UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Especificação Formal de Jogos de Inteligência Artificial

Eliseu César Miguel

Orientador: Vladimir Oliveira Di Iorio
Co-orientador: Roberto da Silva Bigonha

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

Belo Horizonte
8 de outubro de 2004

UNIVERSIDADE FEDERAL DE MINAS GERAIS

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

30 anos

FOLHA DE APROVAÇÃO

Especificação Formal de Jogos de Inteligência Artificial

ELISEU CÉSAR MIGUEL

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. VLADIMIR OLIVEIRA DI IORIO - Orientador
Departamento de Informática - UFV

Prof. ROBERTO DA SILVA BIGONHA - Co-orientador
Departamento de Ciência da Computação - UFMG

Prof. ALCIONE DE PAIVA OLIVEIRA
Departamento de Informática - UFV

Profa MARIA LUIZA D'ALMEIDA SANCHEZ
Departamento de Engenharia de Telecomunicações - UFF

Profa MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 08 de outubro de 2004.

Dedicatória

Aos meus pais José Miguel e Therezinha Martins
e à minha tia Adelaide Martins Cunha Lopes.

Agradecimentos

Gostaria de agradecer, antes de mais nada, a Deus, que me concedeu a nobre oportunidade de dedicar aos estudos, me oferecendo um ambiente familiar que sempre me apoiou.

Agradecimentos especiais ao meu orientador Vladimir, que sempre conduziu esta pesquisa com sabedoria, companheirismo e muita paciência ao lidar com minhas teimosias.

À minha família que sempre primou pelos estudos, dando significativa importância a todas as conquistas que meus irmãos e eu alcançamos. A meu pai José Miguel, minha Mãe Therezinha Martins, minha tia Adelaide e meus irmãos, agradecimentos especiais pela cobrança e partes do financiamento nos momentos de escassez.

Aos professores do Departamento de Informática da UFV que sempre nos ofereceram um ambiente sadio para estudos e desenvolvimento, principalmente ao professor Alcione de Paiva Oliveira que em mim depositou tamanha confiança.

À minha querida Danielle Paiva Rangel que, ao meu lado, compartilhou bons e maus momentos durante meu trabalho. Sempre compreensiva, nunca deixou de me auxiliar com sua atenção e carinho.

Aos fucionários do Departamento de Informática da UFV: Eliana, Altino, Mariza, Paulinho e o Maurício que, com presteza, sempre dispensaram tratamento com carinho e seriedade a todos alunos deste departamento.

Ao Professor Roberto da Silva Bigonha e à Professora Mariza Andrade da Silva Bigonha pelo apoio e sugestões que tanto contribuíram para transpor os obstáculos surgidos durante o desenvolvimento do trabalho.

Agradecimentos especiais à UFMG pela oportunidade oferecida ao firmar o convênio DCC/UFMG com o DPI/UFV que me proporcionou toda estrutura para o desenvolvimento deste trabalho.

A todos os amigos da turma de mestrado e laboratório, principalmente ao Gustavo e ao Balbino, com os quais muitas noites dividi estudando Projeto e Análise de Algoritmos. Também ao amigo João com quem tanto discuti os macetes do Latex.

À professora Maria Luiza D'Almeida Sanchez pela significativa participação na avaliação do trabalho.

Ao amigo e conselheiro Vinícius Maciel que, mesmo distante, sempre desprendeu parte do seu tempo com importantes sugestões sobre o trabalho.

Finalmente, ao grupo Sepultura, que veio se apresentar no Brasil em um momento em que tanto trabalhavamos oferecendo, assim, uma oportunidade de descontração. Realizado em uma fazenda, boas recordações do show e do amigo Gustavo Willam "desmaiado" no pasto.

Resumo

Desde o surgimento das primeiras pesquisas na área de inteligência artificial, os jogos vêm sendo usados como objeto de estudo. Além de despertar o interesse das pessoas tanto no lazer como no meio acadêmico, os jogos são mecanismos para a aplicação das técnicas de inteligência artificial, que os vê como problemas de difícil solução. Contudo, a formalização das regras que descrevem esses jogos não recebem atenção especial no momento em que o problema é proposto. Geralmente formalizadas em linguagens lógicas ou até mesmo em linguagens naturais, o entendimento das regras dos jogos pode ser vago ou até mesmo ambígüo.

As Máquinas de Estado Abstratas (ASM) constituem um conceito expressivo e elegante para modelagem matemática de sistemas dinâmicos discretos. Aplicadas com sucesso em vários tipos de sistemas, as ASM são descritas sob um rigoroso embasamento matemático, o que torna suas especificações mais precisas e confiáveis.

Fazendo uso do formalismo das Maquinas de Estado Abstratas(ASM) e suas linguagens de programação, este trabalho descreve uma excelente alternativa para se formalizar, com clareza e rigor matemático, as regras dos jogos de computadores e o comportamento dos agentes inteligentes envolvidos no ambiente desses jogos.

Abstract

Computer games are one of the oldest areas of research in Artificial Intelligence. Most people are interested in games just for fun, but they are relevant also for academic purposes. Some games are difficult to be solved. When writing programs that provide a solution for these games, it may be necessary to apply sophisticated Artificial Intelligence techniques. In order to produce correct solutions, it is important to understand exactly the rules of the game. But when a new game is proposed, it is not unusual to present its rules in natural language. This lack of formalization may result in difficulties for understanding correctly the rules.

Abstract State Machines (ASM) are an expressive and elegant formalization method used in modelling dynamic discrete systems. With semantics rigorously defined, ASM have been applied succesfully on the formalization of several kind of systems.

This work shows that ASM is an excellent alternative to give a clear and precise specification of the rules of computer games. Using ASM-based programming languages, examples of formalization of the behaviour of intelligent agents in computer games are presented.

Sumário

Lista de Figuras

Capítulo 1

Introdução

O uso de jogos de computadores para o desenvolvimento da inteligência artificial não é novidade. Em 1950, o primeiro programa de jogo de xadrez foi escrito por Claude Shannon e Alan Turing. Com isso, um computador que joga xadrez passou a ser uma prova de que uma máquina pode executar tarefas que exigem "inteligência" [46].

Na área de ensino de inteligência artificial(IA), os jogos de computadores também são usados como dispositivo para estímulo aos alunos no estudo e aprendizagem das técnicas que a IA oferece. Jogos como o xadrez [16, 46], o jogo de damas [43], o MasterMind [7, 20, 23] e o jogo da velha [17, 46], dentre muitos outros, ganharam implementações diversificadas, passando a ser, além de um instrumento de ensino, uma forma de comparar as heurísticas oferecidas pela IA.

Muitos jogos são objetos de estudo interessantes para a IA por não permitirem soluções simples. O xadrez, por exemplo, tem um fator de ramificação na ordem de 35, e as partidas chegam até a 50 jogadas com freqüência, o que exige uma árvore de busca de 35^{100}nós [46]. Assim, o jogo de xadrez, dentre outros, pode ser uma alternativa para se estudar os métodos de pesquisa em árvores como, também, forçar os desenvolvedores a utilizarem com destreza os recursos computacionais.

Para especificar as regras dos jogos e o comportamento dos agentes que interagem nos jogos, o paradigma lógico vem sendo usado há muito tempo, como nos exemplos apresentados em [46]. Já em sala de aulas, os professores recorrem, também, às linguagens naturais para descrever os ambientes dos jogos quando propõem jogos de computadores em atividades pedagógicas.

Uma possível conseqüência do uso da linguagem natural na descrição dos jogos é o surgimento de dúvidas e ambigüidade no entendimento dos sistemas por parte dos alunos. Além disso, a linguagem natural não oferece padronização ao descrever as regras dos jogos.

Pensando nisso, identificar e utilizar um modelo formal que permita especificar os jogos de computadores gerando regras claras e sem ambigüidade pode ser, também, uma importante opção para o ensino de inteligência artificial. Além de oferecer facilidade no aprendizado, é importante que o modelo formal seja definido sob um forte

embasamento matemático, o que tornaria as especificações mais precisas.

O desenvolvimento deste trabalho tem como objetivo estudar a utilização do formalismo das Máquinas de Estado Abstratas (ASM, do inglês Abstract State Machines)[29, 30, 51] na especificação formal das regras de jogos de inteligência artificial. Criadas com o objetivo de simular algoritmos em uma maneira direta, no nível de abstração desejado, as ASM provêm recursos suficientes para se especificar os jogos, além de serem de fácil aprendizagem.

Na Seção 1.1, a relação entre jogos de computadores e a inteligência artificial é tratada com mais detalhes. Na Seção 1.2, são descritos os objetivos do trabalho. A Seção 1.4 exibe a organização do trabalho com uma breve descrição dos conteúdos dos capítulos seguintes.

1.1 Jogos e Inteligência Artificial

Esta seção aborda os conceitos importantes dos jogos de computadores e de inteligência artificial estabelecendo uma relação entre os dois. Inicialmente, na Seção 1.1.1, pode-se encontrar a definição para *jogos de inteligência artificial*, que é de suma importância para o trabalho, além de vários exemplos de jogos importantes para a IA. A Seção 1.1.2 discute os conceitos dos agentes inteligentes. Com os agentes, torna-se possível implementar soluções para muitos jogos de computador. Finalmente, a Seção 1.1.3 descreve os recursos que um ambiente para se desenvolver os jogos de computadores deve oferecer.

1.1.1 Jogos de Inteligência Artificial

Atualmente, existe um grande número de jogos de computadores comerciais que usam técnicas de inteligência artificial para especificar o comportamento dos agentes que disputam esses jogos. Esses agentes são chamados de *bots*[49]. Alguns destes jogos possuem *bots* especificados com características fixas para que sejam controlados por pessoas. Jogos disputados por pessoas que controlam agentes atiradores, em inglês first-person shooter(FPS), podem ser incluídos nesta categoria. Os jogos FPS são jogos 3D de ação em que o jogador move-se em vários cenários e coleciona diferentes armas para destruir os adversários. Um exemplo é o jogo *Doom II* [44]. Para o presente trabalho, o interesse concentra-se em outro tipo de jogos, em que os *bots* são controlados por programas implementados pelos oponentes. Esses jogos fazem usos de técnicas de inteligência artificial para implementar soluções para seus *bots* e são conhecidos como jogos de inteligência artificial [53].

Entretanto, alguns dos modernos jogos FPS também fazem uso da IA em seus agentes. Um exemplo é o jogo *Half-Life* [27] que oferece um kit de implementação de seus bots na linguagem C++. Alguns jogos, apesar de considerados de entretenimento, têm somente *bots* programados, não permitindo jogadores humanos. Um

exemplo é o jogo *AI Wars* [24], que dispõe de uma linguagem que mistura comandos especializados com recursos básicos de programação.

Outro projeto interessante, o *Robocode* [39], oferece um ambiente para disputas de *robôs tanques de batalha* que podem ser implementados em Java. Um dos propósitos do projeto é oferecer um ambiente para o ensino da linguagem Java. No caso da disciplina de inteligência artificial ministrada na UFV, o trabalho [42] discute as vantagens alcançadas ao se incluir os jogos de computadores como atividade acadêmica. Uma das plataformas que dão suporte a essa novidade na disciplina é o projeto *Robocode* [39].

O projeto GameBots [2, 38] foi desenvolvido para educação e pesquisa em inteligência artificial. Para isso, foi criado um ambiente de teste para sistemas multiagentes usando uma extensão do jogo *Unreal Tournament* [25]. Ao contrário de outros jogos de IA, Gamebots não define apenas um único ambiente de execução. Oferece uma grande variação de regras e ambientes que podem ser estendidos de várias formas, usando a linguagem "script" *UnrealScript* [48], baseada em C++. A comunicação entre o servidor do jogo e os agentes é feita via *sockets*, permitindo que os agentes sejam programados em diversas linguagens. Exemplos usando Java e Soar [45] podem ser apreciados.

Este trabalho tem características similares às do projeto GameBots, descrito na Seção 1.1.1, visto que os jogos propostos não têm tarefas predefinidas para seus agentes. Porém, nossa abordagem é mais geral, uma vez que nós podemos definir qualquer tipo de jogo. Os jogos propostos não são restritos a um ambiente ou estilo predefinido. No projeto Gamebots, os jogos podem ser estendidos e novos jogos podem ser criados usando *Unreal Scritpt*, mas de forma limitada. Para o entendimento das regras dos jogos propostos, é necessário conhecer a linguagem de *script*, que não foi desenvolvida para ser um bom método de especificação formal. Nós acreditamos que as Máquinas de Estado Abstratas [29, 30, 51] são uma elegante solução para este problema. A semântica das regras poderá ficar clara se elas forem bem especificadas em ASM.

1.1.2 Agentes em Inteligência Artificial

Ao longo dos anos, a área de inteligência artificial (IA) desenvolveu inúmeras técnicas de resolução de problemas e representação do conhecimento no intuito de solucionar problemas para os quais não existe uma solução algorítmica ou, pelo menos, uma solução algorítmica de complexidade computacional aceitável. No entanto, faltava, até recentemente, um arcabouço (*framework*) que agrupasse todas essas técnicas, permitindo uma forma organizada de projeto e análise das soluções. O arcabouço surgido para preencher essa lacuna foi a definição de soluções usando *agentes* ou *sistemas de agentes* [19].

Agentes possuem diversas definições dentro de IA. Umas das mais abrangentes é a proposta por Russel e Norvig [46] que diz que *um agente é qualquer coisa que*

pode ser vista como percebendo seu ambiente por meio de sensores e atuando sobre esse ambiente por meio de atuadores. Com essa definição, é possível enquadrar um grande número de entidades como um agente.

Contudo, segundo os mesmos autores, a IA está interessada em um tipo particular de agentes, capazes de realizar ações racionais, denominados *agentes racionais ideais*. Eles definem um agente racional ideal da seguinte forma: *para cada possível seqüência de percepções, um agente racional ideal deve fazer a ação que espera-se maximizar sua medida de desempenho, com base nas evidências fornecidas pela seqüência de percepções e pelo conhecimento embutido no agente.*

Russel e Norvig também classificam os agentes segundo sua arquitetura nos seguintes tipos, listados em ordem de complexidade de implementação:

agente reflexivo ou reativo: é o mais simples de todos e simplesmente reage às alterações do ambiente segundo suas regras internas;

agente com estado: é um agente reflexivo que leva em consideração o histórico das percepções;

agente com objetivo: possui característica adicional de selecionar a regra que mais o aproxima de cumprir um determinado objetivo;

agente baseado em utilidade: procura maximizar sua medida de utilidade.

Este trabalho enfatiza a especificação de agentes reativos e agentes com estado. Apesar de sua simplicidade, esses tipos de agentes possuem uma ampla gama de aplicações, principalmente em ambientes dinâmicos, que demandam respostas rápidas, como é o caso de ambientes de jogos e simuladores.

1.1.3 Um Ambiente para Jogos de Inteligência Artificial

Um ambiente para desenvolvimento de jogos envolvendo agentes racionais poderia ter os seguintes componentes:

1. Um formalismo que, associado a uma linguagem de programação, permita descrever tanto as regras dos jogos, como o comportamento dos agentes que participam dos jogos e outros problemas de inteligência artificial.

2. Mecanismos para assegurar que os agentes irão obedecer as regras descritas.

3. Uma interface gráfica animada para exibir as transformações associadas aos jogos.

Relacionado ao Item 1, o paradigma Lógico vem sendo muito usado ao longo do tempo em implementações de jogos de computadores. Uma linguagem desse paradigma que pode ser citada é o Prolog, que foi utilizada na implementação do jogo MasterMind em [20].

As Máquinas de Estado Abstradas (ASM) [29, 30, 51], descritas com mais detalhes no Capítulo 2, surgem como um formalismo matemático que pode também oferecer todos os recursos para atender as exigências do Item 1. Introduzidas por Yuri Gurevich em [29, 30], as ASM constituem um conceito expressivo e elegante para modelagem matemática de sistemas dinâmicos discretos. Aplicadas com sucesso em vários tipos de sistemas, como relatado pela literatura [5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 28, 32, 33, 34, 52], as ASM são descritas sob um rigoroso embasamento matemático, o que torna suas especificações mais precisas e confiáveis.

Apesar de ASM não ser uma linguagem de programação, várias linguagens baseadas nesse formalismo, como descrito no Capítulo 3, podem ser utilizadas para gerar especificações formais e executáveis das regras descritas em ASM. Assim, é possível realizar testes práticos e visualização dos resultados da execução das regras formalizadas em ASM com uso de interfaces gráficas acopladas às linguagens de programação.

Já em relação ao Item 2, para assegurar que os agentes irão seguir as regras dos jogos pré-estabelecidas, pode-se usar mecanismos de encapsulamento e controle de visibilidade que algumas linguagens do paradigma de ASM oferecem. Um exemplo é a linguagem Machĩna [50], usada com sucesso em [19], que oferece recursos como a utilização de *módulos* que permitem definir objetos e serviços públicos ou privados, podendo, assim, satisfazer as exigências do Item 2.

Finalmente, para satisfazer o Item 3 descrito no início desta seção, contamos com algumas facilidades de integração de bibliotecas gráficas com determinadas linguagens de programação baseadas em ASM. Um exemplo é a biblioteca gratuita *Clanlib* [41], desenvolvida na linguagem C++. Outro exemplo, o *.Net* da Microsoft oferece facilidades para o desenvolvimento gráfico como suporte para a linguagem AsmL [21, 22], também da Microsoft.

1.2 Objetivos do Trabalho

O objetivo central deste trabalho de dissertação de mestrado é estudar a aplicação do modelo formal de Máquinas de Estado Abstratas no desenvolvimento de jogos de computadores e demais aplicações que envolvam agentes e inteligência artificial.

Fatores importantes, como a avaliação da facilidade e clareza oferecidas pelo modelo ao descrever os sistemas, bem como a clareza nas demonstrações formais baseadas neste modelo, quando aplicado ao domínio específico de sistemas de agentes e Inteligência Artificial, são objetos do estudo.

Para o desenvolvimento do trabalho, localizar e dominar ferramentas que viabilizem a implementação de exemplos de agentes inteligentes e sua iteração com o seu ambiente é de suma importância. Procurou-se utilizar ferramentas que permitissem a implementação dos sistemas estudados, ou parte destes, tornando assim os resultados mais confiáveis.

Ao descrever as regras de jogos e o comportamento dos agentes envolvidos, o presente trabalho enfatiza o uso de ASM puro, citando as possibilidades de mapeamentos para linguagens como é o caso do MasterMind com especificações executáveis em AsmL, no Capítulo 4. Isso permite atribuir um perfil mais genérico e abstrato aos modelos, tornando as especificações executáveis apenas um mecanismo para testes e validação dos sistemas especificados em ASM.

Com o desenvolvimento do presente trabalho, espera-se mostrar que a utilização de ASM na especificação das regras dos jogos de computadores, assim como no comportamento dos agentes envolvidos, pode eliminar a possível ambigüidade que as descrições realizadas em linguagens naturais pode permitir. Espera-se mostrar, também, que o uso de ASM nos jogos de inteligência artificial oferece clareza e facilidade de compreensão às regras especificadas.

Outro objetivo do trabalho é cercar e apontar os pontos positivos e negativos das linguagens escolhidas, assim como os pontos positivos e negativos do próprio modelo ASM, quando usados para esse fim. Um ponto importante a ser avaliado, por exemplo, são os mecanismos de visibilidade oferecidos pelas linguagens de especificação escolhidas para descrever o comportamento de agentes. É importante que as linguagens de programação usadas pelo trabalho forcem os agentes a seguir rigorosamente as regras impostas por um determinado jogo.

Sugestões de novas construções sintáticas para as linguagens escolhidas são sugeridas com o objetivo de tornar mais simples e sucintas as especificações desenvolvidas. Assim, é possível que dêm origem, em trabalhos futuros, a novas versões das linguagens aqui utilizadas, principalmente para a linguagem Machĩna [50].

1.3 Metodologia de Desenvolvimento

Nesta seção, definem-se as atividades desenvolvidas durante o trabalho.

- **Estudo do formalismo ASM** - A primeira atividade que se destaca e tem relevante importância é o estudo do formalismo ASM, bem como exemplos de casos específicos em que o formalismo foi aplicado com sucesso. Essa tarefa é viabilizada por uma gama de trabalhos disponíveis na literatura [29, 30, 51].

- **Seleção e estudo dos casos de estudo**- Esta atividade se resume na escolha e busca de soluções para os exemplos que estejam no domínio proposto - agentes e técnicas de IA - e que sejam adequados à aplicação do modelo ASM.

- **Avaliação das ferramentas e linguagens de programação disponíveis baseadas no paradigma ASM** - Nesta atividade, o foco de interesse é selecionar e dominar ferramentas adequadas para codificação dos sistemas escolhidos ou parte destes. Um ponto importante a ser analisado é a disponibilidade de compiladores ou interpretadores oferecidos pelas ferramentas, assim como as

possíveis bibliotecas gráficas necessárias para se exibir a evolução dos agentes no ambiente.

- **Formalização e implementação dos casos de estudo utilizando o formalismo ASM** - Com esta etapa, os sistemas escolhidos ganham implementações nas linguagens escolhidas, usando-se o modelo ASM. Além da formalização, essa etapa engloba a codificação destes, utilizando-se as ferramentas selecionadas na realização da atividade anterior.

- **Avaliação da facilidade e clareza oferecidas pelo modelo ASM no desenvolvimento das formalizações dos sistemas propostos** - O interesse, nessa atividade, é avaliar os critérios citados, facilidade e clareza, na descrição das regras que compõem o ambiente onde os agentes irão reagir, bem como as regras que especificam o comportamento de cada agente que irá interagir com o ambiente e com outros agentes.

- **Desenvolvimento de provas de propriedades relevantes dos sistemas** - Algumas propriedades dos sistemas estudados são de grande relevância para o completo entendimento destes. Assim, esta atividade cerca essas propriedades relevantes e as valida com o desenvolvimento de provas formais baseadas no modelo ASM.

