior legibilidade e contribuiria para um maior reúso.

Construído o Modelo Básico da caldeira a vapor, foram aplicadas as regras de refinamento presentes em MRM, obtendo corretamente o código Machĭna e a especificação NuSMV correspondentes. Então, esta especificação NuSMV foi submetida ao ambiente de verificação. O resultado não foi o esperado, pois o sistema NuSMV não retornou nenhuma resposta quanto à análise do sistema como um todo.

Então, foi necessário alterar o código gerado a fim de inspecionar separadamente cada módulo, onde a comunicação entre agentes foi simulada. Assim, garantiu-se algumas propriedades, e para as outras foram fornecidos contra-exemplos, que, por meio de análises, chegou-se a conclusões satisfatórias e a correção do problema, conforme demonstrado com a propriedade P2 do módulo *SBC*, que representa o sistema de controle da caldeira a vapor.

Mesmo com os problemas de verificação encontrados, consideram-se satisfatórios os resultados obtidos, como no exemplo apresentado, principalmente no que diz respeito à abstração, conseqüentemente legibilidade, do Modelo Básico nos domínios da Aplicação e dos Modelos, assim como poder de documentação e verificação, após a alteração do código gerado.

Capítulo 6

Conclusão

Este texto apresentou o Método de Refinamento Machĭna, no qual é possível realizar especificações de problemas, de pequeno a grande porte, em um nível de abstração adequado à interpretação em ambos os domínios da Aplicação e dos Modelos, facilitando a validação de projetos. Foi proposta uma linguagem de alto nível baseada em Diagrama de Transição de Estados (DTE) e Diagrama de Saída de Estado (DSE) para escrita do Modelo Básico e regras de refinamento que o transforma em linguagem Machĭna. O módulo de verificação permite transformar Machĭna em uma entrada de verificador de modelos, sendo possível realizar automaticamente a verificação do sistema em termos de sua especificação no Modelo Básico. Assim, é possível evitar erros futuros ou falta de funcionalidades. Posteriormente, em um nível um pouco mais detalhado, utilizando os Diagramas de Saída de Estado, é possível descrever mais características e observar o sistema em partes, concentrando a atenção em pontos importantes do sistema separadamente. A diferença de granularidade de abstração entre DTE e DSEs é importante para capturar e expressar as primeiras necessidades e os principais comportamentos do sistema. A descrição gráfica, baseada em diagramas, contribui para maior legibilidade, consequentemente, maior precisão ao representar as necessidades. A notação de DTE e DSEs é abstrata o suficiente para ser utilizada em outras linguagens do modelo formal ASM. Além disso, LMM é bastante intuitiva, na qual garante-se um grande poder de expressão e fácil aprendizado.

A utilização do Glossário facilita o desenvolvimento, separando completamente o comportamento geral da aplicação de detalhamento de características mais próximas da implementação. As definições no Glossário dos elementos dos diagramas representam a primeira etapa do refinamento, realizado por substituição de máquinas (refinamento de sub-rotinas), que corresponde às técnicas comumente encontradas nos ambientes estudados. Em particular, a definição de propriedades e elementos de pré e pós-condições são fundamentais para verificar automaticamente a implementação em termos da es-

pecificação.

Ao estudar o modelo de execução da linguagem Machĭna, constatou-se sua execução assíncrona e seus pontos de sincronização. Como a Linguagem de Modelagem Machĭna herda características desta linguagem, foi possível descrever uma linguagem em alto nível de descrição do Modelo Básico que modela comportamentos síncronos e assíncronos.

Acreditamos que o poder de expressão, a legibilidade e o formalismo presentes na LMM para descrição do Modelo Básico possibilitam abordagem didática, como na análise de comportamento de algoritmos, e especificação de protocolos para validação e verificação antes de sua implantação. Além disso, sistemas de pequenos e grandes portes podem ser definidos modelando comunicações entre agentes e execuções síncronas e assíncronas, e por fim, *hardware* podem ser verificados e o MRM pode gerar um simulador, facilitando a validação do mesmo.

A especificação de sistemas no Modelo Básico e o resultado gerado pelo verificador de modelos constituem a documentação completa do sistema. Como o código executável é obtido da mesma fonte, implementação e documentação são mantidas sempre coerentes entre si.

Ao realizar a verificação do Modelo Básico, foi necessário aplicar técnicas de refinamento no código Machĭna, que representa o sistema, e nas propriedades definidas. Com o propósito de obter um módulo de verificação automática na Linguagem de Modelagem Machĭna, foi possível estender o benefício também para a linguagem Machĭna. Além disso, o refinamento de Machĭna para uma linguagem de verificador de modelos exigiu a definição de uma linguagem intermediária mais simples, denominada Core-Machĭna, onde é possível realizar trabalhos de extensão na linguagem Machĭna.

O refinamento de Machĭna para NuSMV é um processo semi-automático dentro do Método de Refinamento Machĭna, contribuindo para maior transparência na verificação do Modelo Básico. Dentre os processos que propõem aplicar verificação de modelos em ASM, nenhum aborda o refinamento de tipos abstratos de dados como foi realizado em MRM. Por fim, a verificação em MRM utilizando NuSMV, por meio de contra-exemplos, possibilita a obtenção de dados de entrada do sistema para depuração, no caso de alguma das propriedades reconhecidas não ser satisfeita. Embora foram encontrados alguns problemas, os resultados encontrados utilizando verificação de modelos foram satisfatórios, e mostra-se possibilidades de trabalhos futuros.