- **Avaliação da facilidade no desenvolvimento de provas formais usando o modelo ASM** - Por ter uma sintaxe similar às linguagens baseadas no paradigma imperativo, esta etapa do trabalho deixa claro que o entendimento de especificações implementadas usando-se o formalismo ASM são naturais e de fácil compreensão aos profissionais acostumados com o paradigma imperativo.

1.4 A Organização do Trabalho

O Capítulo 2 apresenta, de forma resumida, os conceitos do formalismo das Máquinas de Estado Abstratas.

O Capítulo 3 faz uma avaliação de algumas linguagens de programação baseadas em ASM e que podem ser utilizadas na especificação e geração de códigos executáveis de jogos de computadores formalizados em ASM.

O Capítulo 4 apresenta uma aplicação do formalismo de ASM para o jogo *O MasterMind* [7, 20, 23]. Inicialmente, as regras do sistema *MasterMind*, incluindo uma solução para o agente que desvenda o código secreto, são formalizadas em ASM puro. Com uso da linguagem Asml [21, 22], o jogo *MasterMind* ganha uma implementação.

O Capítulo 5 é um bom exemplo de como as ASM se comportam na especificação de sistemas multiagentes. Com uso da linguagem Machĩna [50], o jogo *O Mundo Wumpus* [46], em sua nova versão multiagentes, é especificado e discutido

em detalhes. Demonstrações de algumas propriedades interessantes são apresentadas, mostrando o poder e facilidades oferecidas pelo formalismo ASM para este fim.

Finalmente, o Capítulo 6 conclui o trabalho, descrevendo suas contribuições e os trabalhos que poderão vir daqui por diante.

Nos apêndices A e B, as especificações dos exemplos dos Capítulos 4 e 5 podem ser apreciados.

Capítulo 2

Máquinas de Estado Abstratas

Com base no tutorial de ASM apresentado em [51], este capítulo apresenta o modelo de especificação formal ASM, anteriormente conhecido como *Evolving Algebras*.

Máquinas de Estado Abstratas (ASM, do inglês *Abstract State Machines*), introduzidas por Yuri Gurevich em [29, 30], constituem um conceito expressivo e elegante para modelagem matemática de sistemas dinâmicos discretos.

As ASM possuem recursos para modelar interação do programa com o "mundo exterior", possibilitando formalizar as ações do ambiente no qual o sistema está inserido. A metodologia de ASM provê recursos expressivos para especificar a semântica operacional de sistemas dinâmicos discretos, em um nível de abstração natural e de uma maneira direta e essencialmente livre de codificação [18]. Com isso, tem-se por objetivo diminuir a distância que há entre modelos formais de computação e métodos práticos de especificação. A metodologia utiliza conceitos simples e bem conhecidos, o que facilita a leitura e a escrita de especificações de sistemas.

Na literatura, há vários exemplos de utilização de ASM na especificação formal de sistemas, dentre os quais podemos citar: arquiteturas [8, 9, 12], linguagens de programação [13, 14, 15, 32, 52], sistemas distribuídos [5, 6, 10, 33] e de tempo real [28, 34], entre outros [11].

A Seção 2.1 apresenta, de maneira informal, uma breve introdução ao formalismo das Máquinas de Estado Abstratas, dando enfoque ao modelo seqüencial. A Seção 2.2 apresenta o modelo formal de ASM. O texto se baseia em [51] e [19], incluindo como contribuição adicional a Definição 15.

Uma abordagem mais completa sobre ASM, além das definições de ASM para sistemas distribuídos, podem ser encontradas em [18, 30, 51].

2.1 Introdução ao Modelo ASM Seqüencial

A *assinatura* de uma ASM seqüencial $\mathcal{A}$ é uma coleção finita de nomes de funções, cada uma com uma aridade fixa. Um *estado* de $\mathcal{A}$ é um conjunto não vazio, o

superuniverso, junto com interpretações dos nomes da assinatura em funções sobre os elementos do superuniverso. As interpretações são denominadas *funções básicas* do estado. À medida que $\mathcal{A}$ muda de estado, as interpretações dos nomes podem ser alteradas, mas o superuniverso se mantém inalterado. Funções básicas que não se alteram são chamadas de *funções estáticas*, enquanto as demais são chamadas *funções dinâmicas*.

Formalmente, supondo um superuniverso X, uma função básica de aridade r é uma função $X^r \rightarrow X$. Quando $r = 0$, a função é denominada *elemento distinto*. O superuniverso sempre contém os elementos distintos *true*, *false* e *undef*, definidos como *constantes lógicas*. O elemento *undef* é utilizado para representar funções parciais, por exemplo, $f(\overline{a}) = undef$ significa que f é indefinida para a tupla $\overline{a}$. Uma relação *r-ária* sobre X pode ser vista como uma função $X^r \rightarrow \{true,false\}$. Um *universo* U é um tipo especial de função básica: uma relação unária geralmente identificada pelo conjunto dos elementos x tais que $U(x) = true$, i.e., $\{x : U(x)\}$.

Um programa de $\mathcal{A}$ é uma regra de transição, que pode ser composta por regras básicas e não básicas. As regras básicas são: *regra de atualização*, *construtor de bloco* e *construtor condicional*.

Uma regra de atualização tem o formato $f(\overline{t}) := t_0$, onde f é o nome de uma função da assinatura de $\mathcal{A}$, $\overline{t}$ é uma tupla de termos cujo tamanho é igual à aridade de f e t_0 é outro termo. Os termos não possuem variáveis livres e são construídos recursivamente usando-se nomes de elementos distintos e aplicação do nome de uma função a outros termos. De maneira informal, a semântica é a seguinte: a tupla $\overline{t}$ é avaliada, e o valor da função básica f aplicada à tupla é alterado para o valor da avaliação de t_0. Ou seja, o nome f passa a ter uma nova interpretação.

Um construtor condicional tem genericamente a forma

if g_0 **then** R_0 **elseif** g_1 **then** R_1 ... **elseif** g_k **then** R_k **endif**

Sua semântica é a seguinte: uma regra R_i, $0 \leq i \leq k$, será executada se os termos booleanos $g_0, ..., g_{i-1}$ são avaliados para *false* e g_i é avaliado para *true*.

Uma regra bloco da forma

$$R \equiv R_1, \ldots, R_n$$

é um conjunto de regras. Sua semântica é a seguinte: o estado formado por R é o resultado da execução de todas as regras R_i em paralelo. Por exemplo, a execução da regra bloco

$$f(0) := 15, f(1) := 2$$

produz um novo estado, no qual o valor da função f, no ponto 0, é 15 e, no ponto 1, é 2.

Em uma regra de bloco, se uma atualização contradiz outra, uma escolha não determinística é realizada.

Uma especificação ASM contém a definição de um estado inicial, S_0, e uma regra, R, que define as mudanças de estado. A execução de uma especificação é uma

seqüência de estados $\langle S_n : n \geq 0 \rangle$, onde um estado S_i é obtido executando a regra R em S_{i-1}.

Um aspecto importante que deve ser modelado na especificação de sistemas é que, em geral, eles são afetados pelo ambiente. O ambiente se manifesta por meio de algumas funções básicas, chamadas *funções externas*. Um exemplo típico de função externa é uma entrada fornecida pelo usuário. Pode-se pensar em funções externas como *oráculos*, tais que, a especificação fornece argumentos e o oráculo fornece o resultado [30].

Além das regras básicas mostradas, há também as regras que utilizam variáveis[1]. Em ASM, variáveis são utilizadas para modelar paralelismo, não-determinismo e a "criação"de novos elementos. As regras que utilizam variáveis são as regras *import*, *choose* e *var*.

Uma regra *import* é da forma

$$\textbf{import } v \ R_0 \textbf{ endimport},$$

onde v é uma variável e R_0 é uma regra. O efeito dessa regra é executar R_0 em um estado em que a variável v está associada a um valor importado de um universo especial chamado *Reserve*. Este universo está contido em X, o superuniverso dos estados da máquina, e contém todos os elementos que serão importados.

Em geral, regras *import* são utilizadas para estender universos, isto é, adicionar novos elementos aos universos. Assim, regras da forma

$$\textbf{import } v \ U(v) := \mathit{true}, R_0 \textbf{ endimport}$$

onde U é um nome de universo e R_0 é uma regra, podem ser escritas utilizando a seguinte regra *extend*:

$$\textbf{extend } U \textbf{ with } v \ R_0 \textbf{ endextend}.$$

Uma regra *choose* é da forma

$$\textbf{choose } v \textbf{ in } U \textbf{ satisfying } g \ R_0 \textbf{ endchoose}$$

onde v é uma variável, U é o nome de um universo finito, g é um termo booleano e R_0 é uma regra. O efeito desta regra é executar a regra R_0 em um estado no qual a variável v está associada a um valor pertencente ao universo U. Este valor é escolhido de maneira não-determística e satisfaz a guarda g.

Uma regra *var* é do tipo:

$$\textbf{var } v \textbf{ ranges over } U \ R_0 \textbf{ endvar}$$

onde v é uma variável, U é um universo finito e R_0 é uma regra. O efeito dessa regra é criar uma instância de R_0 para cada elemento pertencente ao universo U.

[1]Variáveis são símbolos que podem denotar elementos do superuniverso.

Em cada instância de R_0, a variável v está associada ao elemento correspondente de U. Após criadas as instâncias, todas são executadas em paralelo.

Para simplificar as especificações, vamos supor que conjuntos como o dos números inteiros, dos racionais, das listas e dos conjuntos estejam contidos no superuniverso de um estado.

Uma execução de um programa de $\mathcal{A}$ é uma seqüência de estados, onde o estado seguinte é obtido a partir do anterior através da execução da regra de transição do programa. A maioria das implementações de ASM determina o final da execução quando o disparo da regra de transição não produz nenhuma atualização.

Para permitir uma interface com o mundo externo, o modelo ASM oferece *funções externas*. Uma função externa não precisa retornar necessariamente um mesmo valor para chamadas com os mesmos parâmetros, se essas forem disparadas em passos diferentes de uma execução.

2.2 O Modelo Formal de ASM

2.2.1 Definição da Máquina

ASM são sistemas de transição que especificam computações cujos estados são álgebras [18]. Os universos de álgebras que formam os estados da computação constituem o superuniverso da ASM.

Definição 1 *(Vocabulário) Um* vocabulário Υ *é uma coleção finita de nomes de função ou relação, cada nome com uma aridade fixa. Estão presentes em todo vocabulário: o sinal de igualdade, true, false e undef e os operadores booleanos usuais.*

A função *undef* é utilizada para modelar funções parciais.

Definição 2 *(Estado) Um* estado S *é uma álgebra [26], dada por:*

- *um vocabulário* Υ*, o vocabulário do estado* S*;*
- *um conjunto* X*, denominado o superuniverso de* S*, no qual estão contidos os conjuntos dos números inteiros, dos valores lógicos, dos números racionais, das cadeias de caracteres, das listas, das tuplas e das partes de conjuntos;*
- *uma função* $Val : \Upsilon \rightarrow X^* \rightarrow X$*, que fornece a interpretação dos nomes de funções pertencentes a* Υ *em funções de* $X^* \rightarrow X$*.*

Para os conjuntos do superuniverso, estão definidas as operações usuais (como, no caso dos inteiros, adição, multiplicação, etc.).

A noção de estado é importante, pois cada estado constitui um passo na execução da máquina.

Definição 3 *(Funções Estáticas e Funções Dinâmicas) Uma função é* dinâmica *se puder sofrer atualizações, isto é, se durante uma mudança de estado, a sua interpretação puder ser modificada em alguns pontos. Caso contrário, dizemos que a função é* estática.

Os conceitos de funções estáticas e dinâmicas são importantes nas definições de regras e atualização, dadas abaixo. Basicamente, o caráter dinâmico do sistema é modelado pelas alterações, de estado para estado, na interpretação de funções dinâmicas.

Definição 4 *(Termo)* Termos *são definidos recursivamente como:*

- *uma variável é um termo;*
- *um nome de função ou relação de zero argumento é um termo;*
- *se f é um nome de função ou relação r-ária, $r > 0$, e $\bar{x} = (x_1, \ldots, x_r)$ é uma tupla onde cada $x_i, 1 \leq i \leq r$ é um termo, então $f(\bar{x})$ é um termo.*

Termos sem variáveis são interpretados, em um estado S, da maneira descrita a seguir:

- se f é um nome de função ou relação de zero argumento, sua interpretação é $Val_S(f)$;
- se f é um nome de função r-ária e $(x_1, \ldots, x_r)$ é uma tupla de comprimento r, então

$$Val_S(f(x_1, \ldots, x_r)) = Val_S(f)(Val_S(x_1), \ldots, Val_S(x_r))$$

A interpretação de termos que contêm variáveis será mostrada na Seção 2.2.2, onde definiremos as regras não-básicas, isto é, as regras que utilizam variáveis.

Definição 5 *(Endereço) Um* endereço *em um estado $S = (\Upsilon, X, Val)$ é um par $(f, \bar{x})$, onde $f \in \Upsilon$ é um nome de função dinâmica, e $\bar{x} \in X^*$ é uma tupla cujo comprimento é igual à aridade de f.*

Definição 6 *(Atualização) Uma* atualização *em um estado $S = (\Upsilon, X, Val)$ é um par (l, y), onde l é um endereço em S e $y \in X$.*

A noção de atualização é importante para a definição de regras e disparo de regras. Para cada regra definimos um conjunto de atualizações, que conterá os pontos em que haverá mudança na interpretação de alguns nomes de funções dinâmicas no estado seguinte.

Definição 7 *(Consistência de um Conjunto de Atualizações) O conjunto de atualizações* Δ *é* consistente, *se*

$$((f, \bar{x}), y_1) \in \Delta \wedge ((f, \bar{x}), y_2) \in \Delta \Rightarrow y_1 = y_2,$$

isto é, se não houver, em Δ, *duas atualizações diferentes para o mesmo endereço.*

Definição 8 *(Regra)* Regras *são definidas recursivamente por:*

- *A regra de atualização* $R \equiv f(\bar{t}) := t_0$ *é uma regra. Nesta expressão,* f *é um nome de função dinâmica,* t_0 *é um termo e* $\bar{t}$ *é uma tupla de termos cujo comprimento é equivalente à aridade de* f. *Se* f *é um nome de relação, então* t_0 *deve ser booleano. Definiremos o conjunto de atualizações de* R *em um estado* S, $Updates(R, S)$, *como o conjunto* $\{(l, y)\}$, *onde* $l = (f, Val_S(\bar{t}))$ *e* $y = Val_S(t_0)$.

- *Um bloco de regras* $R \equiv R_1, \ldots, R_k$ *é uma regra; o seu conjunto de atualizações é dado por*

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

isto significa que, para disparar um bloco de regras, disparam-se todas as regras que o compõem simultaneamente. Note que a ordem de $R_1, \ldots, R_k$ *não é relevante na execução do bloco.*

- *Se* k *é natural,* $g_0, \ldots, g_k$ *são termos booleanos, e* $R_0, \ldots, R_k$ *são regras, então*

$$R \equiv \textbf{ if } g_0 \textbf{ then } R_0 \textbf{ elseif } g_1 \textbf{ then } R_1 \ldots \textbf{ elseif } g_k \textbf{ then } R_k \textbf{ endif}$$

é uma regra. O conjunto de atualizações desta regra é dado por

$$Updates(R, S) = \begin{cases} Updates(R_i, S) & se\ Val_S(g_i) \wedge \forall j < i : \neg Val_S(g_j) \\ \emptyset & se\ \forall i : \neg g_i \end{cases}$$

Estas regras são conhecidas como regras básicas.

Estas regras são as mais simples de ASM e permitem a especificação de apenas alguns tipos de sistemas reais. Outros tipos de regras mais poderosas serão mostrados nas seções subseqüentes. Observe que **não existe** composição seqüencial de regras, isto é, uma regra da forma

$$R_1; R_2$$

onde primeiro executa-se R_1 e em seguida R_2. Isto porque um programa em ASM descreve somente um passo do algoritmo. A composição seqüencial pode gerar estruturas de execução muito complexas, o que pode tornar o raciocínio sobre a especificação muito mais difícil [31].

Definição 9 *(Disparo de uma Regra) O* disparo de uma regra R *em um estado* S *produz um novo estado* S', *tal que, se* $Updates(R, S)$ *for consistente, então*

$$Val_{S'}(f, \bar{x}) = \begin{cases} y & se\ ((f)(\bar{x}), y) \in Updates(R, S) \\ Val_S(f)(\bar{x}) & caso\ contrário. \end{cases}$$

Se o conjunto $Updates(R, S)$ *não for consistente, então o efeito do disparo de* R *em* S *será nulo, isto é, o estado* S' *resultante será igual a* S.
Definiremos $Fire(R, S)$ *como o estado resultado do disparo de* R *em* S.

Outra abordagem utilizada para tratar conjuntos de atualização inconsistentes é fazer uma escolha não-determinística da atualização que será disparada [35].

Definição 10 *(Especificação ASM) Uma* especificação ASM *é uma tupla* $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$, *onde*

- $\Upsilon = \Upsilon_0 \cup \Upsilon_e$ *é um vocabulário composto pela união de um vocabulário pré-definido,* Υ_0, *e um vocabulário especificado pelo usuário,* Υ_e. *O vocabulário pré-definido* Υ_0 *contém os nomes de relação Boolean, Integer, String, List, Set, que serão interpretados, respectivamente, como os universos dos valores lógicos, dos números inteiros, das cadeias de caracteres, das listas e das partes de conjuntos.* Υ_0 *contém também as operações usuais sobre estes conjuntos;*

- $\mathcal{A}$ *é um conjunto de* Υ*-álgebras ou estados* S, *cada um consistindo no vocabulário* Υ *e um conjunto* X *(o superuniverso de* S*) de funções, que são as interpretações em* X *dos nomes de funções de zero argumento pertencentes a* Υ. *Um nome de função de aridade* r, $r > 0$, *é interpretado como uma função de* $X^r \rightarrow X$. *O conjunto* X *é comum a todos os estados pertencentes a* $\mathcal{A}$;

- $S_0 \in \mathcal{A}$ *é o estado inicial, onde as interpretações de alguns nomes de funções são dadas; nomes de funções cujas interpretações não são dadas em* S_0 *são interpretados como undef;*

- $\mathcal{P}$ *é uma regra que descreve as modificações das interpretações de nomes de funções de uma álgebra (estado) para outra.*

O vocabulário de uma ASM reflete apenas os recursos verdadeiramente invariantes de um algoritmo, em vez de detalhes de um estado em particular.

O estado inicial S_0 pode ser definido através de qualquer formalismo existente, em particular, via regras de transição do modelo ASM. Pode-se utilizar também mecanismos como lambda cálculo, funções recursivas, entre outros.

Definição 11 *(Execução de uma Especificação ASM) A* execução de uma especificação ASM $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$ *é uma seqüência* $\mathcal{S} = \langle S_n : n \geq 0 \rangle$ *de estados pertencentes a* $\mathcal{A}$, *onde* $S_{n+1} = Fire(\mathcal{P}, S_n)$, *isto é,* S_{n+1} *é obtido a partir de* S_n, *disparando a regra* $\mathcal{P}$ *em* S_n.

Na maioria dos sistemas dinâmicos discretos, a execução pode ser influenciada pelo ambiente. Intuitivamente, um ambiente ativo é um agente externo. Desta maneira, utilizamos *funções externas* na modelagem do ambiente no qual o sistema está inserido. Além disso, utilizam-se funções externas para:

- prover ocultamento de informações – qualquer característica do sistema, cujo funcionamento não é relevante no entendimento da especificação, pode ser modelada por meio de funções externas;
- inserir não-determinismo ao modelo – considerando que as ações do ambiente acontecem de maneira não-determinística, utilizamos funções externas para descrever implicitamente o não-determinismo; na Seção 2.2.2 mostraremos uma construção para especificarmos explicitamente o não-determinismo.

2.2.2 Regras Não-Básicas

Nesta seção apresentaremos as regras de transição não-básicas, que são as regras que utilizam variáveis. A introdução de variáveis ao modelo ASM dá maior poder às definições, permitindo, por exemplo, importar elementos, o que permite estender universos, e especificar o não-determinismo de sistemas.

Para a regra *import* que definiremos a seguir, consideraremos que existe um universo de nome *Reserve*, contido no superuniverso do estado. Este universo contém os elementos que serão importados. Cada regra *import* retira um elemento de *Reserve*.

Definição 12 *(Regra* Import*) A regra* import *é uma regra da forma*

$$R \equiv \textbf{import } v \ R_0 \textbf{ endimport},$$

onde v é uma variável e R_0 é uma regra.

A semântica desta regra é a seguinte: escolhe-se um elemento a do universo Reserve e associa-o à variável v. Desta forma, executa-se a regra R_0, no estado S_a, que é uma expansão do estado S, no qual interpretamos v como a. O conjunto de atualizações para esta regra é dado por

$$Updates(R, S) = \{((Reserve, a), false)\} \cup Updates(R_0, S_a).$$

O efeito desta regra é "criar"um novo elemento. Em geral é utilizada com uma regra da forma $U(v) := true$, onde U é um nome de universo, para inserir o novo elemento ao universo U, modelando uma expansão de U.

Definição 13 *(Regra* Var*) A regra* var *é uma regra da forma*

$$R \equiv \textbf{var } v \textbf{ ranges over } U \ R_0 \textbf{ endvar},$$

onde v é uma variável, U é o nome de um universo finito e R_0 é uma regra.

A semântica desta regra é a seguinte: para cada elemento x de U, executamos a regra R_0 no estado S_x, que é uma expansão do estado S, tal que a variável v é interpretada como x. Esta execução cria o conjunto de atualizações $Updates(R_0, S_x)$. O efeito da regra é a união de todos os conjuntos $Updates(R_0, S_x)$, tal que $x \in U$, isto é,

$$Updates(R, S) = \bigcup_{x \in U} Updates(R_0, S_x).$$

O efeito desta regra é criar uma instância de R_0 para cada elemento pertencente ao universo U. Em cada instância de R_0, a variável v está associada ao elemento correspondente de U. Após criadas as instâncias, todas são executadas em paralelo. Esta regra é usada para modelar paralelismo síncrono.

Definição 14 *(Regra* Choose*)*
A regra choose *é uma regra da forma*

choose v **in** U **satisfying** g R_0 **endchoose**,

onde v é uma variável, U é o nome de um universo finito, g é um termo booleano e R_0 é uma regra. O efeito desta regra é executar a regra R_0 em um estado S_a, no qual a variável v está associada a um elemento $a \in U$. O elemento a é escolhido de maneira não-determinística e torna a guarda g verdadeira.

Algoritmos não-determinísticos são úteis, por exemplo, como descrições (especificações) em alto nível de algoritmos "reais". Esta regra é usada para modelar explicitamente o não-determinismo. Outra maneira de modelar o não-determinismo é permitir que a inconsistência de conjuntos de atualizações sejam resolvidas escolhendo-se não-deterministicamente qual atualização disparar. Entretanto, esta abordagem pode dificultar a leitura de uma especificação e a demonstração de propriedades sobre ela.

Definição 15 *(*Type*) Para dar mais clareza às especificações implementadas em ASM, uma opção é estender ASM incluindo a construção Type. O uso de Type permitirá a composição de tipos já definidos. Um exemplo importante é a definição de registros em ASM. Essa nova construção pode ser mapeada diretamente para as definições básicas de ASM, como demonstrado no exemplo a seguir.*

Type $NovoTipo$ **is** ($IdenA : TipoA$; $IdenB : TipoB$; ...)

pode ser mapeado para o seguinte conjunto de definições:

Universe $NovoTipo$
$novoTipo\_IdenA$: $NovoTipo \rightarrow TipoA$
$novoTipo\_IdenB$: $NovoTipo \rightarrow TipoB$
...

Onde novoTipo_IdenA e novoTipo_IdenB são funções.

Exemplo 1 *(Registro Posição)*

O exemplo ilustra o uso da construção Type *em ASM para a geração de um registro simples capaz de formar uma dupla de inteiros referentes a uma posição no plano.*

Type Posicao is $(x : Integer; y : Integer)$

que pode ser usado como:
$p : Posicao$
...
$p.x := 10,$
$p.y := 20$
onde x *e* y *permitem acesso aos "componentes do registro"* p.

A estrutura Posicao pode ser mapeada para a seguinte estrutura definida em ASM:

Universe Posicao
$Posicao\_x : Posicao \rightarrow Integer$
$Posicao\_y : Posicao \rightarrow Integer$

e seu uso é mapeado como:
extend Posicao with p
$Posicao\_x(p) := 10,$
$Posicao\_y(p) := 20$
endextend

2.3 Conclusões

Este capítulo apresentou o modelo de especificação formal ASM e seus principais conceitos.

Uma observação importante sobre os programas em ASM é a similaridade destes com os programas implementados em linguagens imperativas comuns. Uma característica que os diferencia é a forma de representar um estado, por meio de funções. Outra característica que difere as especificações ASM dos programas imperativos comuns é a ausência de um mecanismo explícito de iteração. Em ASM, a iteração é realizada com a aplicação repetida da regra de transição sobre o estado corrente, até que um ponto fixo seja atingido e determine o fim da execução. Finalmente, devemos ressaltar que ASM implementa a execução paralela das regras.

Apesar de o modelo ASM ser bastante geral e não especificar uma série de características necessárias a uma definição de uma linguagem de programação real,

várias linguagens de programação baseadas em ASM emergiram nos últimos anos. Assim, tornou-se possível gerar especificações executáveis de sistemas formalizados em ASM. Além de tornar ASM mais completo, as linguagens de programação e seus ambientes de programação permitem a validação desses sistemas tornando-os mais interativos e servem como estímulo para se alcançar os objetivos do trabalho que aqui são propostos.

As linguagens de programação baseadas em ASM são discutidas no Capítulo 3.

Capítulo 3

As Linguagens Baseadas em ASM

3.1 Introdução

Atualmente, pesquisadores e empresas vêm investindo no desenvolvimento de ferramentas para especificação e formalização de algoritmos baseados no Paradigma de ASM. Várias ferramentas podem ser encontradas na Internet [36].

A implementação com a utilização de ferramentas baseadas em ASM dos sistemas *MasterMind*, Capítulo 4 e *O Mundo Wumpus*, Capítulo 5, aqui sugeridos, é uma forma de tornar as especificações mais consistentes. Isso porque permite realizar a execução de módulos ou dos sistemas como um todo e avaliar, assim, os resultados gerados pela execução.

Para isso, foi realizada uma avaliação das ferramentas disponíveis com o objetivo de se escolher uma que tivesse características necessárias para dar suporte à implementação dos referidos sistemas. Dentre as ferramentas mais importantes, podemos citar:

- Machĭna [50]
- AsmGofer[47]
- XASM [3]
- AsmL [22, 21]

Em seguida, descreve-se de forma compacta as linguagens avaliadas, detalhando apenas para as linguagems Machĭna e AsmL que implementam os exemplos *Master-Mind* e O *Mundo Wumpus*, respectivamente.

3.2 A Linguagem AsmGofer

Baseada em Gofer [37] e nas bibliotecas TCL/TK [1], AsmGofer é uma extensão da linguagem funcional Gofer para especificações em ASM. Vários sistemas foram

formalizados, com sucesso, em AsmGofer e podem ser apreciados em [15, 47].

A utilização da linguagem AsmGofer para implementação dos jogos no presente trabalho foi desmotivada pelo pouco suporte oferecido pelo tutorial da linguagem [47] disponibilizado pelos autores, que era simplificado e contendo apenas conceitos básicos. Além disso, não foi localizado um manual de referência da linguagem, o que poderia permitir o aprendizado e, conseqüentemente, a utilização desta linguagem.

Em relação às bibliotecas gráficas, a literatura cita apenas a integração do AsmGofer com a biblioteca TCL. Para os propósitos do trabalho, essa característica torna-se um fator limitador, principalmente pela escassez de manuais bem elaborados que possam auxiliar o trabalho dos desenvolvedores.

3.3 A Linguagem Xasm

Criada por Matthias Anlauff [3], a linguagem Xasm oferece uma série de extensões para a linguagem das Máquinas de Estado Abstratas, além de comandos básicos ASM.

Xasm possui um compilador para a linguagem C, o que permite executar as especificações sem a necessidade de um interpretador.

Uma característica interessante de Xasm é a possibilidade de criar abstrações de funções. Essas funções podem ser executadas em vários passos, independente da especificação em que estão inseridas. Isso é uma característica fundamental para gerenciamento de aplicações volumosas.

Outra característica interessante de Xasm é a possibilidade de especificar uma gramática que descreve uma linguagem L qualquer. Um analisador léxico e um analisador sintático são automaticamente gerados. Um programa escrito em L pode ser lido e convertido para um formato interno, para então ser processado. Isso facilita a utilização em uma das aplicações mais importantes do modelo ASM, que é a descrição da semântica de linguagens de programação.

3.4 A Linguagem Machĭna

Machĭna [50] possui bom manual de referência e tutorial com especificação da linguagem. De fácil instalação e geração de código intermediário em C++, permite a integração direta com bibliotecas gráficas como, por exemplo, a biblioteca gratuita Clanlib [41]. Exemplos de especificações em Machĭna podem ser apreciados em [19].

Contudo, a utilização de Máquina pode ser comprometida uma vez que a linguagem ainda não dispõe de um compilador estável. Outra ferramenta ainda não existente é um depurador, importante para o programador e capaz de aumentar a produtividade na programação ao auxiliar nas correções de erros de lógica e aprendizagem da linguagem.

Acreditamos que, em pouco tempo, esses problemas serão solucionados e que Machĭna passará a oferecer todas ferramentas necessárias para implementações baseadas em ASM.

Algumas características de Machĭna a tornam uma linguagem muito importante para o trabalho. Além de fortemente baseada em ASM, permite implementações de sistemas multiagentes, ao contrário de AsmL.

Outra característica importante em Machĭna é seu mecanismo de visibilidade. Com a palavra reservada *public*, o programador determina a permissão de escrita e ou leitura de valores armazenados em memória, além da permissão na execução de funções ou ações implementadas nos módulos.

A seção seguinte descreve as características de um *Módulo* em um programa escrito em Machĭna.

3.4.1 Módulos

A principal estrutura sintática de Machĭna é um módulo. Um módulo contém uma regra de transição e declarações de tipos, ações e funções ASM. Uma declaração qualificada com *public* permite sua visibilidade fora do módulo.

Para executar uma regra de transição de um módulo, é necessário criar dinamicamente um agente do módulo. A exceção é o módulo principal e obrigatório, chamado *modulo Machina*.

Podemos dividir um módulo em duas seções. A primeira seção, definida pela palavra chave *import*, permite a importação de nomes públicos, como funções, ações e estruturas de outros módulos. Na segunda seção são feitas as declarações de funções, ações e regras de transição do módulo. Esta seção pode ser subdividida nas seguintes subseções:

Tipos

Com o uso da palavra reservada *type*, esta subseção permite a criação de novos tipos a partir de tipos predefinidos ou tipos gerados por composições. Tipos Compostos podem ser, em Machĭna: listas, conjuntos, tuplas e agentes. Em relação ao tipo "Agent", este refere-se a um agente genérico, porém, o tipo "Agent of M" define um agente baseado em um módulo de nome "M".

O tipo "Agent" é de grande importância para o presente trabalho, pois permite a codificação de sistemas multiagentes, permitindo a codificação do exemplo "O Mundo Wumpus", descrito no Capítulo 5. Além disso, o suporte a sistemas multiagentes em Machĭna é claro e bem definido, o que pode ajudar no desenvolvimento de outras linguagens, como AsmL, descrita na Seção 3.5, que, até o momento, não dispõe desse recurso.

Funções

Machĭna permite a especificação de quatro tipos de função, que são: estáticas, dinâmicas, derivadas e externas. Em um módulo Machĭna, as palavras reservadas *static*, *dynamic*, *derived* e *external* distinguem essas funções, respectivamente.

Veja, abaixo, um resumo de cada uma:

- **funções estáticas** são funções ASM que não sofrem atualizações. Seus valores são sempre constantes durante a execução e não fazem chamadas a funções dinâmicas ou externas.
- **funções dinâmicas** são funções ASM que podem sofrer atualizações.
- **funções derivadas** são funções estáticas que fazem chamadas a funções dinâmicas e externas.
- **funções externas** são funções definidas em outros sistemas. Machĭna permite a utilização de funções implementadas em outros ambientes, onde, normalmente, estas funções são implementadas em outras linguagens, como C++ e Java.

Ações

Para facilitar a implementação de um módulo e permitir a abstração de regras de transição, Machĭna define ações. Assim como as funções, as ações também podem ser externas.

Em uma especificação feita em Machĭna, uma ação pode ser vista como um único passo do ponto de vista de quem disparou sua execução. Isso faz com que o uso de ações permita abstrair regras de transição com mais clareza, além de gerar um código mais organizado e de mais fácil manutenção.

Uma característica interessante que a linguagem reserva às ações é o recurso definido pela palavra reservada *loop*. Com esse construtor, o programador prende a execução da máquina de estados na ação até que um comando *return* informe o fim da execução da ação.

No final de um módulo Machĭna, além destas seções, as regras de transição implementadas pelas ações são referenciadas. Veja mais detalhes na Seção 3.4.2.

3.4.2 A Chamada das Regras de Transição de um Módulo

Uma característica interessante na linguagem Machĭna é a possibilidade de se executar uma regra de transição apenas uma vez ao iniciar as transições. Para isso, em um módulo, basta usar a palavra reservada *init* para criar uma seção onde as regras de transição são chamadas.

```
module "NomeDoMódulo"
  import:
     ...
  initial state:
     type
        ...
     static
        ...
     dynamic
        ...
     derived
        ...
     external
        ...
     action
        ...
  init
     ...
  transition
     ...
  ...
end
```

Figura 3.1: Módulo Machĭna.

Com a palavra reservada *transition*, cria-se uma seção onde são chamadas as regras de transição do módulo. As regras desta seção serão executadas a cada estado até que a máquina execute a regra *stop*, que finaliza toda execução.

A Figura 3.1 exibe a estrutura de um módulo em Machĭna.

É importante lembrar que, como Machĭna é fortemente baseada em ASM, uma implementação nesta linguagem força o programador a seguir o paradigma de ASM, o que pode ser um bom caminho para entender ASM e iniciar seu aprendizado. Para mais informações sobre a linguagem Machĭna, utilize seu manual de referêcia [50].

3.5 A Linguagem AsmL

O sucesso na instalação do compilador da linguagem AsmL, desenvolvida pela Microsoft, e o acesso ao manual de referência e ao tutorial da linguagem estimularam a escolha desta linguagem para formalização dos sistemas aqui avaliados. Além disso, um grande número de exemplos disponibilizados pelos autores facilitou a aprendizagem da linguagem, o que é importante quando se investe em um novo paradigma.

Apesar de alguns problemas terem sido observados nessa versão de AsmL, como por exemplo a inexistência de um depurador, essa ferramenta tornou possível a especificação, em ASM, do sistema *O MasterMind*, descrito no Capítulo 4.

3.5.1 Características de AsmL

A linguagem possui recursos avançados, descritos a seguir, permitindo, assim, muito poder ao programador. Uma característica importante a ser citada é o controle de visibilidade, que utiliza artifícios para tornar os elementos públicos, privados ou protegidos nas especificações, o que é uma importante característica das linguagens orientadas a objetos, o ocultamento de informações.

Outra característica importante é a integração da linguagem AsmL com a plataforma .Net da Microsoft, facilitando, assim, a implementação de Interfaces Gráficas e o uso de funções e outros elementos do .Net nas formalizações desenvolvidas em AsmL.

A seção seguinte descreve alguns recursos importantes que possibilitaram que a linguagem AsmL fosse utilizada neste trabalho.

3.5.2 Recursos da Linguagem AsmL

Vários recursos importantes da linguagem AsmL podem ser citados. Estes facilitam a implementação e aumentam o poder de expressão de AsmL. Dentre os recursos, podemos citar:

- O comando **Step**. Com o comando Step, o programador força a mudança de estado e, conseqüentemente, as atualizações previstas pela regra disparada. Apesar de facilitar a codificação, este recurso pode comprometer a clareza do código gerado.

- Funções pré-definidas. Um vasto conjunto de funções internas estão disponíveis para programadores de AsmL. Como exemplo, *max* e *min* permitem retornar o máximo ou mínimo em um conjunto qualquer de elementos enumeráveis definido pelo usuário em um programa AsmL. As funções podem ser encontradas em [21, 22]

- Existência de **classes**. A existência de classes possibilita uma organização sofisticada do código. Essa característica permite ao programador a criação de *objetos*, aqui ligeiramente diferentes dos convencionais, como ocorre nas linguagens orientadas a objetos. Em AsmL, objetos atuam apenas como identidades que distinguem uma variável de outra. São elementos abstratos que não possuem estado ou outra característica qualquer. Ao contrário, apenas a transição da máquina provê a noção dinâmica de estado ao modelo.

- Possibilidade de inicialização de variáveis ou constantes a partir de métodos pré-definidos pela linguagem ou implementados pelo programador. Essa característica permite clareza ao código ou, no caso de instanciação de constantes, permite que sejam inicializadas por resultados de computações provenientes de códigos existentes.

  $$\textbf{var } valorFitness \ = \ (max\ x \mid x\ in\ Indices(controle))$$

  O valor máximo dentre os índices do mapeamento dado pelo conjunto *controle* é atribuído à função *valorFitness*.

- A codificação de programas em AsmL é muito parecida com as codificações em outras linguagens de paradigmas tradicionais como o imperativo e, principalmente, o orientado a objetos.

  Iniciar o aprendizado de um paradigma com o uso de uma nova linguagem de programação com características marcantes de outros paradigmas pode ser uma armadilha. No caso de AsmL, ao contrário de minimizar o tempo de aprendizagem, o programador pode não abstrair os conceitos importantes do formalismo de ASM e comprometer sua abstração, já que AsmL tem muitos recursos e características do paradigma de orientação a objetos.

Durante a implementação do *MasterMind* vários problemas foram evidenciados. Para solucionar esses problemas, optou-se por criar novas estruturas ou até mesmo utilizar funções auxiliares que permitissem a completude da implementação. A Seção 3.5.3 discute esses problemas.

3.5.3 Problemas na Implementação

Como na maioria das linguagens de programação que estão amadurecendo, alguns problemas foram encontrados durante a implementação do jogo *MasterMind* na linguagem AsmL. Dentre os mais importantes, podemos citar:

- O não funcionamento correto dos construtores de classes. Durante a implementação do sistema MasterMind, foi observado que a chamada de métodos dentro do construtor, que dependiam de objetos instanciados dentro do próprio construtor, não permitia a utilização destes objetos durante a execução. Veja o exemplo abaixo:

```
//construtor da classe AgenteJogador
1  public AgenteJogador(objetoTabuleiro as Tabuleiro)
2  step
3      tab = objetoTabuleiro
4  step
5      inicio()
```

  Observe que o construtor *AgenteJogador* recebe, durante a criação do objeto, uma referência para um objeto da classe *Tabuleiro* referenciado por *objetoTabuleiro* e inicializa seu objeto interno *tab*, também do tipo *Tabuleiro*. Em seguida, é chamada a execução do método *inicio()*, que utiliza o objeto *tab* em sua computação.

  É interessante notar que, apesar de a compilação ser realizada com sucesso, a execução é interrompida quando o método *inicio()* tenta acessar métodos do objeto *tab*. Como os comandos das linhas 3 e 5 são separados pelo comando *Step*, o erro sugere que a atualização necessária para a referência correta do objeto *tab* no método *inicio()* não ocorreu como prevista.

- Escopo do tipo estruturado *enum*. Foi observada uma divergência relacionda ao escopo de um tipo estruturado *enum* em relação aos outros tipos definidos na implementação. Observe o código abaixo.

```
enum StatusDoJogo
    JogoEmAndamento
    JogoFinalizado
    JogoGanho
    JogoPerdido
publicclass Basica
    public type Cromossomo = Map of Integer to Integer
    public structure DuplaIntPegs
```

PCertas as Integer

PErradas as Integer

Observe que os tipos *Cromossomo* e *DuplaIntPegs* estão definidos no escopo da classe *Basica* e podem ser utilizados pelas classes derivadas de *Basica* por serem públicos. Porém, a tentativa de se introduzir o tipo *StatusDoJogo* no escopo da classe *Basica* não permitiu, às classes derivadas, sua visualização.

- Impossibilidade de realizar *atualizações Locais*. Diferentemente do padrão das linguagens de programação, que permitem passagem de parâmetros por referência ou por valor absoluto, dentre outros, observou-se em AsmL que os parâmetros dos métodos não permitiam atualização local, ou seja, seu valor pode somente ser **acessado** no escopo dos métodos. Essa característica inviabiliza a implementação de algumas estruturas desejadas, principalmente quando deseja-se atualizar algum atributo da classe passado como parâmetro. Soluções para contornar esse problema podem ser viabilizadas com a utilização de variáveis globais chamadas diretamente no escopo dos métodos, o que torna o método menos versátil. Outra opção é fazer com que o método retorne o valor local da variável desejada. Essa estratégia pode ser incoveniente quando as atualizações dentro de um método alteram mais de uma variável. Isso exige uma estrutura especial para agrupá-las antes do retorno do método, o que compromete a clareza e objetividade do código. Veja o exemplo que se segue:

```
protected zeraPontuacao(...)
  var pop as cadeiaCromossomo = popula
  step
    forall i in [0..tamanhoPop]
      pop(i).pontuacao := 0
  step
    return pop
```

No Exemplo, o método zeraPontuação tem como tarefa fazer com que todos os indivíduos da população recebam, para o campo pontuação, o valor zero. A atualização no parâmetro *popula* não é permitida no escopo de *zeraPontuação*. Assim, a variável local *pop* recebe o valor do parâmetro *popula* e, após as atualizações necessárias, é retornada pelo método, em **return** *pop* para atualização a variável global que a recebe.

- A inexistência de *undef* que, apesar de substituído pelo valor *null* do tipo *Null*, não tem o mesmo comportamento descrito na Teoria de ASM. Veja o exemplo:

```
if(ponto in Indices(controle)) then
```

controle(ponto) := *controle(pontos)* + 1

else

controle(ponto) := 1

No exemplo, *Indices(...)* é uma função pré-definida que gera um conjunto de índices, ou seja, dos valores domínio, em um mapeamento qualquer. Nesse caso, *controle* é um mapeamento de inteiros para inteiros e o teste averigua se o valor inteiro *ponto* pertence ao domínio de *controle*. O teste é realizado apenas para inicializar o mapeamento *controle* caso não exista o valor *ponto* no domínio da função *controle*. Esse teste poderia ser mais claro se permitisse a sintaxe:

if (*controle(ponto)* <> *undef*) **then**

porém, essa estrutura não é oferecida pela linguagem AsmL.

- Não funcionamento de algumas estruturas internas como esperado e definido. Um caso interessante ocorrido durante implementações em AsmL foi observado com os comandos **foreach** e **forall**.

  O comando *foreach* é usado quando se deseja realizar uma computação em todos elementos de um conjunto de forma seqüencial, permitindo que o próximo elemento a ser computado perceba a alteração realizada no computado anteriormente. Já o comando *forall* difere do anterior por realizar a mesma computação em paralelo, não permitindo que um elemento que esteja sofrendo uma computação veja a computação realizada em outro elemento do conjunto.