Pode-se resumir as contribuições deste trabalho da seguinte forma:

- definição de uma linguagem de alto nível para especificação de sistemas;
- definição de uma linguagem gráfica para descrição do Modelo Básico dentro do modelo de Máquina de Estados Abstrata;

- distinção dos níveis de abstração utilizados para descrever o Modelo Básico;
- ambiente que proporciona fácil validação do sistema em ambos os domínios, da Aplicação e dos Modelos;
- ambiente de especificação de sistemas assíncronos, voltados para o modelo de Máquina de Estado Abstrata;
- definição de um método de refinamento para a linguagem Machĭna;
- implementação do Método de Refinamento Machĭna, tornando o processo de transformação automático;
- obtenção da documentação coerente com a implementação, pois ambos são extraídos da mesma fonte de informação;
- processo de verificação do Modelo Básico de forma semi-automática, proporcionando maior transparência para o desenvolvedor;
- refinamento de tipos abstratos de dados para obter uma representação para verificadores de modelos;
- possibilidade de utilizar o resultado da verificação como entrada para depuração do sistema, com o intuito de realizar novas validações.

6.1 Implementação da Ferramenta

Para a aplicação das regras de refinamento apresentadas nesta dissertação, foi desenvolvida uma ferramenta à qual os exemplos deste trabalho foram submetidos. Inicialmente, foi especificação um *XML Schema* para definir a representação interna do Modelo Básico em XML. Posteriormente, as regras de refinamento são aplicadas sobre esta representação do Modelo Básico até obter os códigos Machĭna e NuSMV finais. O resultado de cada regra é expressa em outro XML, que é utilizado para aplicação da regra seguinte. Assim, a implementação registra passo-a-passo o processo de refinamento.

O ponto de partida para o processo de refinamento é a representação interna em XML do Modelo Básico, sendo assim, a interface gráfica para o Método de Refinamento Machĭna pode ser incorporada posteriormente. Com esta ferramenta, a partir do XML dado, gera-se o código Machĭna e a especificação NuSMV correspondente. A documentação completa não é gerada nessa implementação, pois depende da representação

gráfica de Diagramas de Transição de Estados e Diagramas de Saída de Estado, que, neste trabalho, foram especificados separadamente.

O objetivo da implementação foi avaliar o Método de Refinamento Machĭna em um ambiente real. Foi possível resolver alguns dos principais problemas encontrados durante a avaliação realizada no Capítulo 5. Para o problema da restrição de ajuste (*Fitness Constraint*), foi criado um bloco onde as restrições podem ser manualmente descritas na forma de expressões condicionais *if-then*, considerando os estados do DTE, seus fluxos e definições das ações. Essas expressões são adicionadas ao bloco de invariantes do sistema. Assim, o impacto da definição manual é minimizado quando realiza-se o refinamento automático de Machĭna para C-Machĭna.

Outro problema resolvido na ferramenta foi a definição de uma especificação NuSMV para cada módulo com simulação da comunicação entre os agentes, possibilitando a verificação em partes do modelo completo. Esta abordagem está melhor descrita na Seção 5.6.

A ferramenta implementada pode ser considerada o núcleo da aplicação do Método de Refinamento Machĭna, sendo possível desenvolver futuramente os módulos faltantes com fácil integração ao núcleo existente.

6.2 Trabalhos Relacionados

Alguns trabalhos tiveram maior influência para a definição do Método de Refinamento Machĭna e a Linguagem de Modelagem Machĭna. Primeiramente, *The ASM Refinement Method* [Bör03a] serviu como um guia para definir o Método de Refinamento Machĭna, enquanto o trabalho *The ASM Ground Model Method as a Foundation for Requirements Engineering* [Bör03b] apresentou as principais características e o papel que um Modelo Básico deve conter, que foram aplicadas à Linguagem de Modelagem Machĭna.

Posteriormente, algumas construções da LMM foram influenciadas pela descrição da caldeira a vapor realizada em *Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code* [BBD+96], trabalho que também teve papel importante na validação da linguagem proposta de construção do Modelo Básico. Os diagramas DTE e DSEs foram inspirados nos diagramas de estados de UML [AN05, RJB04, BRJ05] e o guia do usuário *Unified Modeling Language User Guide* [BRJ05] teve importante papel na apresentação da notação de LMM.

O livro de ASM, conhecido como *The ASM Book* (*Abstract State Machines: A Method for High-Level System Design and Analysis* [BS03]) foi a principal referência para o trabalho Método de Refinamento Machĭna, pois contém as descrições completas do modelo ASM, do Método de Refinamento ASM, transformação ASM para FSM e

aponta os principais problemas existentes no Método de Refinamento ASM, além de documentar inúmeras referências que foram utilizadas para a definição de MRM.

Por fim, *The ASM Workbench* [Del01, Del99] serviu como inspiração de objetivos na definição do ambiente de MRM, e o livro *Model Checking* [CGP00] apresentou o modelo para verificação automática, assim como o trabalho *Model Checking Abstract State Machines* [Win01] descreveu a transformação de ASM para SMV, influenciando o módulo de verificação automática pretendido pelo Método de Refinamento Machĭna.