  Observe o exemplo, onde *indice* é, a cada passo, um valor presente na lista definida por [0..tamanhoPopulacao]:

  1 **step foreach** *indice in* [0..*tamanhoPopulacao*]

  2 *pop(indice).pontuacao* := *funcaoFitness(pop(indice), falsoCodigo)*

  Observe que o comando na linha"2" realiza operações de atribuição em cada indivídio da população, neste instante referenciados pela função *pop*, de forma singular, ou seja, uma atualização em um indivíduo não depende da atualização feita no indivíduo anterior. Isso sugere que o código poderia ser implementado de forma que todas atualizações ocorressem em paralelo, com o uso do comando *forall*, como descrito abaixo:

  1 **forall** *indice in* {0..*tamanhoPopulacao*}

  2 *pop(indice).pontuacao* := *funcaoFitness(pop(indice), falsoCodigo)*

  Nesse caso, *indice* não é mais referente aos valores contidos em uma lista seqüencial como antes, e sim representa os valores definidos pelo conjunto {0..tamanhoPopulacao}. Porém, apesar de compilar com sucesso, esse código gera erro em tempo de execução, o que contradiz a teoria de ASM.

- Erros de compilação com alguns tipos pré-definidos pela linguagem. Observou-se o surgimento de erros de compilação com tipos pré-definidos pela linguagem, como é o caso do tipo *Integer?*. Este é definido, em AsmL, como o tipo Integer que, além dos valores do conjunto numérico dos inteiros, como em qualquer outra linguagem, também pode receber o valor *null* do tipo *Null*, definido em AsmL. Essa inclusão do valor *null* tem como objetivo permitir que uma função ou mapeamento do tipo *Integer?* possa receber o valor *undef*. Assim, não compilando, essa opção torna-se indisponível e faz com que a linguagem fique restrita em relação ao poder de computação proposto por ASM.

- A inicialização de objetos. A inicialização de objetos em AsmL, se comparada a outras linguagens, tem uma estrutura pouco convencional. Veja o exemplo abaixo:

  classe *Estrutura*

  var X **as** *Integer* $= 0$

  var Y **as** *Integer*

  var S **as** *String*

  $Main()$

  $Object$ $=$ **new** $Estrutura(1, "Celular")$

  É interessante notar que, no exemplo, o construtor da classe *Estrutura* está implícito, visto que não foi implementado pelo programador. O construtor recebe dois parâmetros. O primeiro, um valor *inteiro* e o segundo uma *string*. Ambos serão valores atribuídos às variáveis Y e S que, diferentemente de X, não foram inicializadas.

- Inexistência de um Depurador para auxílio na implementação de sistemas. É de se esperar que, com a evolução da linguagem e de suas ferramentas, surja um depurador para AsmL que, até o momento, não conta com essa ferramenta. Mesmo assim, é possível avaliar a evolução das variáveis, funções e mapeamentos com o uso de mensagens enviadas ao console, como exemplo.

- Dificuldades em se separar códigos em módulos. AsmL não exige uma estrutura bem definida de classes e métodos como nas linguagens orientadas a objetos convencionais. Isso dá liberdade ao programador de projetar suas classes, permitindo que sejam definidos métodos fora do escopo de qualquer classe. Esse processo pode comprometer a organização estrutural da implementação, já que métodos isolados não pertencem a um objeto específico. Veja o exemplo

  $imprime(z\ as\ Integer)$

  $WriteLine(z)$

```
classe Estrutura
  var X as Integer = 0
  var Y as Integer
  var S as String
Main()
  Object = new Estrutura(1, "Celular")
  imprime(Object.X)
```

Como podemos ver, o método *imprime()* está definido no mesmo módulo onde todo programa se encontra. Não foi, porém, definido para *imprime()* um escopo específico, podendo esse ser usado no programa principal *Main()* ou no escopo da classe *Estrutura*.

- A impossibilidade de se passar um parâmetro por referência e poder alterá-lo localmente torna o código menos simples, pois exige a utilização de variáveis locais que devem ser retornadas pela função ou a utilização das variáveis globais no escopo local. É interessante resaltar que, para os propósitos do trabalho, essa desvantagem da linguagem quanto à programação acaba trazendo mais clareza ao código gerado, já que as variáveis que sofrerão alterações estão nitidamente vinculadas aos valores de retorno da função. Porém, caso AsmL oferecesse, também, a passagem de parâmetro por referência, o programador encontraria mais felxibilidade ao desenvolver suas implementações.

- A não definição de sistemas multiagente em AsmL. O Manual de Referências de AsmL [22] refere-se a *MultiAgentes* como um recurso a ser definido no futuro. Essa característica não impossibilita, necessariamente, a implementação de sistemas multiagentes em AsmL. Porém, para que o programador desenvolva tais sistemas, é de se esperar que a implementação exija muito esforço para controle dos agentes envolvidos no sistema.

- A possibilidade de se usar *polimorfismo*, o que diminui a clareza da formalização. Por permitir a utilização de polimorfismo, AsmL pode deixar dúvidas quanto a formalização de sistemas implementados nessa linguagem. Isso ocorre pelo fato de não ser possível ter certeza de qual método será executado quando mais de um possui a mesma assinatura, visto que, para tratar o polimorfismo, o método usado é definido em tempo de execução.

- Quebra de ortogonalidade. Em muitos casos, a possibilidade de se codificar os mesmos elementos de forma diferente pode trazer dificuldade no aprendizado da linguagem e, até mesmo, na tomada de decisão na hora da codificação. Um exemplo pode ser ilustrado com as palavras chaves *initially* e *var*, que têm a mesma semântica, sendo que recomenda-se a utilização de *initially* na inicialização de variáveis no início da execução, como é o caso do método

Main() ou outros métodos, enquanto o uso do *var* seria para atributos de classes e outros. Porém , nada impede a utilização de ambos ao mesmo tempo como no exemplo:

public *imprime*()

initially k *as Integer* $= 6$

var x *as Integer* $= 6$

$WriteLine(k + x)$

Outro exemplo observado foi a possibilidade de se usar o comando condicional *if* ora com a cláusula *then* e ora sem essa cláusula, sem trazer problemas de compilação ou execução.

3.6 Conclusões

A escolha de AsmL como linguagem de programação para o sistema MasterMind deu-se pelo estágio avançado em que a linguagem AsmL se encontra em relação às ferramentas necessárias ao suporte à programação como, por exemplo, compilador, materiais de referência e exemplos de sistemas nela implementados.

Já a linguagem Machĭna, apesar de dispor de um compilador ainda instável, é uma linguagem desenvolvida fortemente baseada no paradigma de ASM e, principalmente, permite implementações de sistemas multiagentes [50] e um excelente controle de visibilidade. No Capítulo 5 é sugerida uma versão multiagentes para *O Mundo Wumpus*, o que motivou a utilização de Machĭna na especificação desse sistema.

Contudo, a escolha das linguagens AsmL e Machĭna não caracteriza as demais linguagens como "piores" ou "melhores", principalmente pelo fato de todas possuírem exemplos implementados com sucesso e citados pela literatura.

Nesse contexto, é interessante citar que a especificação da linguagem Machĭna em relação aos sistemas multiagentes é bem definida e descrita nos manuais da linguagem [50]. Esse fator, por exemplo, distancia Machĭna de Asml, visto que esta ainda não dispõe de construções desta natureza.

Apesar de Asml ter apresentado problemas durante a implementação dos sistemas aqui sugeridos, como descrito na Seção 3.5.3, esses problemas não inviabilizaram a implementação de tais sistemas. Ao contrário, apenas forçaram a tomada de decisões na codificação que fugiram ligeiramente do esperado pelo programador.

Capítulo 4

O MasterMind

4.1 Introdução

O Mastermind é um jogo que exige raciocínio lógico. É constituído de um tabuleiro e um conjunto de pinos com cores preestabelecidas. Dois agentes interagem no jogo: o *tabuleiro* e o *jogador*. Inicialmente, o *tabuleiro* faz uma combinação de cores, gerando um código secreto desconhecido pelo *jogador*. Uma configuração possível para o jogo pode ser definida da seguinte forma: um código secreto formado pela combinação de quatro cores definidas em um conjunto de seis cores possíveis e um total de dez passos para que o *jogador* descubra o código secreto.

Definido o código secreto, o *tabuleiro* irá interagir passo a passo com o *jogador*. Na tentativa de acertar o código secreto, o *jogador* cria uma combinação de cores e a envia ao *tabuleiro* que, em seguida, tem como tarefa calcular o número de cores que estão na posição correta e o número de cores que estão na posição incorreta em relação ao código secreto em seu poder. Essas informações são enviadas ao *jogador* para que este possa elaborar novas tentativas que serão enviadas ao *tabuleiro*. Observe que as informações fornecidas ao *jogador* não permitem inferir quais cores e quais posições estão corretas. Aos acertos de cores em posição correta e em posição incorreta, exibidos pelo agente *tabuleiro* em cada jogada damos os nomes *Pegs Pretos* e *Pegs Brancos*, respectivamente.

Assim, usando um conjunto de estados passados conhecido, o *jogador* poderá fazer novas combinações de cores na tentativa de alcançar seu objetivo, ou seja, obter a *solução* para o MasterMind.

Como pode ser observado, o MasterMind tem características que o tornam um sistema viável para a análise proposta pelo trabalho, como exemplo, estados discretos bem definidos e interação entre agentes, neste caso, o *jogador* e o *tabuleiro*. Além disso, por ser um jogo de raciocínio lógico, sua implementação permite a utilização de técnicas de inteligência artificial na codificação do agente *jogador*, que precisa inferir novas jogadas para vencer o jogo.

Objetivando a clareza na expressão e visualização das formalizações do MasterMind, optou-se por mapear o universo *Cores* para um conjunto de *caracteres*, de mesmo tamanho. Dessa forma, para a configuração sugerida ao MasterMind com um conjunto de seis cores, faremos referência ao universo definido pelo conjunto $Cores = \{B, A, L, V, M, P\}$, considerando a equivalência: B para *branca*, A para *amarela*, L para *laranja*, V para *vermelha*, M para *marrom* e, finalmente, P para *preta*.

Nas próximas seções, descreve-se a formalização do MasterMind com o uso de ASM.

4.2 Formalização do Agente Tabuleiro

O *agente tabuleiro*, descrito na Seção 4.1, destaca-se, no MasterMind, pela importância no contexto do trabalho. Isso se dá pela responsabilidade que este possui em gerir as regras do jogo, não apenas no sentido de utilizá-las para determinar os *Pegs Pretos* e *Pegs Brancos* a cada tentativa do *jogador*, mas também por ser responsável pela contagem de vezes que o *jogador* pode tentar descobrir o código, definindo se este é o ganhador ou o perdedor ao fim do jogo.

É interessante observar que, além dessas atribuições destinadas ao *tabuleiro*, em um ambiente que se propõe o uso de jogos de computadores para o ensino de inteligência artificial, o *tabuleiro* deve ser especificado de forma que as regras do jogo, que são de sua responsabilidade, fiquem claras e de fácil leitura, permitindo que os oponentes que implementarão o *jogador* tenham visão de todo escopo do jogo durante sua especificação, sem dúvidas ou qualquer questionamento.

Porém, a tarefa de especificar essas regras pode não ser tão simples. Veja a seguinte situação: Para o código secreto definido como $Codigo = [B, B, L, P]$, o jogador pretende fazer uma jogada com a tentativa $T = [B, L, B, B]$. Recebida a tentativa T, qual seria a resposta correta que o *tabuleiro* deveria enviar ao jogador? Nesse caso, a cor B contribui para a geração de um ou dois *Pegs Brancos*? Essas perguntas podem não ser respondidas de forma correta caso as regras que determinam o trabalho do *tabuleiro* não estejam bem definidas. A Seção 4.2.1 trata, de forma detalhada, como o *tabuleiro* deve proceder ao receber uma tentativa do *jogador*, fazendo com que essas perguntas sejam respondidas claramente.

Além disso, essas regras não podem estar fora de alcance do *jogador*, que precisa conhecê-las para avaliar o quanto suas tentativas estão longe do código real e, assim, ter o poder de inferência em busca de tentativas cada vez melhores.

Em busca de respostas para as perguntas aqui lançadas, outro problema pode emergir: de posse das regras que definem o trabalho do *tabuleiro*, como formalizá-las para que estas fiquem claras e de fácil compreensão a quem as aprecia?

Na Seção 4.2.1, discutiremos as regras do MasterMind. A Seção 4.2.2, apresenta a formalização em ASM dessas regras.

4.2.1 As Regras do MasterMind

As regras do MasterMind podem ser descritas, em linguagem natural, como:

1. Definição do conjunto de cores. Com essa regra, o *tabuleiro* permite ao *jogador* conhecer o conjunto de cores que permitirá a geração do código secreto que ele deverá descobrir.

2. Definição do tamanho do código. O número máximo de cores que formam o código secreto deve ser definido pelo *tabuleiro* e de conhecimento público no escopo do jogo.

3. Geração do código secreto. Essa regra é a única que exige regras de visibilidade que tornem o código privativo ao *tabuleiro* ou seja, mais ninguém pode ter acesso ao código secreto. Além disso, a geração do código secreto deve ser feita de forma aleatória escolhendo elementos do conjunto de cores previamente estabelecido pelo *tabuleiro*.

4. Controlar o Status do Jogo. Nessa regra, o *tabuleiro* deve determinar quantas tentativas o *jogador* poderá utilizar para descobrir o código secreto. É através dessa regra que o *tabuleiro* define, ao final, se o *jogador* é *perdedor*, caso utilize todas as possíveis jogadas sem acertar o código secreto, ou *ganhador*, caso descubra o código até o limite de jogadas.

5. Geração de Pegs. A cada passo do jogo, definido por cada tentativa do *jogador* em acertar o código secreto, o tabuleiro deve calcular e informar qual foi o número de acerto de cores em posição correta, os Pegs Pretos, e a quantidade de cores que, apesar de pertencerem ao código original, não se encontram na posição correta definida pelo *tabuleiro*, ou seja, os Pegs Brancos.

A formalização das regras mais importantes do jogo será discutida na Seção 4.2.2. Porém, apesar da importância de todas as regras aqui descritas, uma que exige atenção especial é a última listada, referente à geração dos *Peg Pretos* e dos *Peg Brancos*. Isso pode ser justificado pela quantidade de detalhes que envolvem esse procedimento. Com isso, o refinamento desta regra é feito a seguir, antes de sua formalização, também na Seção Seção 4.2.2.

4.2.2 A Formalização das Regras do MasterMind

Avaliação dos Peg Pretos

A estratégia que aqui se descreve para solucionar a geração de *Peg Pretos* e *Peg Brancos* não é única. Porém, optar por uma solução que não estabeleça vínculo entre as avaliações dos *Peg Pretos* e *Peg Brancos* é conveniente para tornar a formalização dessa regra mais clara e elegante.

Para fazer referência ao mapeamento $(Inteiros \rightarrow Cores)$ que forma o código secreto e as tentativas do *jogador*, define-se:

$$TypeCombinacao :: (Integer \rightarrow Cores)$$

tornando mais claro o trabalho de especificação do MasterMind em ASM.

Para descrever a solução para a regra de geração de Pegs, estabelecemos uma situação exemplo onde o código é definido como $Codigo = [B,\ B,\ L,\ P]$ e a tentativa definida como $T = [B,\ L,\ B,\ B]$. Veja a Figura 4.1.

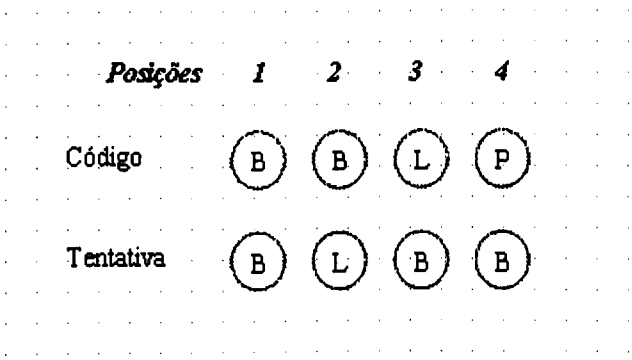


Figura 4.1: Avaliação de Pegs.

Avaliar os *Peg Pretos* é a parte mais simples e intuitiva do problema. No exemplo ilustrado pela Figura 4.1, fica claro que apenas a cor B, localizada na primeira posição da tentativa, é igual à cor de mesma posição no código. Assim, o resultado da avaliação de *Peg Pretos* aplicada à tentativa e ao código da Figura 4.1 deve retornar um valor inteiro *1*.

Para especificar uma função em ASM que descreva a regra de avaliação de *Peg pretos*, definiremos um universo *Indices*, subconjunto dos números inteiros e descrito de forma tal que $Indices = \{x \in Inteiros \mid 1 \leq x \leq NCombina\}$, onde *NCombina* é o tamanho do código secreto e da tentativa.

O vocabulário contém as funções *codigo* e *tentativa*, ambas de aridade 1, além da função f definida como $f :: Inteiro \rightarrow Inteiro$. A avaliação de *f(x)* resultará em 1 quando a comparação da cor de índice x do código for igual à cor de mesmo índice na tentativa e 0, caso contrário.

Abaixo, o algoritmo:

```
var y ranges over Indice
   if codigo(y) = tentativa(y)then
      f(y) := 1
   else
      f(y) := 0
   endif
endvar
```

A aplicação dessa especificação ASM no exemplo da Figura 4.1 resultará em uma configuração onde apenas $f(1) = 1$. Uma vez atualizados os valores em f, resta-nos apenas aplicar uma função *somatorio* para somar todos os valores da imagem de f, que será o resultado da avaliação dos *Peg Pretos*, como descrito abaixo:

$$pegPretos := somatorio(imagem(f))$$

É importante ressaltar que a função de ordem mais alta *imagem* tem como resultado a geração de uma *lista* de elementos representando todos os valores resultantes da aplicação de f a elementos de seu domínio. Assim, caso um elemento esteja repetido na imagem de um conjunto qualquer, sua duplicidade será considerada na lista gerada, diferentemente das funções que geram conjuntos após sua avaliação. Repare que a ordem dos elementos na lista não tem relevância para nossos propósitos.

A utilização da função *somatorio* é um típico caso do uso de abstração, o que pode trazer facilidade ao entendimento de especificações e de códigos em geral. Aproveitamos, então, para ressaltar essa importante característica de ASM. Dada a formalização da regra de geração de *Peg Pretos* em alto nível, fica claro como a regra se comporta, mesmo não estando a função *somatorio* completamente especificada, já que seu entendimento é óbvio.

No final, a função de aridade zero *pegPretos* armazenará o valor da avaliação dos *Peg Pretos* para o exemplo ilustrado na Figura 4.1.

Avaliação de Peg Brancos

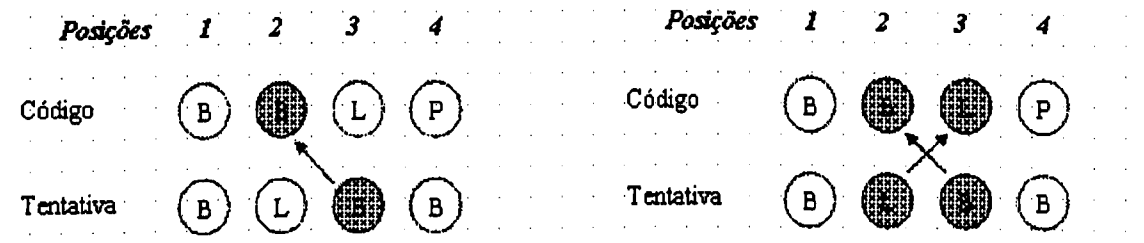
Partimos, agora, para uma situação um pouco mais complexa, que é a avaliação dos *Peg Brancos*.

Para facilitar a compreensão do leitor, trata-se apenas de uma cor do universo *Cores* e, em seguida, amplia-se a solução às demais.

Na Figura 4.1, observa-se duas ocorrências da cor B no código e três ocorrências da mesma na tentativa. Agora, uma pergunta nos ajudará a elaborar uma estratégia para solucionar o problema dos Peg Brancos. Qual poderá ser a contribuição máxima oferecida por uma cor qualquer no valor resultante dos Peg Brancos? É interessante notar que uma determinada cor pode contribuir para os Peg Brancos em, no *máximo*, o valor de sua ocorrência *mínima* entre os conjuntos, seja esta no código ou na tentativa. Isso fica claro quando revista a definição de Peg Brancos, que pode ser definida como "a quantidade de ocorrências de cores repetidas no código e na tentativa, desde que essas ocorrências não sejam de uma mesma cor na mesma posição, o que caracterizaria os Peg Pretos".

Pela definição, também pode ficar claro o porquê da contribuição de uma determinada cor ser, em certos casos, menor que o mínimo de sua ocorrência. Observe que, caso ocorra a participação da cor na geração de Peg Pretos, essa deve ser desconsiderada na geração dos Peg Brancos, o que pode ser visto na repetição da cor B nas posições 1 do código e da tentativa na Figura 4.1.

A Figura 4.2 (a) refere-se à participação da cor B na contagem dos Peg Brancos enquanto a Figura 4.2 (b) é a aplicação da avaliação dos Peg Brancos para todas as cores. O valor resultado dessa avaliação para o exemplo ilustrado na Figura 4.2 é o Inteiro 2, sendo a contribuição da cor B igual a 1, assim como a contribuição da cor L.



(a)Avaliação da cor B. (b)Avaliação de todas as cores.

Figura 4.2: Avaliação de Peg Brancos no MasterMind.

Vejamos, agora, uma possível solução para se especificar um algoritmo em ASM para mapear a ocorrência de todas as cores do universo *Cores* no código e na tentativa, permitindo selecionar os mínimos, o que equivale à soma dos *Peg Brancos* com os *Peg Pretos*, aqui chamado simplesmente de *peg*. Observe que, de posse das funções de aridade 0 *peg* e *pegPretos* poderemos avaliar a função, também de aridade 0, *pegBrancos*.