É possível encontrar vários trabalhos que propõem ambientes para se aplicar o Método de Refinamento ASM. São apresentadas linguagens para descrição do Modelo Básico em diversos níveis e trabalhos complementares mostram a utilização do método de refinamento. A seguir, apresentam-se os principais ambientes que influenciaram o trabalho Método de Refinamento Machĭna.

A linguagem AsmL (*Abstract State Machine Language*) [GRS04, BS01, Res06] foi desenvolvida com a proposta de especificar, simular e testar interfaces de componentes COM da *Microsoft* na plataforma .NET. Esta linguagem oferece descrição baseada em componentes e orientação a objetos. AsmL suporta pré e pós condições de funções, gerando exceções caso alguma dessas cláusulas não seja respeitada. Sua utilização como Modelo Básico, e no método de refinamento ASM, permite que AsmL seja executada, antes que se gere o código executável, por meio da *AsmL Test Tool*. Como recurso de documentação do sistema, é possível utilizar *Microsoft Word* para descrever e especificar o sistema, e posteriormente submeter este documento à ferramenta *AsmL Test Tool* [Res06] ou ao compilador da linguagem, que gera código C e pode ser associado à plataforma .NET.

XASM (*Extensible ASM*) [Anl00, AK01] é um projeto *Open Source* onde é possível especificar sistemas no modelo ASM com a possibilidade de depurar em ambiente visual e animar passo a passo a execução. Especialmente para especificação de gramáticas, uma visão diferenciada é apresentada onde mostram-se valores de funções referentes aos nodos da árvore de *parser*. É apresentado um compilador para C onde é possível compilar módulos separadamente e colocados em bibliotecas para posterior uso, aumentando a reusabilidade de componentes. XASM oferece uma interface para C onde é possível definir funções em C para serem utilizadas como estáticas ou como funções ASM monitoradas, e definições de funções externas. Além disso, esta linguagem contém um recurso baseado na proposta de Knuth, *Literate Programming* [Knu83], onde um arquivo contendo a especificação e documentação em LaTeX é submetido diretamente ao compilador XASM, gerando a documentação e o código em C partindo da mesma fonte. É possível encontrar diversas especificações que aplicam o método de refinamento ASM utilizando XASM como Modelo Básico.

The ASM Workbench [Del99] define uma linguagem de specificação ASM, chamada

ASM-SL (*ASM Specification Language*), e uma arquitetura para uma ferramenta, *ASM Workbench*. O ambiente desenvolvido é um arcabouço para desenvolvimento ASM e inclui funcionalidades básicas, como *parser*, representação interna do modelo em uma árvore sintática abstrata, verificação de tipos, um interpretador para simulação e animações de execução. Um módulo de grande relevância realiza a transformação do Modelo Básico em um FSM, obtendo uma linguagem intermediária, ASM-IL (*ASM Intermediate Language*), e seu respectivo refinamento para uma entrada de verificador de modelos, no caso o SMV. Tal módulo pode ser conferido em [Win01].

AsmGofer [Schb, Scha] é um sistema para especificação em ASM construído como uma extensão da linguagem funcional Gofer [pe06] onde foram introduzidos conceitos de estados e atualizações paralelas. Esta ferramenta apresenta uma interface gráfica com facilidade para depuração e suporte a técnicas de *Literate Programming*. Os conceitos de estruturas e composição implementadas em AsmGofer facilitam estruturação e composição de sub-máquinas, facilitando a aplicação dos diferentes tipos de refinamento.

A linguagem CoreASM [FGG06, FGG05, FGGM06, Far06] foi recentemente desenvolvida para estar o mais próximo possível da definição ASM no *Lipari Guide* [Gur95]. Assim, seu objetivo é proporcionar um ambiente de alto nível para especificação de sistemas, com um embasamento matemático, modelos abstratos e sem tipos, permitindo o refinamento passo a passo para implementação. O projeto CoreASM consiste em um *parser*, um interpretador, um escalonador e um mecanismo abstrato de armazenagem. Assim é possível executar as máquinas ASMs especificadas. A estrutura da ferramenta é baseada em *plugins*, facilitando extensões à linguagem caso necessário.

Embora nenhum destes ambientes descreva uma linguagem gráfica de especificação, existem trabalhos que propõem uma semântica formal para diagrama UML, principalmente de diagramas de transição de estados e atividades, como é o caso dos trabalhos de Ileana Ober [Obe03], de Alexander Knapp e Stephan Merz [KM02] e Börger, Cavarra e Riccobene [BCR00b, BCR00a]. Desta forma, surgem indicativos para integrações dos ambientes apresentados com uma linguagem gráfica, como é o caso do trabalho [Cav00], que estende AsmGofer para ser um simulador de máquinas de estados de UML. No caso dos trabalhos de Ileana Ober [Obe00, Obe03] e no trabalho [CRS03], utilizou-se *ASM Workbench* para definir uma semântica para diagramas UML. Outros detalhes sobre a utilização de diagramas UML para descrição ASM podem ser encontrados no livro sobre ASM [BS03, Seção 6.5.1].