Seja a função *ocorrencia* :: $Cores \rightarrow (Integer, Integer)$, onde cada elemento desse conjunto representará a quantidade de vezes que uma determinada cor aparece no código e na tentativa, respectivamente pela ordem na dupla de inteiros. Seja, também, a função *conta* capaz de contar o número de ocorrência de uma *cor* na imagem de um mapeamento de Integer para Cores e definida como *conta* :: $Cores \rightarrow (Combinacao) \rightarrow Integer$.

Assim, podemos avaliar a função *ocorrencia* como:

var y **ranges over** $Cores$
$ocorrencia(y) := (conta(y, codigo), conta(y, tentativa))$

A aplicação da função *ocorrencia* para o exemplo da Figura 4.1 produzirá as seguintes interpretações exibidas na Figura 4.3

De posse do conjunto gerado pela função *ocorrencia*, o próximo passo é a geração do conjunto com as ocorrências mínimas de cada cor. Para isso, usaremos a função *minimo* de assinatura *minimo* :: $(Cores \rightarrow Integer)$ responsável por mapear, para

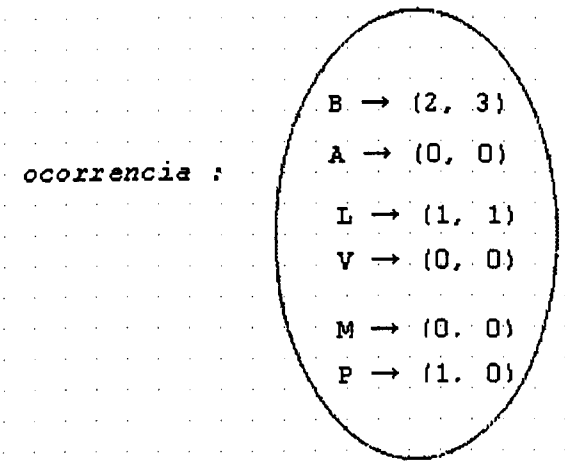


Figura 4.3: Ocorrência das cores no código e tentativa.

cada cor, o valor de sua ocorrência mínima dada pela função *ocorrencia*. Veja o algoritmo:

var y **ranges over** *Cores*
  let $(a, b) = ocorrencia(y)$
  in
    if $a < b$ **then**
      $minimo(y) := a$
    else
      $minimo(y) := b$

A avaliação da função *minimo* para o mapeamento *ocorrencia* da Figura 4.3 pode ser representada pela Figura 4.4

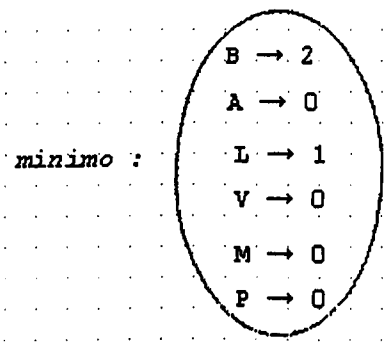


Figura 4.4: Ocorrência mínima das cores entre a tentativa e o código.

Agora, para obter o valor dos *peg*, consideraremos as funções *somatorio* e *imagem*, previamente definidas na Seção 4.2.2

Veja o algoritmo:

$$peg := somatorio(imagem(minimo))$$

Finalmente, para se obter os *pegBrancos* desejados, apenas subtraímos *peg* de *pegPretos* avaliados anteriormente, como ilustra o algoritmo abaixo:

$$pegBrancos := peg - pegPretos$$

Para formalizações subseqüentes na Seção 4.3, incluiremos em nosso superuniverso a função *avaliaPegs* com a seguinte assinatura:

$$avaliaPegs :: (Combinacao) \rightarrow (Combinacao) \rightarrow (Integer, Integer)$$

Como resultado, *avaliaPegs* retorna a dupla de inteiros referentes aos *Peg Pretos* e *Peg Brancos* da avaliação de um mapeamento *Combinacao* sobre outro mapeamento do mesmo tipo.

As regras de *avaliação de pegs* descritas no formalismo de ASM, para o exemplo apresentado, com código secreto $Codigo = [\ B,\ B,\ L,\ P]$ e tentativa $T = [B,\ L,\ B,\ B]$, deixam claro que as dúvidas referentes à "como o agente *tabuleiro* deve proceder ao receber uma tentativa" podem ser facilmente contornadas com clareza e precisão. Observe que o embasamento matemático a que as regras especificadas foram submetidas pode eliminar as dúvidas na interpretação dessas regras tornando, assim, o ambiente do jogo único. Essa observação pode ser estendida à todas as regras do jogo que forem especificadas corretamente em ASM.

4.3 Uso de IA na Solução do Agente Jogador

Finalizada a formalização das regras do MasterMind e do *tabuleiro*, trabalhar uma possível solução para o *jogador* é conveniente para experimentar não só o poder do formalismo de ASM, como o poder de implementação de alguma linguagem desse paradigma.

A utilização de técnicas de IA na formalização do *jogador* pode ser um bom caminho, tanto para possibilitar ao *jogador* a escolha de melhores tentativas, como para avaliar o suporte oferecido pelas ferramentas empregadas.

Várias técnicas que solucionam o *agente jogador* podem ser encontradas na literatura, sugerindo desde a utilização de linguagens lógicas, como implementações em Prolog [20], até o uso de técnicas mais avançadas como algoritmos genéticos e *simulated annealing* [7].

É importante ressaltar que a procura e análise de melhores técnicas para a implementação do agente *jogador* foge ao escopo do presente trabalho, que, ao contrário, visa avaliar a aplicação destas no formalismo de ASM. Assim, uma solução satisfatória é a utilização de algoritmos genéticos, tanto por solucionar o problema em um número reduzido de passos [7] como por ser uma técnica simples de implementar e bem consolidada dentre as técnicas oferecias pela IA.

4.3.1 Implementação do Agente Jogador usando AG

No período entre 1950 e 1965, a inteligência artificial foi marcada pelas idéias de redes neurais artificiais e a emergência das idéias sobre algoritmos evolucionários. Nesse período, também chamado de *subsimbólico*, a representação do conhecimento não era feita simbolicamente mas numericamente, provavelmente devido às limitações computacionais da época [4].

Os algoritmos evolucionários surgiram como métodos de pesquisa onde se preocupava em conseguir uma boa solução para problemas com vasto espaço de pesquisa. Assim, os algoritmos evolucionários formam uma classe de algoritmos de pesquisa probabilística e de otimização baseados no modelo de evolução orgânica, onde a natureza é a fonte de inspiração. Hoje, os principais representantes desse paradigma computacional, e que foram desenvolvidos independentemente, são conhecidos como: *estratégias evolutivas, programação evolucionária e algoritmos genéticos.* [4]

Para modelar a evolução orgânica, portanto, é necessária uma grande variedade de conceitos, tais como: genótipo, fenótipo, cromossomo, seleção natural, maximização, aprendizagem etc. pois existe uma inter-relação entre algoritmos genéticos e a biológica [4].

Para aplicar algoritmos genéticos como estratégia para solucionar o MasterMind, definir uma função algébrica que permita pontuar os indivíduos do conjunto de candidatos à solução do problema, aqui definido como universo *Populacao*, é fundamental para obter sucesso. Essa função pode ser chamada de *função genética.* Na verdade, os indivíduos são selecionados para a reprodução com base em suas pontuações definidas por essa função, além de determinar quais indivíduos são escolhidos para as comparações com o código real, sendo os de pontuação maior mais propensos a serem a solução do problema.

Outro desafio é determinar o tamanho do universo *Populacao* e quais técnicas serão usadas durante a reprodução dos indivíduos nele presentes. Isso garante a preservação de uma "variedade genética" de qualidade, enquanto os indivíduos da população convergem para a solução esperada.

Antes de partir para a formalização de uma possível solução para uma *função genética* em ASM, descreve-se de forma resumida os elementos envolvidos nessa solução.

Uma breve descrição do problema

Um indivíduo da população deve conter uma combinação de cores de mesmo tamanho que o código a ser desvendado. Assim, quando um elemento qualquer do universo *Populacao* possuir uma combinação de cores igual ao código, este terá a solução para o problema. Além disso, a cada *tentativa* que o jogador realizar em busca de descobrir o código secreto, duas tarefas serão realizadas, como listado abaixo:

1. Inicialmente, obter os *Peg Pretos* e *Peg Brancos* para o elemento de população que foi avaliado junto ao código real e armazená-los num conjunto especial, aqui chamado de universo *Historico*. Observe que os elementos do universo *Historico*, que armazena as *tentativas*, permitem avaliar o quanto os elementos da população estão *distantes* do código real, visto que eles possuem os *Pegs* de comparação com o código secreto.

2. Dado o universo *Historico*, gerar pontuações para todos os elementos de *Populacao*, aqui chamados de *individuos*, para sugerir os mais aptos à solução. Essa etapa contará com o auxílio de uma função chamada *parcelas* que brevemente será formalizada.

Os elementos do universo *Historico* são formados por dois campos, que são a combinação de cores, um mapeamento $Inteiros \rightarrow Cores$, já definido, e uma dupla de inteiros, (Integer,Integer), que significará os *Pegs*. Já os elementos de *Populacao*, os *individuos*, deverão ter dois campos também, sendo que o segundo apenas um *Inteiro*, que armazenará a pontuação destes.

Aqui, chamaremos de *falso código* os elementos do universo *Historico*. Isso porque o jogador pode avaliar todos os indivíduos de sua população contra os *falsos códigos* que têm uma relação direta com o código verdadeiro dado pelos seus Pegs.

Na Figura 4.5, não levando em consideração as posições, *individuo* difere do *falso código* em apenas uma cor, assim como o *falso código* difere do *código real*, já que a soma de seus Pegs é igual a 3. Além disso, *indivíduo* tem duas cores em posições diferentes do *falso código*, que são as cores de posições *2* e *4*. Veja a Figura:

Pela Figura 4.5 podemos perceber, também, que *individuo* é uma possível solução para o problema, devendo esse ser bem pontuado. Por isso o valor *0* na Figura 4.5, o que será explicado quando formalizada a *função genética*.

Uma especificação para a função genética em ASM

Vejamos, agora, uma especificação para *função genética* para o MasterMind. Sejam as definições:

Type Populacao is
(*Combina* : *Combinacao*;

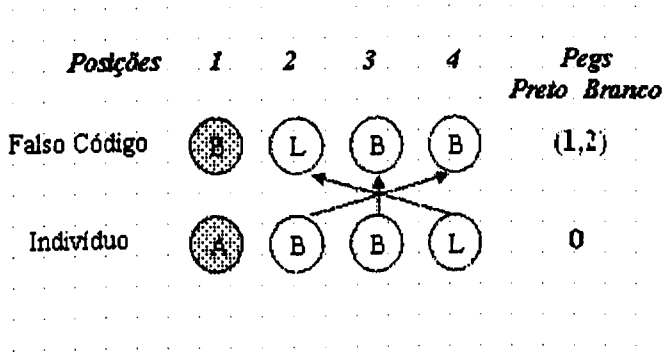


Figura 4.5: Pontuação de um indivíduo da população

Pontos : *Integer*;
)

Type Historico is
(*Combina* : *Combinacao*;
Pegs : (*Integer*, *Integer*);
)

Para auxiliar a formalização da *função genética*, define-se a função de assinatura *parcelas* como: *parcelas* : (*Combinacao*) → (*Historico* → *Integer*). Com ela, poderemos saber, para qualquer elemento de *Populacao*, qual sua distância em relação ao código real dada uma tentativa do *Historico*. Para obter, então, *pontos* desse elemento, basta fazer o somatório de todas as distâncias desse elemento ao código real.

Nosso superuniverso também possui a função *avaliaPegs*, definida na Seção 4.2. Para nossa especificação, *fcod* é uma variável que faz referência a um elemento do universo *Historico*

Veja o algoritmo:

let
$combH = fCod.Combina$
$(pretoH, brancoH) = fCod.Pegs$
$(pretoY, brancoY) = avaliaPegs(y, combH)$
in
$(parcelas(y))(fCod) := -(|pretoH - pretoY| + |brancoH - brancoY|)$

Veja que a execução dessa regra atribuirá um valor inteiro na função *parcela* para o mapeamento de domínio y. Esse valor inteiro pode ser interpretado como a "distância" que y está em relação ao código real em relação ao elemento *fCod*,

elemento esse que carrega consigo os *pegs* resultantes da avaliação com o código real, aqui representados pela dupla *fCod.Pegs*.

Essa regra de especificação pode ser abstraída pela macro *pontoParcial* parametrizada por *y* e *fCod*, ambos do tipo *Combinacao*. Em futuras formalizações, faremos referência à esta macro.

Como a macro *pontoParcial* :: (*Combinacao*, *Combinacao*) é capaz de pontuar uma *Combinacao* em relação a outra, para gerar todas as parcelas para uma Combinacao *ind* qualquer, dado o universo *Historico* completo, é necessário utilizar a seguinte especificação:

var *fCod* **ranges over** *Historico*
pontoParcial(*ind*, *fCod*)

A Figura 4.6 exibe valores da avaliação da função *Parcelas* aplicada ao elemento {B, L, B, B} do universo *Populacao* quando o universo *Historico* é constituído de quatro elementos, mostrados no subconjunto da mesma figura. Os pegs das tentativas do Histórico foram geradas quando avaliadas com o código real definido como *Codigo* = { *B, B, L, P* }, assim como nos exemplos anteriores. É interessante relembrar que cada *Combinacao* de *Populacao* será mapeada para um mapeamento definido como (*Historico* → *Integer*), como ocorre com o elemento {B, L, B, B}.

Fazer com que cada *parcela* seja não positiva é apenas uma opção de implementação. Assim, quanto mais próximo de zero a pontuação final de um elemento de *Populacao*, mais próximo este estará do código real e, conseqüentemente, um forte candidato a nova tentativa de acerto.

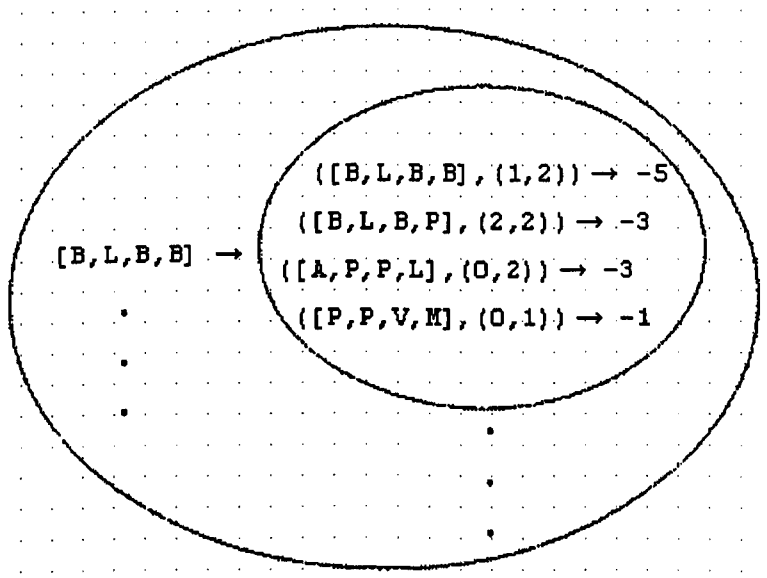


Figura 4.6: A função Parcelas para o código real {B,B,L,P}

Agora, para aplicar a todos elementos do universo *Populacao* basta estender a regra como:

var *ind* **ranges over** *Populacao*
  var *fCod* **ranges over** *Historico*
    pontoParcial(*ind.Combina*, *fCod*)

Finalmente, a especificação que pontua todos elementos de *Populacao*

var *y* **ranges over** *Populacao*
  y.Pontos := *somatorio*(*imagem*(*parcelas*(*y.Combina*)))

Diferentemente do que ocorre com a maioria das soluções em algoritmos genéticos, aqui os pontos dos elementos de *Populacao* devem ser recalculados toda vez que uma nova tentativa for incluída em *Historico*.

A pontuação de cada elemento da população depende diretamente do histórico que, quando alterado com a inclusão de uma nova tentativa, é intuitivo que a pontuação de cada elemento da população seja modificada. Assim, quanto maior o domínio de *Historico*, mais precisa será a pontuação dos candidatos à solução do problema.

Observe, também, que uma nova tentativa t enviada ao histórico é capaz de gerar uma nova linha na avaliação da função *parcelas* para cada elemento de seu domínio. Assim, sabendo qual pontuação a nova tentativa t produz a um elemento y da população, a pontuação de y recalculada será a soma do valor acumulado em *Pontos* de y com a nova pontuação gerada pela execução da macro *pontoParcial* para y e t. Veja a regra de especificação:

$$y.Pontos := y.Pontos + pontoParcial(y.Combina, t.Combina)$$

A Reprodução dos Indivíduos

Definida a *função genética*, outra função importante a ser definida refere-se a como *Populacao* irá evoluir, ou seja, quando "morre" ou como se"reproduz" um indivíduo. Como o *jogador* tem controle da qualidade genética de sua população, este pode excluir uma parcela de indivíduos que não oferecem relevância na busca da solução para o problema. Logicamente, essa exclusão exigirá a geração de novos indivíduos que podem ser obtidos "cruzando" os indivíduos que sobreviveram ao descarte dessa parcela, visto que na heurística de algoritmos genéticos, uma opção sugerida na literatura é manter o número de indivíduos constante como definido para a população inicial [40].

Para tornar o processo de reprodução mais claro, elimina-se uma porcentagem pré-definida pelo *jogador* de elementos de *Populacao*. Esses são selecionados pela

pontuação armazenada dada pela função genética. Também, os elementos que não forem excluídos serão transferidos para um universo especial chamado *Reprodutores*. Assim, nossa população ficará, temporariamente, vazia. Veja a Figura 4.7

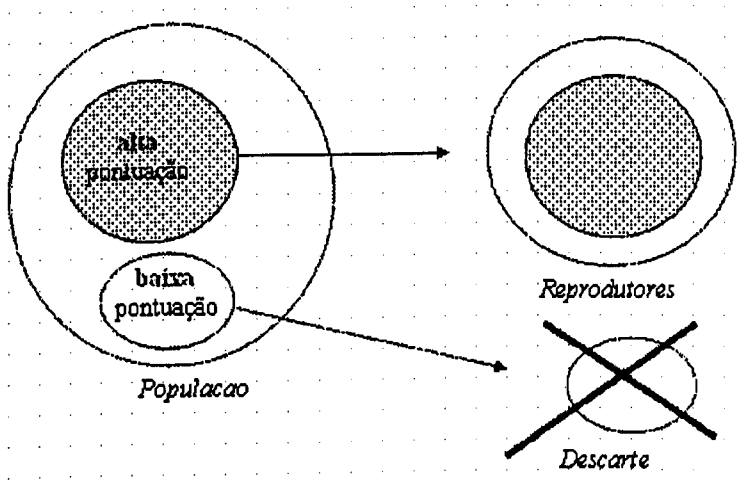


Figura 4.7: Reproduçao

A regra de especificação a seguir exibe um exemplo de como transferir elementos do universo *Populacao* para o universo *Reprodutores* ou, ao contrário, apenas descartá-los do sistema, dado um valor inteiro x limite para a pontuação mínima de aproveitamento desses elementos:

var y **ranges over** *Populacao*
  if$(y.Pontos > x)$**then**
    $Reprodutores(y) := true;$
  endif
  $Populacao(y) := false;$

Nosso próximo passo será gerar novos indivíduos, pontuá-los utilizando os elementos de *Historico* e armazená-los em *Populacao*.

A geração de novos indivíduos

Uma função para a geração de novos indivíduos deve ser cautelosa durante a reprodução. Um dos problemas relatados e contornados durante a formalização do MasterMind em AsmL foi a convergência dos indivíduos reproduzidos. Após várias *gerações*, observou-se que os indivíduos melhores geravam clones que diminuíam a variedade genética na população, impossibilitando, conseqüentemente, que a população gerasse a solução para o problema.

Uma alternativa para evitar essa convergência é não permitir a geração de um indivíduo já presente em *Populacao*. Além disso, optou-se por gerar os novos indivíduos considerando a carga genética de *NCombina* indivíduos da população, onde *NCombina* é o tamanho do código real, ou seja, um novo elemento tem *NCombina* genitores.

Assim como definido na Seção 4.2.1, consideramos o universo *Indices*, subconjunto dos números inteiros, definido como $Indices = \{x \in Inteiros : 1 \leq x \leq NCombina\}$, onde *NCombina* é o tamanho do código secreto e da tentativa, além, é claro, dos universos *Reprodutores* e *Populacao*.

A combinação de cores do novo indivíduo será gerada por elementos selecionados aleatoriamente no universo *Reprodutores* e mapeado pela função *novoIndividuo*, também presente em nosso superuniverso.

var indice **ranges over** Indices
  choose y **in** *Reprodutores*
    $novoIndividuo(indice) := y.Combina(indice)$;

Antes de tornar-se um elemento do universo *Populacao*, o novo indivíduo será pontuado utilizando as funções *pontoParcial* e, em seguida, o somatório de suas parcelas armazenadas no mapeamento *parcelas*

var $fCod$ **ranges over** *Historico*
  $pontoParcial(novoIndividuo, fCod)$;

Para incluir o novo elemento em *Populacao*, o cálculo de sua pontuação será realizado. Veja a regra de especificação:

let
  $pontosInd = somatorio(imagem(parcelas(novoIndividuo)))$;
in
  $Populacao((novoIndividuo, pontosInd)) := true$;

Terminada a reprodução dos novos indivíduos, a última preocupação antes do jogador selecionar uma nova tentativa para jogar contra o código real é trazer os elementos do universo *Reprodutores* para o universo *Populacao*, como exibido na regra de especificação abaixo:

var r **ranges over** *Reprodutores*
  $Reprodutores(r) := false$;
  $Populacao(r) := true$;

Uma especificação completa para o MasterMind foi desenvolvida na linguagem AsmL. A Seção 4.4 discute essa especificação e posiciona AsmL no contexto do trabalho realizado.