Todos estes trabalhos se baseiam na especificação de UML. Embora tenha uma proposta de modelagem universal, os esforços nesta linguagem são, na maioria, direcionados à linguagens imperativas e orientadas a objetos. Dessa forma, a utilização direta

de UML para modelar sistemas em ASM pode acarretar em representações desnecessárias, como é o caso de definições de execução em paralelo, princípio no qual é implícito ao modelo ASM. Além disso, conceitos presentes em ASM podem não ter uma definição específica em UML. Em decorrência destes fatos, o Método de Refinamento Machĭna propõe sua linguagem de especificação de Modelo Básico, a Linguagem de Modelagem Machĭna, baseada em diagramas de estados de UML, porém com representações mais simples e com construções direcionadas ao modelo ASM.

Considerando AsmL, XASM, *The ASM Workbench* e AsmGofer, mencionados anteriormente, Börger relata dois principais problemas em [BS03, Seção 8.3]. Primeiro, estes ambientes apenas suportam técnicas de refinamento considerando sub-máquinas, refinamento de dados e sub-rotina. O segundo, nenhum deles apresenta o modelo ASM assíncrono. Embora CoreASM não esteja nesta avaliação de Börger, este sistema também não resolve nenhum desses dois problemas.

Com a utilização de Diagrama de Transição de Estados, Diagrama de Saída de Estado e Controle Global, Seções 3.1.3, 3.3.3 e 3.1.3.9 respectivamente, em LMM, é possível utilizar automaticamente o refinamento de Sub-Rotinas e Extensão Conservativa, e com Tipo Abstrato de Dados, realiza-se o Refinamento de Dados, que não estão presentes nos ambientes mencionados anteriormente. Assim, é possível abordar todos os tipos de refinamento mencionados na Seção 2.2.2.

A definição de propriedades no Glossário de LMM é fundamental para expressar características do Domínio da Aplicação e as demais ferramentas citadas não apresentam tal possibilidade embutida na descrição do Modelo Básico. Integrada à linguagem de escrita do Modelo Básico, as propriedades têm importante papel na verificação automática e transparente da especificação.

Além disso, LMM é definida em termos da linguagem Machĭna e herda suas características, sendo assim, LMM permite modelar execuções assíncronas naturalmente, e pontos de sincronismo são definidos por comunicação entre agentes onde se encontram parâmetros classificados como **out** ou **inout**.

Ainda não está completamente definido um ambiente de execução real para a linguagem Machĭna. Os trabalhos realizados por Mário Lobado [Lob06] e Kristian Santos [dS06] têm o objetivo de construir um compilador de Machĭna para gerar código em C++ com determinadas otimizações de código. Porém, alterações com novas construções na linguagem impediram o uso adequado deste compilador. Na Seção de trabalhos futuros, 6.3, este tema será novamente abordado com novas propostas.

6.3 Trabalhos Futuros

Conforme descrito no Capítulo 5, ainda existem alguns estudos que podem ser realizados para melhorar a representação gráfica do Modelo Básico. É possível propor uma maneira mais adequada e não intrusiva para descrever a preparação inicial do sistema, assim como para expressar em alto nível a comunicação entre os agentes via serviços disponibilizados. Uma outra melhoria é a possibilidade de separação de comportamentos representados no Controle Global de um módulo, visando maior legibilidade e reúso.

Além disso, atualmente, as abstrações apresentadas pelos Diagramas de Saída de Estado e pelo Diagrama de Transição de Estados possibilitam a validação manual do sistema realizada sob a supervisão de clientes e desenvolvedores, isto é, realizada nos domínios da Aplicação (cliente) e dos Modelos (desenvolvedores). Um possível trabalho seria possibilitar a execução do Modelo Básico interativamente recebendo como entrada dados do usuário ou saídas de contra-exemplo do verificador de modelos. Outro tema interessante é a animação em cima dos diagramas de LMM enquanto o sistema é executado. Estes artefatos teriam o objetivo de facilitar a depuração do sistema e tornar a validação um processo mais amigável.

Durante o refinamento de Machĭna para uma entrada de verificador de modelos, alguns pontos da linguagem não foram possíveis de serem tratados. Pode-se citar o tratamento da regra de ajuste no refinamento de funções com vários argumentos, ações com regra de transição com *loop* e alguns tipos de Machĭna que não foram tratados. Durante os estudos de verificação automática, foi possível constatar a densidade do tema e a dificuldade quando aplicada ao modelo ASM. Este é assunto que ainda pode ser bastante explorado, principalmente para o ambiente do Método de Refinamento Machĭna descrito.

A transformação para NuSMV, representando o sistema como um todo, causou uma situação onde não foi possível obter uma resposta do modelo NuSMV. As alterações realizadas mostraram que a decomposição dos módulos para obter a verificação separada foram satisfatórias, indicando uma nova abordagem melhor que a inicial.

Durante o refinamento de Machĭna para uma entrada de verificador de modelos, gera-se uma linguagem intermediária chamada de C-Machĭna(Core-Machĭna). Esta pode ser considerada o núcleo da linguagem Machĭna. Sendo assim, é possível reduzir o trabalho de construção de um compilador ou interpretador se estes forem desenvolvidos para C-Machĭna, que é uma linguagem mais simples e com menos construções a serem avaliadas. Além disso, futuras construções em Machĭna podem ser definidas sem a necessidade de alterar o compilador, bastando apenas apresentar o refinamento Machĭna para C-Machĭna correspondente, conforme apresentado na Seção 4.4 para a

versão atual de Machĭna.

Outra vantagem da linguagem C-Machĭna é sua proximidade com linguagens de entrada de verificadores de modelos. Então, é possível realizar, com certa facilidade, o refinamento da especificação em C-Machĭna para outras ferramentas além de NuSMV. Também seira interessante avaliar uma transformação para provadores de teorema e confrontar essas abordagens em termos de ambientes e ferramentas para contribuir com melhores resultados na verificação de sistemas.

Pode-se resumir os trabalhos futuros da seguinte forma:

- descrição mais adequada da preparação do sistema, de forma não intrusiva e utilizando uma representação apropriada ao Domínio dos Modelos;
- obter uma representação gráfica para modelar comunicação entre agentes de forma simples e objetiva;
- incorporação de novos comportamentos ao sistema de forma independente entre si, melhorando a legibilidade do Controle Global;
- desenvolvimento de um ambiente para validação interativa e ilustrativa;
- refinamento de Machĭna para C-Machĭna dos itens faltantes: alguns tipos e ações com *loop*;
- tratamento para reconhecimento automático da regra de ajustes em funções com argumentos;
- especificação da linguagem C-Machĭna, onde construções de Machĭna são definidas em termos de construções mais simples;
- desenvolvimento de um compilador, interpretador ou ambiente de execução para C-Machĭna, de forma a simplificar este trabalho;
- transformação Machĭna para NuSMV decomposta em módulos, que são verificados separadamente simulando as comunicações entre agentes.
- transformação de C-Machĭna em entradas para diferentes verificadores de modelos, permitindo melhor avaliação dos métodos formais de verificação.

Referências Bibliográficas

[ABL95] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, *Preliminar report for the Dagstuhl-Seminar 9523: Methods fors Semantics and Specification*, 1995.

[ABL96] J.-R. Abrial, E. Börger, and H. Langmaack, *The steam boiler case study: Competition of formal program specification and development methods*, Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control (J.-R. Abrial, E. Börger, and H. Langmaack, eds.), Lecture Notes in Computer Science, vol. 1165, Springer-Verlag, 1996, pp. 1–12.

[Abr96a] J.-R. Abrial, *The b-book*, Cambridge University Press, Cambridge, 1996.

[Abr96b] Jean-Raymond Abrial, *Steam-Boiler Control Specification Problem*, Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995). (London, UK), Springer-Verlag, 1996, pp. 500–509.

[AK01] M. Anlauff and P. Kutter, *Xasm Open Source*, Web pages at http://www.xasm.org/, 2001, http://www.xasm.org/.

[AN05] Jim Arlow and Ila Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*, 2nd ed., Addison-Wesley Object Technology Series, Addison-Wesley Professional, 2005.

[Anl00] M. Anlauff, *XASM – An Extensible, Component-Based Abstract State Machines Language*, Abstract State Machines: Theory and Applications (Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, ed.), LNCS, vol. 1912, Springer-Verlag, 2000, pp. 69–90.

[Bac80] Ralph-Johan Back, *Correctness preserving program refinements: Proof theory and applications*, Mathematical Center Tracts, vol. 131, Mathematical Centre, Amsterdam, The Netherlands, 1980.

[Bac81] Ralph-Johan Back, *On correct refinement of programs.*, J. Comput. Syst. Sci. **23** (1981), no. 1, 49–68.

[BBD+96] C. Beierle, E. Börger, I. Durdanovic, U. Glässer, and E. Riccobene, *Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code*, Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control (J.-R. Abrial, E. Börger, and H. Langmaack, eds.), LNCS, no. 1165, Springer, 1996, pp. 62–78.

[BBK+04] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums, *Interactive verification of uml state machines*, Proc. 6th Int. Conf. Formal Engineering Methods (ICFEM'04), vol. 3308, 2004, pp. 434–448.

[BCR00a] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene, *An asm semantics for uml activity diagrams*, AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (London, UK), Springer-Verlag, 2000, pp. 293–308.

[BCR00b] ———, *Modeling the dynamics of uml state machines*, ASM '00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications (London, UK), Springer-Verlag, 2000, pp. 223–241.

[Ber02] Daniel M. Berry, *The inevitable pain of software development, including of extreme programming, caused by requirements volatility*, International Workshop on Time-Constrained Requirements Engineering 2002 (TCRE'02) (2002), 9–19.

[BG00] E. Börger and R. Gotzhein, *The light control case study*, J. Universal Computer Science **6** (2000), no. 7, 580–585.

[Bör90a] E. Börger, *A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control*, CSL'89. 3rd Workshop on Computer Science Logic (E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, eds.), LNCS, vol. 440, Springer, 1990, pp. 36–64.

[Bör90b] ———, *A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation*, Mathematical Foundations of Computer Science (B. Rovan, ed.), LNCS, vol. 452, Springer, 1990, pp. 1–14.

[Bör92] ———, *A Logical Operational Semantics for Full Prolog. Part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output*, Logic From Computer Science (Y. Moschovakis, ed.), Berkeley Mathematical Sciences Research Institute Publications, vol. 21, Springer, 1992, pp. 17–50.