4.4 Especificação do MasterMind em AsmL

A linguagem AsmL, discutida no Capítulo 3, foi a ferramenta utilizada para gerar uma especificação executável do sistema MasterMind. Muitos recursos dessa linguagem foram importantes na codificação, gerando um produto final que computa os dados como esperado e fornece resultados, em média, em número reduzido de passos [7].

Uma observação importante a ser considerada é a liberdade que AsmL oferece ao programador durante a codificação dos sistemas. Observou-se que a implementação das especificações em ASM na linguagem AsmL é direta, oferecendo todo suporte para esse tipo de mapeamento. Porém, com características muito próximas de linguagens de programação convencionais, AsmL permite ao programador a construção de muitas especificações aparentemente distantes do paradigma de ASM.

Vários exemplos dessa abertura oferecida por AsmL ao programador podem ser percebidas prematuramente no manual da linguagem como, por exemplo:

1. **Step** - Com esse comando, o programador pode, sem dificuldade, programar "mascarando", por completo, as transições da máquina, assim como seus estados. Isso porque o comando *Step* força a máquina a mudar de estado sempre que avaliado, fazendo com que cada linha de código possa ser executada em seqüência quando intercalados pelo comando *Step*.

```
protected avaliaTentativa
var Result as DuplaInt_Pegs = DuplaInt_Pegs(0, 0)
step
  Result.PCertas := avaliaPosCertas(tenta, codigo)
step
  Result.PErradas := avaliaPosErradas(tenta, codigo)
step
  return Result
```

Como citado, cada comando dessa função será executado em um estado diferente, o que pode passar despercebido pelo programador.

2. O comando **while** - Com esse comando, o programador simula laços de repetição que aparentemente não estão vinculados à execução da máquina e sim ao escopo do comando. Veja o exemplo:

   private *avaliaPosErradas*

   var $i = 0$

   var $PErradas = 0$

   step while $i < NCombina$

   $PErradas := localizaCorPosErrada(i, tenta, codigo) + PErradas$

   $i := i + 1$

   step

   return PErradas

   A execução da função *avaliaPosErradas* exige, da máquina, no mínimo dois estados distintos separados pelo comando *Step*. Também, para um programador experiente no paradigma imperativo, o comando *while* geraria um laço de repetição executando dois comandos de atribuição. Veja que, em ASM puro, essas atribuições não seriam permitidas sem a mudança de estado, o que valida as atualizações. Aqui, a mudança de estado foi forçada pelo comando *Step*.

3. A ausência de **undef**. Considerar que todos elementos não mapeados em um universo são avaliados como *undef* é, antes de tudo, uma das premissas de ASM. Alguns transtornos foram evidenciados durante a implementação quando, ao testar se um elemento não pertencia a um determinado universo, *undef* não era avaliado. Veja o exemplo:

   if $(pontos\ in\ Indices(controle))$ **then**

   $controle(pontos) := controle(pontos) + 1$

   else

   $controle(pontos) := 1$

   O código acima mostra como avaliar se um mapeamento $controle :: Integer \rightarrow Integer$ mapeia um valor inteiro *pontos* como $\neg undef$

   A regra de especificação em ASM poderia ser escrita como:

   if $(controle(pontos) \neq undef)$ **then**

   $controle(pontos) := controle(pontos) + 1$

   elseif

   $controle(pontos) := 1$

Uma implementação em AsmL também permite que estruturas mais complexas como vetores e matrizes possam ser facilmente implementadas, como pode ser observado no Apêndice A. Isso implica que o mapeamento de um código escrito em, por exemplo, Java para AsmL pode não forçar o programador a seguir o paradigma de ASM. Isso demonstra uma grande desvantagem de AsmL para profissionais que queiram aprender ASM a partir de AsmL. Observe que aprender Programação Orientada a Objetos usando Java é natural, visto que Java força o programador a utilizar Programação Orientada a Objetos em suas estruturas internas, relação que não existe entre AsmL e o paradigma ASM.

Já em relação ao propósito do trabalho aqui realizado, AsmL também não contribui muito além de permitir especificações executáveis. Isso se dá porque as especificações AsmL não trazem clareza quanto às regras nelas implementadas. Como exemplo, podemos citar a definição de *classes* e principalmente *polimorfismo* em AsmL. No caso de *polimorfismo* mais especificamente, a decisão de qual função será executada é tomada em tempo de execução, o que pode comprometer a clareza na especificação.

Contudo, uma grande vantagem de AsmL é seu controle de visibilidade. Com esse recurso, foi possível implementar o MasterMind com a garantia de que, por exemplo, o código secreto seria invisível fora da classe em que foi definido, visto que este não foi definido como público.

4.5 Conclusões

Este capítulo discutiu uma especificação, em ASM, do jogo MasterMind. As regras do jogo escritas em ASM garantem aos competidores que queiram implementar o agente *jogador* clareza e elegância em suas descrições, tornando-as, assim, incontestáveis.

ASM também facilita, aos programadores e pessoas que tenham convivência com matemática, o entendimento das especificações sem elevado esforço. Essa característica é importante para viabilizar a utilização de ASM em especificações de jogos no ensino de inteligência artificial, já que os alunos envolvidos nesse ambiente têm a matemática como alicerce teórico.

Conclui-se, também, que AsmL pode não ser uma boa opção para aprender ASM. Isso pode ser justificado pela liberdade que AsmL dá ao programador ao decidir quanto às estruturas e forma de se implementar seus projetos, o que não força, necessariamente, a geração de especificações totalmente dentro do paradigma de ASM.

Por isso, as especificações em AsmL para o jogo MasterMind não foram priorizadas no trabalho, ficando essas como uma opção apenas para testar as formalizações realizadas em ASM, que podem ser diretamente mapeadas para AsmL e, em seguida, executadas.

O Capítulo 5 trata da aplicação de ASM em um exemplo de jogo multiagente, forçando os competidores a compartilharem as regras do jogo simultaneamente.

Capítulo 5

O Mundo Wumpus

5.1 Introdução

O *Mundo Wumpus* [46] é um jogo composto por uma caverna, dividida em várias *cavidades*, onde o monstro *Wumpus* reside. Em uma cavidade dessa caverna existe um tesouro de grande valor que é objeto de desejo de um agente *aventureiro* que queira desafiar a fúria do Wumpus. Além da cavidade onde está o Wumpus, outro perigo que a caverna oferece são buracos onde, caso o aventureiro caia em algum deles, a morte é inevitável.

Veja um exemplo na Figura 5.1, extraída de [46].

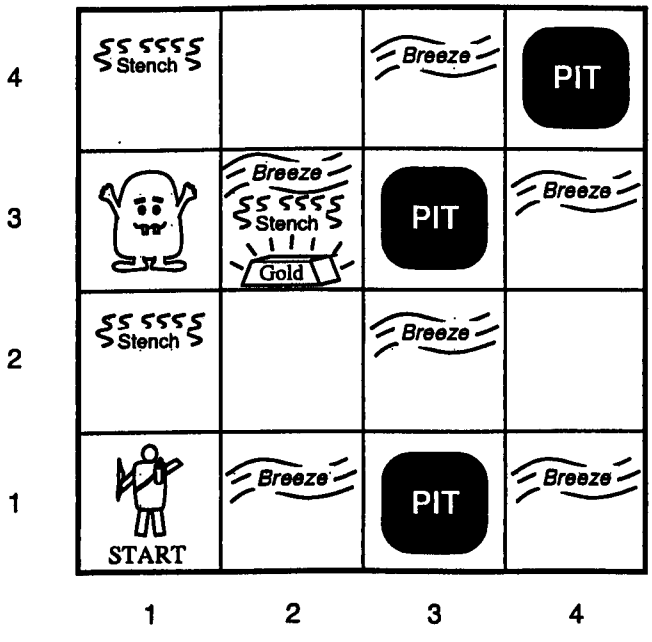


Figura 5.1: O Mundo Wumpus

Para escolher qual caminho seguir dentre as cavidades da caverna, o aventureiro conta com sensores que permitem que ele perceba quais são as condições das cavidades vizinhas à que ele se encontra.

Observe na Figura 5.1 que nas cavidades vizinhas ao *Wumpus* é possível perceber o mau cheiro do monstro, *stench*, enquanto que, nas cavidades vizinhas aos buracos, *pit* , é possível perceber uma brisa, *brezze*. Essas percepções são usadas pelo aventureiro para percorrer, com segurança, a caverna e encontrar o tesouro, que pode ser percebido pelo seu brilho.

Uma forma de referenciar as cavidades da Figura 5.1 é por meio de uma dupla de inteiros formada pelo valor da *linha* e da *coluna*. Assim, o Wumpus encontra-se na *cavidade(3,1)*, enquanto o aventureiro inicia sua busca ao tesouro a partir da *cavidade(1,1)*

O aventureiro também dispõe de uma flecha que ele poderá disparar em qualquer direção para tentar matar o monstro, o que eliminaria um perigo. Caso o monstro seja atingido, o aventureiro ouve seu gemido, o que significa sua morte.

Como objetivo principal, o jogo exige que o *aventureiro* encontre o tesouro e retorne ao ponto de partida. Para isso, o agente poderá contar com seu conhecimento acumulado durante o jogo para retornar sem que seja surpreendido no caminho por um buraco ou pelo monstro.

O Mundo Wumpus Multiagentes Para os propósitos deste trabalho, tornar o jogo *O Mundo Wumpus* um sistema multiagentes permite que competidores implementem seus agentes *aventureiros* e disputem, ao mesmo tempo, qual será o primeiro agente a encontrar o tesouro. Além disso, essa nova versão permite a avaliação do comportamento do formalismo de ASM, e alguma de suas linguagens, na aplicação em sistemas multiagentes, o que é definido em ASM e conhecido como ASM Distribuída [51].

Para simplificar as regras do jogo e atacar diretamente o alvo, optou-se por excluir do jogo a flecha que cada agente carrega. Isso evitaria um detalhamento mais aprofundado das regras do jogo, como por exemplo, quem deve ouvir o gemido do monstro quando atingido? Todos agentes envolvidos no jogo ou apenas o que o atacou?

Por ser um sistema discreto e permitir o emprego de técnicas de inteligência artificial na implementação dos *aventureiros*, o jogo *O Mundo Wumpus Multiagentes* é um bom exemplo a ser tratado neste trabalho. Um ponto interessante a ser discutido nas próximas seções refere-se à questão de justiça e visibilidade. Observe que todos aventureiros presentes na caverna devem ser tratados com igualdade, justiça, e não devem ter acesso à informações restritas à caverna ou a outros aventureiros, a visibilidade.

A partir de agora, as seções seguintes tratam apenas do *Mundo Wumpus Multiagentes.*

5.2 Regras do Mundo Wumpus Multiagentes

O ambiente do jogo *O Mundo Wumpus Multiagentes* é composto de uma *caverna* dividida em *cavidades* indexadas por duplas de inteiros, como no exemplo da Figura 5.1. As cavidades podem ser *vizinhas* ou *não vizinhas*. Duas cavidades são vizinhas quando suas referências na caverna são as duplas de inteiros (x, y) e $(x, y + 1)$ ou as duplas (x, y) e $(x + 1, y)$.

Nessa caverna também estão presentes o monstro *Wumpus*, que exala *mau cheiro* nas cavidades vizinhas à sua moradia, e alguns *buracos*, que geram uma *brisa* em suas cavidades vizinhas. Finalmente, em uma cavidade encontra-se o *tesouro*, que pode ser percebido pelo seu *brilho*.

O jogo começa quando os aventureiros são colocados na cavidade *entrada* da caverna e começam a busca pelo tesouro. Veja, abaixo, especificação informal das regras do Mundo Wumpus.

1. Todos aventureiros deverão iniciar o jogo na cavidade *entrada* (1, 1).
2. A cada passo do jogo, todos aventureiros poderão mudar de cavidade uma única vez.
3. Todo caminhamento dos aventureiros deve ser realizado entre cavidades vizinhas.
4. Dois ou mais aventureiros podem ocupar uma mesma cavidade (x, y) ao mesmo tempo.
5. Todo aventureiro que entrar na cavidade onde o Wumpus reside ou em outra que contenha um buraco estará automaticamente fora do jogo sendo considerado perdedor.
6. Caso dois ou mais aventureiros cheguem ao tesouro ao mesmo tempo, estes serão considerados vencedores em condições de empate. O fim do jogo é dado quando um ou mais agentes alcançam o tesouro, não sendo exigido que retornem à cavidade *entrada*.
7. O monstro Wumpus e os buracos nunca saem de suas cavidades definidas no início do jogo.
8. Uma cavidade não pode ser ocupada, ao mesmo tempo, por um buraco e pelo monstro. A cavidade do tesouro também deverá ser individual.
9. Iniciado o jogo, apenas os aventureiros poderão mudar de uma cavidade para outra em todo sistema.

Para simplificar o texto, dá-se o nome de *ocupantes* aos elementos que estão presentes na caverna, com exceção dos *aventureiros*, ou seja, ao monstro Wumpus e

ao seu mau cheiro, aos buracos e às brisas e, finalmente, ao tesouro e ao seu brilho. Dada a simplicidade das regras do jogo *O Mundo Wumpus*, a seção seguinte parte diretamente para sua formalização.

5.3 Utilização da Linguagem Machĩna

Como discutido na Seção 3.4 do Capítulo 3, a linguagem utilizada para a formalização das regras do jogo *O Mundo Wumpus* é a linguagem Machĩna [50].

Ao contrário da formalização do MasterMind, no Capítulo 4, onde a formalização das regras foi desenvolvida em ASM e, posteriormente, mapeada para a linguagem AsmL, neste capítulo todas as regras serão desenvolvidas diretamente em Machĩna. Essa decisão é motivada pela exigência que o jogo *O Mundo Wumpus* faz quanto à questão de visibilidade, não tratada por ASM. Observe que um agente *aventureiro* não pode ter acesso a algumas informações do ambiente como, por exemplo, se o Wumpus está em sua vizinhança.

Apesar de AsmL implementar um sistema eficiente de visibilidade, esta não atende, até o momento, ao quesito de sistemas multiagentes. Além disso, por ser mais próxima do formalismo de ASM, espera-se que a utilização de Machĩna permita, como resultado final, uma especificação de fácil entendimento ao leitor.

5.4 Formalização dos Módulos do Jogo

Como Machĩna implementa suas especificações divididas em módulos [50], faz-se, aqui, uma sugestão de quatro módulos para a implementação do jogo *O Mundo Wumpus*:

- Módulo **CavernaBase**, onde estruturas de conhecimento necessários ao ambiente e aos aventureiros são definidas;
- Módulo **MundoWumpus**, onde definem-se todas as regras do ambiente do jogo, como a geração da *caverna* e a distribuição de seus elementos: o monstro, os buracos e o tesouro;
- Módulo **AventureiroBasico**, que define um agente simples que procura o tesouro. Esse agente serve apenas para exemplificar o funcionamento do jogo;
- Módulo **Machina**, que é obrigatório e inicia a execução da máquina.

Como o módulo CavernaBase é simples e define estruturas necessárias para o módulo MundoWumpus e todos os aventureiros a serem implementados, sua especificação será descrita inicialmente, permitindo, conseqüentemente, o desenvolvimento dos demais.

5.4.1 Módulo CavernaBase

Uma informação que é de necessidade de todos os envolvidos no jogo é a forma como a caverna é estruturada. Com essa informação, não apenas o módulo *MundoWumpus* poderá estabelecer a posição dos ocupantes da caverna, como os aventureiros saberão como proceder para fazer o caminhamento na caverna em busca do tesouro.

Uma boa opção para a definição da caverna é dividi-la em partes, aqui chamadas de cavidades. Cada cavidade será referenciada por uma dupla de inteiros que fornecerão sua posição no plano. O primeiro inteiro da dupla será dado pela linha da cavidade na caverna, enquanto, o segundo, pela coluna.

A Figura 5.2 traz a especificação completa, em Machĩna, para o módulo *CavernaBase* e, na linha 4, define-se a estrutura definida para gerar as cavidades.

```
1  module  CavernaBase
2   initial  states:
3     type
4       public Cavidade is (x:  Int; y:  Int);
5       public Direcao is public enum { Norte, Leste, Sul, Oeste };
6     dynamic
7       public getLinha (c:  Cavidade):Int := c.x;
8       public getColuna(c:  Cavidade):Int := c.y;
9  end
```

Figura 5.2: Módulo CavernaBase em Machĩna.

Na linha 5 da Figura 5.2, a estrutura *Direcao* é definida. Com ela, um aventureiro poderá informar ao ambiente, descrito no módulo *MundoWumpus*, qual direção pretende seguir.

Já as linhas 7 e 8 descrevem duas funções que retornam, respectivamente, a linha e a coluna de uma determinada Cavidade.

Apesar de o módulo *CavernaBase* definir as estruturas da caverna, observe que este módulo não define as dimensões desta. Isso ficará a cargo do módulo *MundoWumpus*, que é responsável por criar o ambiente do jogo de forma invisível aos aventureiros.

5.4.2 Módulo MundoWumpus

Por gerir o ambiente do jogo, o módulo *MundoWumpus* é o mais importante do jogo. Um ponto crítico está relacionado à questão de visibilidade dos tipos, funções e ações do módulo. Como todos *aventureiros* deverão interagir com esse módulo a cada *ciclo* do jogo, caso os aventureiros tenham acesso a informações sigilosas,

estes poderão "roubar" durante o jogo, ou seja, poderão não seguir as regras como descritas na Seção 5.2.

Porém, antes de tratar dos problemas referentes à visibilidade neste módulo, é necessário definir alguns tipos que permitirão a geração da caverna.

Caverna Pode ser vista como um conjunto de cavidades cada qual com seus ocupantes. A definição dos possíveis ocupantes das cavidades pode ser feita utilizando uma enumeração, como abaixo:

Ocupante is enum
{
  Wumpus,
  MauCheiro,
  Buraco,
  Brisa,
  Tesouro,
  Brilho
};

Uma opção para se implementar a estrutura para a caverna é mapear, para cada cavidade, um conjunto com seus ocupantes, podendo esse conjunto de ocupantes ser vazio. Veja o código:

Caverna is Cavidade → *set of Ocupante*;

Na seção *Static* do módulo, definem-se as dimensões da caverna e o número de buracos que o ambiente oferecerá:

nBuracos : *Int* := 3;
LimiteLinha : *Int* := 10;
LimiteColuna : *Int* := 10;

Como as constantes são privadas ao módulo, nenhum aventureiro poderá obter as informações à respeito das dimensões da caverna, assim como o número de buracos que ela possui. Porém, como os aventureiros iniciarão a busca pelo tesouro a partir da cavidade *inicial*, referenciada pelas coordenadas (1,1), os limites inferiores da caverna são conhecidos por todos.

Com os tipos *Ocupante* e *Caverna* definidos, outra tarefa interessante é definir as regras que distribuirão os ocupantes nas cavidades da caverna para o início do jogo.

Geração da Caverna É uma etapa que ocorre somente no início do jogo, onde os ocupantes são colocados aleatoriamente em suas cavidades e, em seguida, o jogo se inicia.

Como um de seus recursos, Machĩna oferece uma seção no escopo de seus módulos definida pela palavra reservada *init*. Nessa seção, as funções ou ações referenciadas são executadas apenas uma vez ao iniciar a execução da máquina, o que também aumenta a clareza na especificação.

Portanto, com o uso da seção definida por *init*, as funções de especificação da geração da caverna podem ser executadas apenas uma vez sem a preocupação, por parte do programador, com variáveis de controle no código.

Fazendo uso de abstração, também oferecida por Machĩna, a chamada para as construções necessárias para iniciar o jogo pode ser feita com o seguinte comando:

```
//seção executada apenas ao iniciar a exceção da máquina
init
  constroiCaverna(Inicial);
```

onde *constroiCaverna*(*fase* : *SetupCaverna*) é uma ação parametrizada por uma enumeração que agrupa um conjunto de estados, definida como:

```
SetupCaverna is enum
{
   Inicial,
   InsereElementos,
   InicioJogo
}
```

Veja agora, de forma ainda abstrata, a especificação da ação *constroiCaverna*.

```
1  loop constroiCaverna(fase : SetupCaverna) :=
2    case fase of
3     Inicial ⇒
4       inicializaCaverna,
5       inicializaCavidades,
6       fase := insereElementos
7     ;
8     InsereElementos ⇒
9       posicionaElementos,
10      fase := inicioJogo
11    ;
12     InicioJogo ⇒
13      return
```

```
14    ;
15    end
16  end;
```

A seguir, descreve-se, com detalhes, as ações: *inicializaCaverna*, linha 4; *inicializaCavidades*, linha 5 e *posicionaElementos*, linha 9.

Ação inicializaCaverna Tem como função apenas mapear todas as cavidades da caverna para um conjunto de ocupantes vazio. Isso porque, em um segundo passo, os *ocupantes* serão sorteados e inseridos na caverna. Veja, abaixo, a regra para essa tarefa:

```
inicializaCaverna :=
  forall y in {1..LimiteLinha}
    forall x in {1..LimiteColuna}
      caverna((y,x)) := {};
end;
```

Onde *caverna* é uma função que pode ser definida como:

$$caverna\ is\ Cavidade \rightarrow set\ of\ Ocupante$$

Ação inicializaCavidades Vai gerar um conjunto de cavidades com todas as cavidades da caverna. Esse conjunto será utilizado de forma auxiliar na hora de posicionar os *ocupantes* na caverna. Para gerar esse conjunto, utiliza-se a regra definida como:

```
1  inicializaCavidades :=
2    forall y in {1..LiminteLinha}
3      forall x in {1..LimiteColuna}
4        if (x! = 1) and (y! = 1) then
5          todasCavidades ((y,x)) := true;
6  end;
```

Onde *todasCavidades* é uma função que pode ser definidada como:

$$todasCavidades\ is\ set\ of\ Cavidade$$

Visto que a cavidade (1,1) deve ser reservada como entrada para os agentes aventureiros, observe que esta não foi incluída no conjunto de cavidades que serão sorteadas para os *ocupantes* da caverna. A cavidade (1,1) é excluída do conjunto no teste condicional da linha 4 da especificação *inicializaCavidades*.