[Bör03a] E. Börger, *The ASM Refinement Method*, Formal Aspects of Computing **15** (2003), 237–257.

[Bör03b] Egon Börger, *The ASM Ground Model Method as a Foundation for Requirements Engineering.*, Verification: Theory and Practice, 2003, pp. 145–160.

[BPS00] E. Börger, P. Päppinghaus, and J. Schmid, *Report on a Practical Application of ASMs in Software Design*, Abstract State Machines: Theory and Applications (Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, ed.), LNCS, vol. 1912, Springer-Verlag, 2000, pp. 361–366.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson, *Unified Modeling Language User Guide*, 2nd ed., Addison-Wesley Object Technology Series, Addison-Wesley Object Technology Series, 2005.

[BRS+00a] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums, *Formal system development with kiv.*, FASE, 2000, pp. 363–366.

[BRS00b] E. Börger, E. Riccobene, and J. Schmid, *Capturing requirements by Abstract State Machines: The light control case study*, J. Universal Computer Science **6** (2000), no. 7, 597–620.

[BS98a] E. Börger and W. Schulte, *Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation*, Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic (L. Brim and J. Gruska and J. Zlatuska, ed.), LNCS, no. 1450, Springer, August 1998.

[BS98b] ———, *Programmer Friendly Modular Definition of the Semantics of Java*, Formal Syntax and Semantics of Java (J. Alves-Foss, ed.), LNCS, no. 1523, Springer, 1998.

[BS01] M. Barnett and W. Schulte, *The ABCs of Specification: AsmL, Behavior, and Components*, November 2001.

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

[BTIB05] R. S. Bigonha, F. Tirelo, V. O. Di Iorio, and M. A. S. Bigonha, *A linguagem de especificação formal Machĭna 2.0*, Tech. Report RT 001/2005, LLP/DCC/UFMG, Novembro 2005.

[BvW98] R. J. R. Back and J. von Wright, *Refinement calculus: A systematic introduction*, Springer-Verlag, 1998.

[Cav00] A. Cavarra, *Applying Abstract State Machines to formalize and integrate the UML lightweight method*, Ph.D. thesis, University of Catania, Sicily, Italy, 2000.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking*, Proc. International Conference on Computer-Aided Verification (CAV 2002) (Copenhagen, Denmark), LNCS, vol. 2404, Springer, July 2002.

[CCO+05a] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Gavin Keighren, Marco Pistore, Marco Roveri, Simone Semprini, and Andrey Tchaltsev, *Nusmv 2.3 tutorial*, Tech. report, ITC-IRST, 2005, http://nusmv.irst.itc.it/.

[CCO+05b] ———, *Nusmv 2.3 user manual*, Tech. report, ITC-IRST, 2005, http://nusmv.irst.itc.it/.

[CCO+06] ———, *Nusmv home page*, URL, November 2006, http://nusmv.irst.itc.it/.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled, *Model checking*, The MIT Press, 2000.

[CP02] G. Del Castillo and P. Päppinghaus, *Designing software for internet telephony: experiences in an industrial development process*, Theory and Applications of Abstract State Machines (A. Blass, E. Börger, and Y. Gurevich, eds.), Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 2002.

[CRS03] A. Cavarra, E. Riccobene, and P. Scandurra, *Integrating UML static and dynamic views and formalizing the interaction mechanism of UML state*

machines, Abstract State Machines 2003–Advances in Theory and Applications (E. Börger, A. Gargantini, and E. Riccobene, eds.), Lecture Notes in Computer Science, vol. 2589, Springer-Verlag, 2003, pp. 229–243.

[CW00] Giuseppe Del Castillo and Kirsten Winter, *Model checking support for the ASM high-level language*, Tools and Algorithms for Construction and Analysis of Systems, 2000, pp. 331–346.

[CZS+97a] Francisco Corella, Z. Zhou, Xiaoyu Song, Michel Langevin, and Eduard Cerny, *Multiway Decision Graphs for Automated Hardware Verification*, Formal Methods in System Design **10** (1997), no. 1, 7–46.

[CZS+97b] ———, *Multiway decision graphs for automated hardware verification*, Formal Methods in System Design **10** (1997), no. 1, 7–46.

[DB01] J. Derrick and E. Boiten, *Refinement in z and object-z*, Formal Approaches to Computing and Information Technology, Springer-Verlag, 2001.

[Del99] G. Del Castillo, *Towards Comprehensive Tool Support for Abstract State Machines: The ASM Workbench Tool Environment and Architecture*, Applied Formal Methods — FM-Trends 98 **1641** (1999), 311–325.

[Del01] ———, *The ASM workbench. a tool environment for computer-aided analysis and validation of abstract state machine models*, Ph.D. thesis, Universität Paderborn, Germany, 2001.

[Dij68] E. W. Dijkstra, *A constructive approach to the problem of program correctness*, BIT **8** (1968), no. 3, 174–186.

[Dij76] E. W. Dijkstra, *A discipline of programming*, Prentice-Hall, 1976, DIJ e 76:1 1.Ex.

[Dis99] S. Distefano, *Architettura X86 e sistema dei processi di MINIX dalla specifica asm al codice eseguibile*, Master's thesis, University of Catania, Sicily, Italy, 1998/99.