Com a caverna pronta para receber seus ocupantes, o próximo passo é, dada a lista de ocupantes, distribuí-los na caverna. Isso é feito com a execução da ação *posicionaElementos*.

Ação posicionaElementos Faz uso de uma lista de ocupantes, ainda não definida, que deve conter todos ocupantes da caverna. É a partir da lista que a caverna será povoada, ou seja, receberá seus ocupantes.

Dois problemas devem ser tratados ao gerar a lista de ocupantes, como listados:

1. Como o número de buracos, definidos por *nBuracos*, pode variar de um jogo para outro, a lista deverá ser montada dinamicamente na inicialização do jogo, recebendo, como ocupantes, o monstro, o tesouro e buracos.

2. Os ocupantes *Brilho*, *MauCheiro* e *Brisa* não participam da lista de ocupantes. Isso porque suas posições são definidas pelos ocupantes *Tesouro*, *Monstro-Wumpus* e *Buraco*, respectivamente.

Assim, uma possível especificação para a lista de ocupantes pode ser definida como:

$$listaOcupantes : list\ of\ Ocupante\ =\ [Wumpus,\ Tesouro] + listaBuracos(nBuracos);$$

onde $listaBuracos(n : Int)$ é uma função que fornecerá como retorno uma lista de buracos. Para nosso exemplo, a lista será $[Buraco,\ Buraco,\ Buraco]$, já que $nBuracos = 3$.

A estratégia para inserir os ocupantes na caverna, disparada pela ação pode ser bem simples. Observe que, sorteando uma cavidade qualquer no conjunto *todasCavidades*, pode-se pegar um ocupante da lista *listaOcupantes* e inseri-lo na caverna. Obviamente que a cavidade sorteada deve ser retirada do conjunto *todasCavidades* para que não seja sorteada para outro ocupante, que deve ter sua cavidade particular.

A ação *posicionaElementos* se encarrega apenas de retirar um elemento da lista e solicitar que a ação *insereOcupante* o insira na caverna. Assim, quando a lista estiver vazia, nenhum ocupante deverá mais ser inserido na caverna.

Veja como a ação *posicionaElementos* pode ser especificada:

```
1 loop posicionaElementos :=
2   if listaOcupantes = [ ] then return
3   else
4     let (a : x) = listaOcupantes
5     in
6        insereOcupante(a);
```

```
7          listaOcupantes := x;
8        end
9      end
10   end;
```

Veja, agora, a especificação para a ação insereOcupante:

```
1   insereOcupante(elemento : Ocupante) :=
2     let cavidade := sorteiaCavidade(todasCavidades);
3     in
4       (caverna(cavidade))(elemento) := true;
5       todasCavidades(cavidade) := false;
6       if (elemento != Tesouro) then
7           montaVizinhos(getCaracteristica(elemento), cavidade);
8       else
9           (caverna(cavidade))(Brilho);
10    end
11  end;
```

Os itens abaixo descrevem o funcionamento da ação descrita:

- Na linha 2, *cavidade* é obtida pelo resultado da função *sorteiaCavidade* que, de forma não determinística, sorteia uma cavidade no conjunto *todasCavidades*.
- A linha 4 se encarrega de incluir o ocupante, dado pelo parâmetro *elemento*, na caverna.
- A linha 5 garante que a cavidade sorteada não mais participará do sorteio, excluindo-a do conjunto *todasCavidades*.
- No fim da regra, caso o ocupante não seja o *Tesouro*, a ação montaVizinhos, linha 7, coloca nas cavidades vizinhas à do novo ocupante sua característica. Essa característica também será um ocupante, podendo ser *MauCheiro* ou *Brisa*.
- No caso do ocupante *Tesouro*, sua característica, *Brilho*, não é colocada em sua vizinhança e sim em sua cavidade. Isso é feito na linha 9 da especificação.

Terminada a construção da caverna, o jogo pode ser iniciado.

A seguir, descrevem-se as regras de especificação para permitir que o ambiente defina as características básicas que cada aventureiro deverá ter. Essas regras são implementadas no módulo *MundoWumpus*.

Implementação das Estruturas para os Aventureiros Outra observação importante refere-se à estrutura básica dos agentes *aventureiros*. Por ser o ambiente quem controla suas ações durante o jogo, é necessário que estas sejam de conhecimento e controle do ambiente.

Uma solução interessante é o ambiente *MundoWumpus* descrever uma estrutura para que todos aventureiros possam ser definidos a partir desta. Para o jogo *O Mundo Wumpus*, ao ambiente basta saber *onde* o aventureiro se encontra em um dado momento e qual *direção* este deseja seguir. Veja a especificação abaixo:

```
AvDescriptor is (
  nome : String;
  direcao : Direcao;
  posicao : Cavidade;
);
```

O *nome* do aventureiro foi incluído como uma opção para a distinção entre os agentes. Observe que os tipos *Direcao* e *Cavidade* foram definidos no módulo *CavidadeBase*, que deve ser importado pelo módulo *MundoWumpus* para que sejam reconhecidos.

Como o ambiente gerencia muitos agentes, uma opção para associar as informações definidas pela estrutura *AvDescriptor* com seus respectivos aventureiros, é utilizar o mapeamento *avInfor*, descrito como:

$$avInfor : Agent \rightarrow AvDescriptor;$$

Para garantir que os agentes aventureiros não consigam alterar suas informações durante o jogo, o mapeamento *avInfor* não pode ser público.

Um acesso que os aventureiros devem ter é a visualização de suas informações pessoais controladas pelo ambiente sem alterá-las. Para isso, o ambiente disponibiliza funções de visibilidade pública. Outra atenção a ser tomada é garantir que as funções implementadas não permitirão que um aventureiro solicite as informações de outros aventureiros. Veja as especificações em Machĭna:

```
public minhaPosicao : Cavidade := avInfor(self).posicao;
public minhaDirecao : Direcao  := avInfor(self).direcao;
public meuNome      : String   := avInfor(self).nome;
```

Observe que, quando o ambiente vai fornecer uma informação, como por exemplo a direção do aventureiro, esse colhe as informações através do mapeamento *avInfor* para o próprio agente que solicitou a informação. Isso é possível com a utilização da palavra reservada *self* que referencia o agente que disparou a execução da função *minhaDirecao*, em nosso exemplo.

Para uso específico do ambiente, sempre que este precisar saber a posição ou a direção de algum agente aventureiro, não será possível fazê-lo disparando as funções *minhaPosicao*, *minhaDirecao* e *meuNome* definidas anteriormente. Isso porque o parâmetro *self* faz referência, por definição, ao agente que disparou as funções, o que não faz o menor sentido para o ambiente.

Assim, definem-se funções privadas para uso específico do ambiente. Veja, abaixo, as especifições para esas funções:

$$
\begin{aligned}
&posicaoAgente\ (ag:\ Agent)\ : Cavidade := avInfor(ag).posicao;\\
&direcaoAgente\ (ag:\ Agent)\ : Direcao\ := avInfor(ag).direcao;\\
&nomeAgente\ \ (ag:\ Agent)\ : String\ := avInfor(ag).nome;
\end{aligned}
$$

Agora, para que os agentes aventureiros sejam capazes de decidir qual caminho seguir, basta permitir que eles saibam se estão em uma cavidade sem nenhum dos possíveis *ocupantes* ou, caso tenha ocupantes, quais são. É bom lembrar que, caso o aventureiro esteja em alguma cavidade onde mora o Wumpus ou tenha um buraco, esse não precisa de técnicas para obter essa informação, visto que ele será automaticamente eliminado do jogo.

Para prover as informações sobre os ocupantes das cavidades, optou-se por funções lógicas, como descritas abaixo:

$$
\begin{aligned}
&public\ temBrisaAqui\ \ := Brisa\ \ in\ caverna(minhaPosicao);\\
&public\ temBrilhoAqui\ \ := Brilho\ \ in\ caverna(minhaPosicao);\\
&public\ temMauCheiroAqui\ := MauCheiro\ in\ caverna(minhaPosicao);
\end{aligned}
$$

Finalmente, para efetivar a mudança de direção dentro da caverna, o ambiente pode oferecer a função pública:

$$
\begin{aligned}
&public\ novaDirecao(d:\ Direcao)\ :=\\
&\quad avInfor(self).direcao := d;\\
&end;
\end{aligned}
$$

Mais uma vez, com o uso da palavra reservada *self*, Machĩna garante que o aventureiro poderá alterar apenas sua direção, não permitindo que ele interfira nos demais aventureiros do ambiente.

Com a implementação das estruturas básicas oferecidas pelo ambiente para interagir com os aventureiros, os competidores interessados em desenvolver especificações para seus agentes estarão livres para iniciar o trabalho. Agora, cada um poderá fazer uso de técnicas de inteligência artificial que mais lhe for conveniente.

Como exemplo, o trabalho sugere um agente aventureiro básico totalmente desprovido de inteligência quando procura pelo tesouro. Para escolher o caminho, ao

contrário de usar regras de inferência para percorrer a caverna, nosso aventureiro escolhe, aleatoriamente, uma posição e solicita ao ambiente que o mova para ela.

Apesar de burro, nosso agente permite compreender como a linguagem Machĭna implementa um sistema multiagente ao criar e incluir nosso agente no ambiente. A Seção 5.4.3 discute, com mais detalhes, nosso agente aventureiro básico.

Porém, ainda é necessário conhecer as regras que permitem ao módulo *Machina* iniciar a execução da máquina incluindo os agentes no ambiente e também as regras que permitem ao ambiente o gerenciamento dos passos dos aventureiros em busca do tesouro. Essas regras serão especificadas a seguir.

Regras do Ambiente Para que o ambiente do jogo receba os agentes *aventureiros*, definiu-se uma ação chamada *login*. Para agrupar os agentes, o ambiente dispõe do conjunto *aventureiros*, inicialmente vazio, onde todos agentes aventureiros serão incluídos.

É bom lembrar que a ação login será utilizada pelo módulo *Machina* que, ao iniciar a execução da máquina, criará os agentes aventureiros e os incluirá no também recém gerado ambiente, que é um agente do módulo *MundoWumpus*.

A regra abaixo descreve a ação *login*:

```
1  login (a : Agente, nome : String) :=
2     aventureiros(a) := true;
3     avInfor(a) := AvDescriptor(nome, Norte, Cavidade(1, 1));
4  end;
```

onde *aventureiros* é a função definida como: $aventureiros : set\ of\ Agent = \{\}$

Veja que, além de incluir o agente no conjunto *aventureiros*, linha 2, a ação *login* também deve mapear as características do aventureiro no mapeamento *avInfor*, linha 3, definido nesta seção. Todos os agentes entrarão na caverna pela cavidade *inicial*, também descrito na linha 3 da ação *login*.

O leitor pode achar estranho a ação *login*, que será usada pelo módulo *Machina* não ser pública. Como sabemos, para que um módulo da linguagem Machĭna utilize estruturas definidas em outros módulos, estas estruturas devem ser públicas. Porém, no caso da ação login, defini-la como pública pode ser perigoso, pois um agente poderia ser introduzido no jogo a qualquer momento por qualquer agente de outro módulo que importe esta ação.

Assim, para solucionar esse problema, uma sugestão à linguagem Machĭna é que o módulo *Machina* tenha acesso à todas estruturas de um sistema, já que este módulo é principal, obrigatório e controlado pelos implementadores do sistema. Para finalizar a sugestão, cada sistema deve possuir apenas um módulo *Machina*.

Falta-nos definir, agora, a regra de transição do módulo que aqui se descreve. Como tarefa básica, esse módulo deve avaliar, a cada passo, se algum agente alcançou

o tesouro. Caso isso tenha acontecido, o jogo chega ao fim e os agentes que estiverem posicionados na cavidade onde está o tesouro são considerados campeões em condição de empate, caso tenha mais de um.

Por outro lado, caso nenhum aventureiro tenha localizado o tesouro, o jogo deverá prosseguir. Para isso, os aventureiros que nesse momento estiverem em uma cavidade que tenha um buraco ou o monstro *Wumpus* devem ser eliminados do jogo enquanto os demais poderão, mais uma vez, caminhar na caverna.

Abaixo, a regra de transição do módulo *MundoWumpus*:

```
1  forall y in aventureiros
2    let cavidade_y = posicaoAgente(y)
3    in
2      if temTesouro (cavidade_y) then
3          vencedores(y) := true;
4          stop;
5       elseif temWumpus (cavidade_y) or temBuraco (cavidade_y) then
6            aventureiros(y) := false;
7          else
8            caminha(y);
9      end
10   end;
11 end;
```

onde *vencedores* é a função definida como: $vencedores : set\ of\ Agent = \{\}$

Quando encontrados agentes na cavidade onde está o tesouro, esses agentes são incluídos no conjunto *vencedores*, linha 3 e um comando *stop* é executado, linha 4. Em Machĩna, a execução do comando *stop* irá finalizar as transições de todos agentes, finalizando, assim, o jogo.

Observe na linha 6 como os agentes que se encontram nas cavidades que possuem buracos ou o monstro são eliminados do jogo, bastando apenas que esses deixem de ser elementos do conjunto *aventureiros*.

Já na linha 8, o disparo da regra para o caminhamento dos aventureiros fará com que o ambiente mude os agentes para uma cavidade de sua vizinhança baseando-se na direção que cada um deles definiu.

Descreve-se, agora, a ação que faz o agente caminhar:

```
caminha (av : Agent) :=
  let
      linha_av = getLinha(posicaoAgente(av));
      coluna_av = getColuna(posicaoAgente(av));
  in
```

```
    case direcaoAgente(av) of
      Norte ⇒
        if coluna_av < LimiteColuna then
          avInfor(av).posicao := Cavidade(linha_av, coluna_av + 1);
      Sul ⇒
        if coluna_av > 1 then
          avInfor(av).posicao := Cavidade(linha_av, coluna_av − 1);
      Leste ⇒
        if linha_av < LimiteLinha then
          avInfor(av).posicao := Cavidade(linha_av + 1, coluna_av);
      Oeste ⇒
        if linha_av > 1 then
          avInfor(av).posicao := Cavidade(linha_av − 1, coluna_av);
    end
  end;
```

5.4.3 Um Aventureiro Sem Inteligência

Esta seção descreve um aventureiro que procura pelo tesouro sem uso de inteligência ou conhecimento. Apesar de parecer inútil, o módulo *AventureiroSimples* pode ajudar no entendimento de como a execução da máquina, quando iniciada no módulo Machĭna, permitirá a entrada dos aventureiros no ambiente.

```
1  module  AventureiroSimples
2    import:
3       MundoWumpus(novadirecao);
4       CavernaBase(Direcao);
5     action
6        decideDirecaoAleatoria :=
7            choose y in Direcao
8               novaDirecao(y)
9        end;
10   transition
11       decideDirecaoAleatoria
12 end
```

Figura 5.3: Módulo AventureiroSimples em Machĭna.

A cada transição, nosso aventureiro simples irá decidir a direção que ele deseja seguir de forma não determinística, usando, para isso, recursos da linguagem.

Veja, na Figura 5.3 , a especificação completa do módulo *AventureiroSimples*:

A seção seguinte implementa o último módulo, que é obrigatório na linguagem Machĩna, e também chamado de *Machina*.

5.4.4 Módulo Machina

Para a especificação do jogo *O Mundo Wumpus*, o módulo *Machina* tem como objetivo criar um agente responsável pelo ambiente, e irá inserir os aventureiros neste ambiente.

Como somente um módulo para o aventureiro foi implementado, simularemos a entrada de dois aventureiros, com o mesmo código, no ambiente.

A Figura 5.4 traz a especificação completa do módulo *Machina*.

```
 1  module  Machina
 2    import:
 3       MundoWumpus(login);
 4    init
 5       create MundoWumpus,
 6       create x : agent of AventureiroSimples do
 7          login (x, "Pirata"),
 8       create y : agent of AventureiroSimples do
 9          login (y, "Cavaleira")
10    end
11  end
```

Figura 5.4: Módulo Machina.

Por estar na seção *init*, as regras do módulo *Machina* serão executadas apenas uma vez quando a máquina iniciar a execução. Na linha 5 é disparada a execução do ambiente e, nas linhas 6, 7, 8 e 9, são criados dois aventureiros e inseridos no ambiente, com o uso da ação *login*

5.5 Controle dos Agentes Aventureiros

Um ponto que não foi discutido durante a especificação do jogo *O Mundo Wumpus* é quanto ao gerenciamento, por parte do módulo *MundoWumpus*, do momento correto em que os agentes aventureiros devem caminhar na caverna. De acordo com a especificação anterior, não existe uma forma de se estabelecer quando um agente deve escolher sua nova posição, já que não existe sincronia entre a execução das regras de transição dos aventureiros com a execução das regras de transição do *MundoWumpus*, responsáveis por mover os aventureiros na caverna.

Isso significa que nada impede que um aventureiro esteja decidindo uma nova direção a seguir antes mesmo de ele ter caminhado para a cavidade referente a direção escolhida por ele no passo anterior.

Outra situação inconveniente pode ocorrer caso a regra de transição que faz o aventureiro caminhar seja executada mais de uma vez consecutiva. Caso isso ocorra, o ambiente faria o aventureiro caminhar mais de uma vez utilizando a direção indicada por ele para apenas um passo. Isso poderia, acidentalmente, fazer com que o aventureiro caia em um buraco ou entre na morada do monstro Wumpus sem que este tenha decidido por esse caminho.

Várias soluções podem ser avaliadas e especificadas para cercar o problema aqui discutido. Uma delas pode ser definida fazendo com que a regra de transição que move os aventureiros na caverna seja executada somente após a confirmação de que os aventureiros decidiram a nova direção que desejam seguir.

Solução Para especificar esta solução, ao contrário de manter os aventureiros no conjunto *aventureiros*, como definido na Seção 5.4.2, definem-se dois conjuntos para agrupar os aventureiros, como:

```
1 aventureirosJogam : set of Agent = {};
2 aventureirosMovem : set of Agent = {};
```

Inicialmente, como todos aventureiros começam o jogo na cavidade *inicial*, eles devem disparar a regra para escolha da direção que desejam seguir. Por isso, o conjunto *aventureirosJogam* será preenchido primeiro, o que não permitirá que a nova regra de transição do ambiente, a ser definida, os mude de cavidade antes que as direções sejam definidas.

Para isso, é necessário redefinir a ação *login* definida na Seção 5.4.2, passando a ser especificada como:

```
1 login (a : Agente, nome : String) :=
2   aventureirosJogam(a) := true;
3   avInfor(a) := AvDescriptor(nome, Norte, Cavidade(1, 1));
4 end;
```

Ao iniciar o jogo, à medida que os aventureiros escolherem a direção que desejam seguir, eles serão transferidos para o conjunto *aventureirosMovem*. A execução da regra para movê-los será efetivada quando todos aventureiros forem elementos do conjunto *aventureirosMovem*. A constatação de que todos os aventureiros já escolheram sua direção será confirmada quando o conjunto *aventureirosJogam* estiver vazio.

Veja, abaixo, como o módulo *MundoWumpus* deve proceder ao receber a direção escolhida por cada aventureiro apenas redefinindo a ação *novaDirecao*, também

definida na Seção 5.4.2:

```
1   public novaDirecao(d : Direcao) :=
2     if aventureirosJogam(self) then
3        avInfor(self).direcao := d;
4        aventureirosJogam(self) := false;
5        aventureirosMovem(self) := true
6   end;
```

Finalmente, veja, abaixo, a redefinição da regra de transição do módulo *Mundo-Wumpus* para garantir que somente moverá os aventureiros quando todos já tiverem executado a ação *novaDirecao*:

```
 1 if aventureirosJogam = { } then
 2   forall y in aventureirosMovem
 3     let cavidade_y = posicaoAgente(y)
 4     in
 5       aventureirosMovem(y) := false;
 6       if temTesouro(cavidade_y) then
 7           vencedores(y) := true;
 8           stop;
 9         elseif not(temWumpus(cavidade_y)) and
                   not(temBuraco(cavidade_y)) then
10           aventureirosJogam(y) := true;
11           caminha(y);
12       end
13     end
14 end
```

Observe que, na linha 1, todos os aventureiros deixarão o conjunto *aventureiros-Movem*. Caso nenhum agente tenha encontrado o tesouro, o que significaria o fim do jogo, os aventureiros que não caíram em algum buraco, assim como os que não entraram na morada do monstro, serão, novamente, enviados para o conjunto *aventureirosJogam*, linhas 7 e 8.

As redefinições especificadas nesta seção garantem um sincronismo entre as execuções das regras de transição dos aventureiros com a regra de transição do ambiente. Assim, pode-se garantir que os aventureiros caminharão somente após terem definido a direção que desejam seguir.

5.6 Demonstração de Propriedades

Nesta seção, mostramos como especificações escritas em ASM podem facilitar a demonstração de propriedades.

Aproveitamos as demonstrações para explicar com detalhes a forma como a linguagem Machĩna trata a execução de especificações ASM distribuídas, dividindo-as em módulos e usando agentes.

Para as provas desta seção, vamos considerar os códigos atualizados pelas alterações apresentadas na Seção 5.5.

5.6.1 Propriedade 1

As regras apresentadas nunca irão produzir atualizações inconsistentes

Como visto no Capítulo 2, um *endereço* é identificado por um nome de função e uma tupla de valores com comprimento igual à aridade desta função.

A inexistência de atualização inconsistente significa que, em um passo de uma execução, nunca um mesmo *endereço* irá ser atualizado com dois ou mais valores distintos. Embora os agentes executem de maneira concorrente e distribuída, o estado é global. Assim, a inexistência de atualização inconsistente deve ser garantida inclusive para atualizações que ocorram em agentes diferentes.