[dRE98] W. P. de Roever and K. Engelhardt, *Data refinement: Model-oriented proof methods and their comparison*, Cambridge University Press, Cambridge, 1998.

[dS06] Kristian Magnani dos Santos, *Um arcabouço para otimizações em máquinas de estado abstratas*, Master's thesis, Universidade Federal de Minas Gerais, 2006.

[E. 99] E. Börger, *High Level System Design and Analysis using Abstract State Machines*, Current Trends in Applied Formal Methods (FM-Trends 98) (D. Hutter and W. Stephan and P. Traverso and M. Ullmann, ed.), LNCS, no. 1641, Springer-Verlag, 1999, pp. 1–43.

[Far06] Roozbeh Farahbod, *CoreASM User Manual*, September 2006.

[FGG05] R. Farahbod, V. Gervasi, and U. Glässer, *Design and Specification of the CoreASM Execution Engine.*, Tech. Report SFU-CMPT-TR-2005-02, Computing Science, Simon Fraser University, Burnaby, B.C., Canada, February 2005.

[FGG06] ———, *CoreASM: An Extensible ASM Execution Engine*, September 2006.

[FGGM06] R. Farahbod, V. Gervasi, U. Glässer, and M. Memon, *Design and Specification of the CoreASM Execution Engine. Part 1: The Kernel*, Tech. Report SFU-CMPT-TR-2006-09, Computing Science, Simon Fraser University, Burnaby, B.C., Canada, May 2006.

[FL98] J. Fitzgerald and P. G. Larsen, *Modeling systems. practical tool and techniques in software development*, Cambridge University Press, Cambridge, 1998.

[FPB87] Jr. Frederick P. Brooks, *No silver bullet: essence and accidents of software engineering*, Computer **20** (1987), no. 4, 10–19.

[FSS+94] T. Filkorn, H. A. Schneider, A. Scholz, A. Strasser, and P Warkentin, *SVE User's Guide*, Tech. Report ZFE T SE 1, D-81730, Siemens AG, Munchen, Germany, 1994.

[FWD05] David Faitelson, James Welch, and Jim Davies, *From predicates to programs: the semantics of a method language*, Simpósio Brasileiro de Métodos Formais (SBMF) (2005).

[GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes, *Generating finite state machines from abstract state machines*, ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (New York, NY, USA), ACM Press, 2002, pp. 112–122.

[GRS04] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte, *Semantic Essence of AsmL*, Technical Report MSR-TR-2004-27, Microsoft Research, March 2004.

[Gur95] Y. Gurevich, *Evolving Algebras 1993: Lipari Guide*, Specification and Validation Methods (E. Börger, ed.), Oxford University Press, 1995, pp. 9–36.

[HHK96] R. H. Hardin, Z. Har'El, and R. P. Kurshan, *COSPAN*, 423–427.

[Hol97] Gerard J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), no. 5, 279–295.

[IBM07] IBM, *Ibm software - rational method composer - rational unified process*, February 2007, http://www-306.ibm.com/software/awdtools/rup/.

[ISEJ05] Subramanian K. Iyer, Debashis Sahoo, E. Allen Emerson, and Jawahar Jain, *On partitioning and symbolic model checking.*, FM (John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, eds.), Lecture Notes in Computer Science, vol. 3582, Springer, 2005, pp. 497–511.

[JED91] J.R. Burch, E.M. Clarke, and D.E. Long, *Symbolic model checking with partitioned transition relations*, International Conference on Very Large Scale Integration (1991), 49–58.

[JED+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill, *Symbolic model checking for sequential circuit verification*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **13** (1994), no. 4, 401–424.

[K.L92] K.L. McMillan, *The SMV system*, Tech. Report CMU-CS-92-131, CMU, 1992.

[KM02] Alexander Knapp and Stephan Merz, *Model checking and code generation for uml state machines and collaborations*, Proc. 5th Wsh. Tools for System Design and Verification (2002), 69–64.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker, *PRISM: Probabilistic symbolic model checker*, Computer Performance Evaluation / TOOLS, 2002, pp. 200–204.

[Knu83] Donald E. Knuth, *Literate programming*, Technical report STAN-CS-83-981, Stanford University, Department of Computer Science, 1983.

[Knu84] ———, *The TeXbook*, spiral edition ed., Addison-Wesley Professional, January 1984.

[Lob06] Mário Celso Candian Lobato, *Um arcabouço para compilação de linguagens de especificação asm*, Master's thesis, Universidade Federal de Minas Gerais, 2006.

[LP99a] Johan Lilius and Ivan P Paltor, *The production cell: An exercise in the formal verification of a uml model*, Tech. report, Turku Centre for Computer Science, 1999.

[LP99b] Johan Lilius and Ivan Porres Paltor, *vUML: a tool for verifying UML models*, Tech. Report TUCS-TR-272, Turku Centre for Computer Science, 1999.

[LSRC02] Charles E. Leiserson, Clifford Stein, Ronald L. Rivest, and Thomas H. Cormen, *Algoritmos: Teoria e Prática*, tradução da segunda edição ed., Campus, 2002.

[Ltd97] Formal Systems (Europe) Ltd., *Failure Divergence Refinement: FDR2 User Manual*, 1997.

[LW99] Dean Leffingwell and Don Widrig, *Managing Software Requirements - A Unified Approach*, 1st ed., Addison Wesley Professional., October 1999.

[McM93] K. L. McMillan, *Symbolic model checking*, Ph.D. thesis, Kluwer Academic Publishers, 1993.

[McM99] ———, *Getting started with SMV*, Tech. Report CA 94704, Cadence Berkeley Labs, 1999.

[Mea97] L. Mearelli, *Refining an ASM specification of the production cell to C++ code*, J. Universal Computer Science **3** (1997), no. 5, 666–688.

[Mey97] Bertrand Meyer, *Object-oriented software construction*, 2nd edition ed., Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Mor87] Joseph M. Morris, *A theoretical basis for stepwise refinement and the programming calculus*, Sci. Comput. Program. **9** (1987), no. 3, 287–306.

[Mor94] Carroll Morgan, *Programming from specifications (2nd ed.)*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[Obe00] I. Ober, *More meaningful UML models*, Proc. TOOLS (Sydney, Australia), IEEE Computer Society Press, 20–23 November 2000, pp. 146–157.

[Obe03] Ileana Ober, *An ASM semantics for UML derived from the meta-model and incorporating actions*, Abstract State Machines - Advances in Theory and Applications., LNCS, vol. 2589, Proceedings 10th International Workshop, ASM 2003, 2003.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar, *PVS: A prototype verification system*, 11th International Conference on Automated Deduction (CADE) (Saratoga, NY) (Deepak Kapur, ed.), Lecture Notes in Artificial Intelligence, vol. 607, Springer-Verlag, jun 1992, pp. 748–752.

[ORSSC98] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert, *PVS: an experience report*, Applied Formal Methods—FM-Trends 98 (Boppard, Germany) (Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, eds.), Lecture Notes in Computer Science, vol. 1641, Springer-Verlag, oct 1998, pp. 338–345.

[pe06] Gofer Functional programming environment, *Gofer – functional programming environment*, December 2006, http://www.cse.ogi.edu/ mpj/goferarc/index.html.

[PN06] Larry Paulson and Tobias Nipkow, *Isabelle*, URL, November 2006, http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[Rei92] W. Reif, *The KIV System: Systematic Construction of Verified Software*, 11th Conference on Automated Deduction. Proceedings (D. Kapur, ed.), Lecture Notes in Computer Science, Albany, NY, USA, Springer, 1992.

[Rei95] ———, *The KIV Approach to Software Verification*, Korso: Methods, Languages, and Tools for the Construction of Correct Software (M. Broy and S. Jähnichen, eds.), Lecture Notes in Computer Science, vol. 1009, Springer, 1995.

[Res06] Microsoft Research, *The abstract state machine language*, URL, December 2006, http://research.microsoft.com/fse/asml/.

[RGA+96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. -T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, *VIS:*

a system for verification and synthesis, Proceedings of the Eighth International Conference on Computer Aided Verification CAV (New Brunswick, NJ, USA) (Rajeev Alur and Thomas A. Henzinger, eds.), vol. 1102, Springer Verlag, / 1996, pp. 428–432.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual*, 2nd ed., Addison-Wesley Object Technology Series, Addison-Wesley Professional, 2004.

[Scha] J. Schmid, *Executing ASM specifications with AsmGofer*, Web pages at http://www.tydo.de/AsmGofer.

[Schb] Joachim Schmid, *Asmgofer*, http://www.tydo.de/AsmGofer/.

[Sch01a] G. Schellhorn, *Verification of ASM Refinements Using Generalized Forward Simulation*, Journal of Universal Computer Science **7** (2001), no. 11, 952–979.

[Sch01b] J. Schmid, *Compiling Abstract State Machines to C++*, Journal of Universal Computer Science **7** (2001), no. 11, 1068–1087.

[Spi99] M. Spielmann, *Automatic Verification of Abstract State Machines*, Proceedings of 11th International Conference on Computer-Aided Verification (CAV '99), LNCS, vol. 1633, Springer-Verlag, 1999, pp. 431–442.

[Spi00] M. Spielmann, *Model Checking Abstract State Machines and Beyond*, International Workshop on Abstract State Machines ASM 2000, LNCS, Springer-Verlag, 2000, pp. 323–340.

[SSB01] R. Stärk, J. Schmid, and E. Börger, *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001.

[WD96] J. C. P. Woodcock and J. Davies, *Using z: Specification, refinement, and proof*, Prentice-Hall, 1996.

[Wie06] Freek Wiedijk, *The seventeen provers of the world*, Lecture Notes in Computer Science, vol. 3600, Springer, 2006.

[Win97] K. Winter, *Model checking for abstract state machines*, Journal of Universal Computer Science **3** (1997), no. 5, 689–701.

[Win01] Kirsten Winter, *Model checking abstract state machines*, Ph.D. thesis, Technical University of Berlin, 2001.

[Wir71] Niklaus Wirth, *Program development by stepwise refinement*, Commun. ACM **14** (1971), no. 4, 221–227.

[XC05] Manuela Xavier and Ana Cavalcanti, *Mechanised refinement of procedures*, Simpósio Brasileiro de Métodos Formais (SBMF) (2005).

[Y. 85] Y. Gurevich, *A New Thesis*, Abstracts, American Mathematical Society (1985), 317.