Demonstração Começamos pelo módulo *Machina*, que é sempre o primeiro a ser executado. A seção *init* cria um agente do módulo *MundoWumpus* e agentes *aventureiros*, como na Figura 5.4.

A ordem de execução determinada por Machina é:

1. Inicialização de funções com valores especificados dentro do próprio código, como em: $nBuracos : Int = 3$.

2. Em um passo seguinte, execução da seção *init* de cada módulo envolvido no sistema. É importante ressaltar que essa inicialização não depende da criação de agentes para o modulo em questão. A seção *init* é executada uma única vez para cada módulo, antes da criação de qualquer agente.

3. Execução da seção *init* do módulo *Machina*. Nessa seção, agentes de outros módulos podem ser criados.

4. Execução das regras de transição dos módulos para os quais um agente esteja ativo.

Antes de prosseguir, devemos então verificar as inicializações presentes nas declarações das funções em cada módulo. Por inspeção, é facil ver que nenhuma atualização inconsistente é produzida.

Vamos verificar então a seção *init* do módulo *MundoWumpus*.

Observando a especificação apresentada na Seção 5.4.2, pode ser observado que a ação *constroiCaverna* deve ser analisada. Essa ação executa outras ações, em dois passos consecutivos, que iremos analisar em seguida.

No primeiro passo da execução da ação *constroiCaverna*, as ações *inicializaCaverna* e *inicializaCavidades* são executadas em paralelo. Na ação *inicializaCaverna*, a função caverna é atualizada em *endereços* diferentes. Na ação *inicializaCavidades*, a função *todasCavidades* é atualizada em *endereços* diferentes. Assim, não pode haver atualizações inconsistentes.

No segundo passo da execução da ação *constroiCaverna*, a ação *posicionaElementos* é executada, disparando, por conseqüência, execução da ação *insereOcupante*, tendo como argumento um elemento da lista *listaOcupantes*.

Por inspeção, podemos ver que a ação *insereOcupante* não produz atualização inconsistente. Veja, na especificação a seguir, como o código da ação *montaVizinhos* acrescenta elementos em uma cavidade usando o mesmo artifício que a ação *insereOcupante*.

```
montaVizinhos(vizinho : Ocupante, cavi : Cavidade) :=
  let x := getLinha(cavi);
      y := getColuna(cavi);
  in
    if y < limiteColuna then
      caverna((x, y + 1)) := caverna((x, y + 1)) + vizinho
    if x < limiteLinha then
      caverna((x + 1, y)) := caverna((x + 1, y)) + vizinho
    if y > 1 then
      caverna((x, y − 1)) := caverna((x, y − 1)) + vizinho
    if x > 1 then
      caverna((x − 1, y)) := caverna((x − 1, y)) + vizinho
    end
end;
```

Vamos analisar agora a seção *init* do módulo *Machina*, exibida na Figura 5.4.

É preciso analisar os disparos de execuções paralelas de ações *login*. O comando *create* de ASM garante que os elementos criados em um domínio sejam diferentes de quaisquer outros já existentes. Assim, a execução de ações *login* em paralelo irá atualizar as funções *aventureirosJogam* e *avInfor* em *endereços* diferentes, como especificado na nova ação *login* redefinida na Seção 5.5.

Falta verificar, ainda, se atualizações inconsistentes ocorrem na execução das regras dos agentes criados. Os agentes criados pelo módulo *Machina* são:

- um agente do módulo *MundoWumpus*;

- agentes de módulos que controlam os agentes aventureiros.

A única ação pública que pode ser executada por agentes que controlam os aventureiros, e que pode assim afetar o estado global, é a ação *novaDirecao*.

Uma execução em paralelo da ação *novaDirecao* com a regra de transição do módulo *MundoWumpus* pode ocorrer, uma vez que são executadas por agentes diferentes.

Inspecionando o código, pode-se ver que atualizações inconsistentes poderiam ser produzidas nas funções *aventureirosJogam* e *aventureirosMovem*.

Entretanto, as condições que precedem o código da regra de transição do módulo *MundoWumpus* e da ação *novaDirecao* são excludentes, assim apenas o código de uma delas poderia ser completamente executado em um mesmo passo.

Além disso, execuções em paralelo da ação *novaDirecao*, em agentes diferentes, não podem produzir atualizações inconsistentes, uma vez que a função *self* tem uma interpretação diferente em cada agente que executa uma regra de transição.

Com os argumentos apresentados acima, podemos afirmar que nenhuma atualização insconsistente pode ser produzida pela especificação construída para o Mundo Wumpus.

5.6.2 Propriedade 2

Para a execução dos agentes que controlam os aventureiros, a especificação do Mundo Wumpus implementa um esquema que satisfaz critérios de justica.

Essa propriedade significa que os agentes que controlam os aventureiros têm chances iguais para determinar sua direção de movimento. O deslocamento de todos os aventureiros é produzido, em paralelo, na regra de transição do módulo *MundoWumpus*.

Esse deslocamento só é efetivado se a condição que precede as regras for satisfeita, isto é, se o conjunto *aventureirosJogam* for vazio.

Podemos apresentar uma definição um pouco mais formal dessa propriedade como a seguir:

Uma execução do sistema especificado é uma seqüência de estados $s0, s1, s2, ...$, como em qualquer execução de ASM.

O estado global inicial é definido por $s0$.

Seja $nExec(a)$ o número de vezes que a ação *novaDirecao* foi executada com sucesso por um agente a qualquer. Vamos considerar uma execução com sucesso dessa ação quando a condição que a precede for satisfeita, isto é, $aventureirosJogam(self)$ é uma condição verdadeira, o que implica que o agente a é um elemento do conjunto *aventureirosJogam*.

Um esquema para execução que satisfaz critérios de justica deveria implementar o seguinte:

[P1] Quando a regra de transição do módulo *MundoWumpus* for executada com a condição que precede a condição verdadeira, isto é, o conjunto *aventureirosJogam* for vazio, $nExec(a)$ terá o mesmo valor para todos os agentes a que controlam os aventureiros, com exceção daqueles que ja tiverem sido eliminados do jogo. A eliminação de um aventureiro ocorre, como se sabe, quando ele entra em uma célula que contém o monstro *Wumpus* ou um *buraco*.

A propriedade [P1] garante que os aventureiros serão movidos em paralelo após cada um deles ter escolhido a sua direção de deslocamento.

Para mostrar que [P1] é sempre verdadeira em uma execução, vamos apresentar uma demonstração usando indução finita. Para facilitar a demonstração, vamos usar outras duas propriedades, no lugar de [P1]:

[P2] Sejam a e b dois agentes que controlam aventureiros. Em um dado estado, se a pertencer ao conjunto *aventureirosJogam* e b pertencer ao conjunto *aventureirosMovem*, então $nExec(b) = nExec(a) + 1$.

Se a e b pertencerem ao mesmo conjunto, $nExec(a) = nExec(b)$.

[P3] O conjunto dos agentes que controlam os aventureiros, exceto aqueles que foram eliminados do jogo, será igual à união dos conjuntos *aventureirosJogam* e *aventureirosMovem*.

Sempre que as propriedades [P2] e [P3] forem verdadeiras, [P1] também será. Isso acontece porque, quando o conjunto *aventureirosJogam* é vazio e [P3] é verdadeira, então o conjunto *aventureirosMovem* contém todos os agentes que controlam os aventureiros, exceto aqueles que foram eliminados do jogo. Se [P2] também é verdadeira, todos esses agentes executaram com sucesso a regra da ação *novaDirecao* o mesmo número de vezes, pois todos estão dentro do mesmo conjunto *aventureirosMovem*.

A demonstração a seguir, portanto, apresenta uma prova para as propriedades [P2] e [P3].

Demonstração Vamos supor que pelo menos um agente aventureiro foi criado pelo modulo Machina. Caso contrário, a demonstração seria óbvia. Assim, no estado inicial $s0$, após execuções da ação *login* pelo módulo *Machina*, teríamos:

- O conjunto *aventureirosJogam* diferente de vazio, contendo todos os agentes que controlam os aventureiros.
- O conjunto *aventureirosMovem* vazio.
- $nExec(a) = 0$, para todo agente a que controla um aventureiro.

Assim, as propriedades [P2] e [P3] são obviamente válidas para o estado inicial $s0$.

No passo de indução, vamos supor que as propriedades são válidas em um estado sK qualquer da execução, e mostrar que serão válidas em um estado seguinte $sK+1$.

Seja sK um estado em que as propriedades [P2] e [P3] são válidas. Vamos dividir a analise em dois casos:

1. O conjunto *aventureirosJogam* é vazio.

   Neste caso, uma vez que o estado sK satisfaz a propriedade [P3], então o conjunto *aventureirosMovem* contém todos os agentes que controlam os aventureiros, exceto aqueles que foram eliminados do jogo. Como sK satisfaz [P2], então $nExec(a) = nExec(b)$, para quaisquer a e b que pertençam ao conjunto *aventureirosMovem*. Na ação *novaDirecao*, a condição $aventureiros(self)$ será falsa, para qualquer agente aventureiro. Assim suas atualizações não serão computadas no estado sK, e $nExec(a)$ em $sK + 1$ terá o mesmo valor que $nExec(a)$ em sK.

   As atualizações da regra de transição do módulo *MundoWumpus* poderão ser executadas, uma vez que o conjunto *aventureirosJogam* é vazio. Se essa regra não for executada, o estado $sK + 1$ será o mesmo de sK, e assim obviamente continuará satisfazendo todas as propriedades.

   Vamos analisar o caso em que essa regra é executada no estado sK.

   A regra especifica comandos que devem ser executados para todos os elementos do conjunto *aventureirosMovem* - neste caso, para todos os agentes que controlam os aventureiros, exceto aqueles que foram eliminados do jogo.

   Se algum aventureiro for vencedor, não será gerado um novo estado $sK + 1$, pois a execução irá terminar. Caso contrário, a regra verifica se cada agente deverá ser eliminado do jogo.

   Cada agente que controla um aventureiro que não será eliminado, será então retirado do conjunto *aventureirosMovem* e inserido no conjunto *aventureirosJogam*.

   No estado seguinte, $sK + 1$, teremos:

   [P2] válida, porque *nExec* tem os mesmos valores para todos os agentes ainda não removidos do jogo, e todos esses agentes agora irão pertencer ao mesmo conjunto *aventureirosJogam*.

   [P3] válida, pois todos os agentes ainda não removidos do jogo foram transferidos para *aventureirosJogam* e o conjunto *aventureirosMovem* estará vazio.

2. O conjunto *aventureirosJogam* não é vazio. Neste caso, a condição que precede a regra do módulo *MundoWumpus* não é satisfeita. Assim, comandos daquele módulo não são executados no estado sK. Por outro lado, um ou mais agentes que controlam os aventureiros podem executar o codigo da ação *novaDirecao* com a condição $aventureirosJogam(self)$ sendo avaliada como verdadeira.

   Suponha que exista um conjunto de agentes $a1, a2, ..., aN$, onde N pode ser zero, que executem a ação *novaDirecao*, nas condições descritas acima.

   Como sK satisfaz a propriedade [2], então $nExec(a1) = ... = nExec(aN) = V$ e $nExec(b) = V + 1$, para qualquer agente b que pertencer ao conjunto aventureirosMovem.

No estado $sK + 1$ teremos:

- [P2] válida, pois cada agente $a1, a2, ..., aN$, que neste instante é um elemento do conjunto *aventureirosJogam*, será transferido para o conjunto *aventureirosMovem*. Como esses agentes executaram com sucesso o código da ação *novaDirecao*, temos, para $i = 1, ..., N$, $nExec(ai) = V + 1 = nExec(b)$, para qualquer agente b que pertencer ao conjunto aventureirosMovem.
- [P3] válida, pois a união dos conjuntos *aventureirosJogam* e *aventureirosMovem* continua inalterada, apenas alguns elementos são transferidos do primeiro conjunto para o segundo.

Com esses argumentos, podemos afirmar que as propriedades [P2] e [P3] são válidas para todos os estados de uma execução do sistema especificado. Sendo assim, a propriedade [P1] também será válida e podemos concluir que esse sistema implementa um esquema que satisfaz critérios de justica para execução dos agentes que controlam os aventureiros.

5.6.3 Outras Propriedades

Várias outras propriedades interessantes podem ser demonstradas. Por exemplo, seria interessante mostrar que a regra de transição do módulo *MundoWumpus* nunca é executada com sucesso (com o conjunto aventureirosJogam vazio) duas vezes seguidas, sem que todos os agentes que controlam os aventureiros tenham executado, com sucesso, a ação *novaDirecao*. Isso quer dizer que os aventureiros só são movidos nas cavidades da caverna após cada agente que os controla decidir a direção de movimento.

Uma demonstração para essa propriedade poderia seguir os mesmos parametros usados na demonstração da segunda propriedade avaliada nesta seção.

As demonstrações apresentadas mostram como o uso de ASM em especificações facilita uma definição formal de propriedades sobre sistemas, e a prova de que essas propriedades são válidas.

5.7 Um Problema de Eficiência

Uma estrutura que desperta a atenção quanto a eficiênca na execução dos códigos do jogo *O Mundo Wumpus* é o tipo *Cavidade*, declarado no módulo *CavernaBase*.

Como ocorre nas linguagens orientadas a objetos, o módulo CavernaBase aloca memória para armazenar uma cavidade, quando solicitado por um módulo que use essa estrutura, e retorna uma referência para esta posição de memória [50]. No nosso caso, o módulo *MundoWumpus* é quem faz uso direto da estrutura cavidade.

```
module CavernaBase
...
  type
     public Cavidade is (x:  Int; y:  Int);
...
```

```
module MundoWumpus
...
  //ação que permite o caminhamento de um agente aventureiro

  caminha (av:  Agent):=
 ...
   case direcaoAgente(av) of

    Norte ⇒
     if coluna_av <  LimiteColuna then
      avInfor(av).posicao := Cavidade(linha_av, coluna_av + 1 )

    Sul ⇒
       ...
...
```

Figura 5.5: Partes dos módulos.

Veja, na Figura 5.5, partes das implementações dos módulos:

No código do módulo CavernaBase, na Figura 5.5, é importante observar que, apesar do tipo *Cavidade* ser público, os inteiros x e y não o são. Para que os inteiros fossem públicos, a declaração, em Machĭna, deveria ser da seguinte maneira:

$$public\ Cavidade\ is\ (public\ x : Int;\ public\ y : Int);$$

permitindo que todos os módulos que importassem a estrutura *Cavidade* pudessem alterar os valores da dupla sem a necessidade, para isso, de realocar outra área de memória.

No nosso caso, porém, caso os inteiros x e y fossem públicos, quando o aventureiro recebesse uma referência para esses valores para consulta de sua posição, esse agente poderia "roubar" no jogo, alterando sua posição para qualquer outra cavidade sem passar pelas cavidades vizinhas.

Para evitar o "roubo", o código da ação *caminha*, ao mudar a posição de um aventureiro, dado pelo campo *posicao*, utiliza o construtor *Cavidade* para gerar a nova posição do aventureiro.

Porém, a chamada do construtor *Cavidade* exige mais esforço computacional ao descartar a memória da posição atual do aventureiro, anteriormente alocada, para alocar uma segunda e retornar essa nova referência. É importante também ressaltar que essa tarefa ocorre a cada ciclo do jogo para todos os aventureiros, já que eles não permanecem parados na cavena no decorrer do jogo.

Uma solução possível para contornar esse problema seria eliminar o módulo *CavidadeBasica* e incorporar suas regras no módulo *MundoWumpus*. Para nosso problema, essa estratégia seria suficiente, visto que, assim, o módulo *MundoWumpus* seria proprietário dos inteiros que formam a dupla *Cavidade*. Porém, apesar de ser possível eliminar o módulo *CavernaBase* da solução de nosso jogo, essa solução pode não se aplicar a outras situações onde um módulo não possa ser eliminado, por questões lógicas ou apenas de organização do código.

Uma sugestão que poderia ser suficiente para eliminar nosso problema seria uma nova construção na linguagem Machĭna que, uma vez introduzida no módulo *CavernaBase*, o módulo *MundoWumpus* passaria a ter acesso irrestrito às suas estruturas. Na linguagem C++, podemos ver uma estrutura dessa natureza referenciada pela palavra chave *friend*.

Como a linguagem Machĭna ainda está em fase de desenvolvimento, a inclusão da construção *friend* em sua estrutura pode ser avaliada pelos seus idealizadores, o que traria, assim, mais poder à linguagem e mais flexibilidade aos programadores.

5.8 Conclusão

Ao contrário do Capítulo 4, onde as regras do MasterMind foram formalizadas em ASM para que, em seguida, fossem implementadas em AsmL, o uso de Machĭna

na formalização e implementação do jogo *O Mundo Wumpus* foi suficiente para especificar, com clareza, as regras do jogo, além de permitir a geração de códigos executáveis.

Isso pode ser entendido facilmente já que Machĩna é uma linguagem que foi desenvolvida fortemente baseada no paradigma de ASM.

Um recurso importante que a linguagem Machĩna implementa é o suporte aos sistemas multiagentes. Com esse recurso, pouco esforço foi realizado na implementação de uma solução para a versão multiagentes do jogo *O Mundo Wumpus*, onde a sincronia das jogadas dos aventureiros foi tratada com clareza e simplicidade.

Uma sugestão para enriquecer a linguagem Machĩna é a incorporação de um recurso que permita aos módulos estabelecer relações de "amizade" entre si, assim como ocorre entre classes de algumas linguagens como, por exemplo, C++. Dessa forma, o programador teria mais flexibilidade ao tratar a visibilidade em seus módulos, evitando, também, problemas de eficiência, como discutido na Seção 5.7

Finalmente, espera-se que esforços sejam dispensados no desenvolvimento e aprimoramento de ferramentas que permitam que os códigos escritos em Machĩna sejam convertidos em executáveis com mais facilidade, o que faria da linguagem Machĩna uma excelente opção para a formalização de jogos de computadores, além de outras aplicações não discutidas pelo trabalho.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Conclusões do Trabalho

O trabalho desenvolvido enfatizou a utilização de ASM na formalização das regras de jogos de computadores e seus agentes, assim como a implementação desses sistemas com o uso de linguagens baseadas no paradigma das ASM.

Usando os jogos *Mastermind* e *O Mundo Wumpus* como base para a experimentação, o trabalho mostra que o formalismo de ASM possibilitou especificar as regras desses jogos com clareza e elegância, além de reduzir as possíveis ambigüidades no entendimento dos jogos como pode ocorrer quando as regras são especificadas apenas com o uso de linguagens naturais.

É importante ressaltar, porém, que, como ocorre com qualquer técnica utilizada para especificar sistemas, um fator necessário para garantir as vantagens no uso das ASM na especificação das regras dos jogos, assim como no comportamento dos agentes envolvidos, é a garantia de que os conceitos das ASM estejam sob domínio das pessoas que desenvolvem ou fazem uso de tais especificações.

Em relação ao ensino de inteligência artificial, as ASM surgem como uma excelente alternativa para a formalização dos jogos de computadores utilizados em aulas e trabalhos. Como as ASM descrevem especificações matemáticas muito próximas às linguagens de programação convencionais, seu aprendizado se dá de forma direta e simples para os alunos que já convivem com a programação baseada em linguagens do paradigma imperativo.

Assim, uma vez formalizadas em ASM, é possível mapear as regras dos jogos para linguagens de programação e gerar, conseqüentemente, códigos executáveis. Dentre as opções de linguagens para essa tarefa, podemos citar as linguagens específicas desenvolvidas no paradigma de ASM, como descrito no Capítulo 3.

O uso das linguagens de programação baseadas no formalismo de ASM e seus ambientes podem ser uma boa opção para o estímulo do aprendizado de ASM. Outra vantagem que elas oferecem é a possibilidade de gerar especificações executáveis

para os sistemas, o que pode ser aplicado aos jogos de computadores no ensino de inteligência artificial.

Contudo, a linguagem Machĭna, que demonstrou grande importância para o sucesso do trabalho, não oferece, até o momento, um compilador estável. Essa colocação é importante visto que, dentre as linguagens avaliadas, como discutido no Capítulo 3, a que possui definições mais próximas dos conceitos das ASM é a linguagem Machĭna. Isso sugere que Machĭna, associada a um bom compilador e um ambiente de programação, possa vir a ser usada, em um futuro próximo, como um suporte no aprendizado do formalismo ASM.

Além de oferecer especificações formais para os jogos desenvolvidos, ASM permitiu construir demonstrações formais de algumas propriedades importantes para os jogos, como visto na Seção 5.6.

Finalmente, pode-se chegar a uma conclusão importante se compararmos os códigos gerados pelas linguagens lógicas, como o Prolog, com os códigos gerados pelas linguagens do paradigma de ASM, como Asml e Machĭna. Por estarem mais próximos das linguagens do paradigma imperativo, as linguagens baseadas em ASM geram códigos mais eficientes que os códigos das linguagens lógicas. Uma explicação para essa característica pode ser dada pelas desvantagens que a arquitetura de Von Neumann oferece ao empilhamento de chamadas recursivas de funções que não são vitais às linguagens baseadas em ASM, como ocorre com as linguagens do paradigma lógico.

6.2 Principais Contribuições

Como discutido no Capítulo 2, vários exemplos de utilização com sucesso do formalismo de ASM na especificação formal de sistemas podem ser encontrados na literatura, dentre os quais podemos citar: arquiteturas [8, 9, 12], linguagens de programação [13, 14, 15, 32, 52], sistemas distribuídos [5, 6, 10, 33] e de tempo real [28, 34], entre outros [11].

O desenvolvimento deste trabalho, antes de mais nada, torna-se mais uma evidência da importância das ASM no que se refere à formalização e provas de sistemas. Desta vez, as ASM foram aplicadas na especificação de jogos de computadores e no comportamento de seus agentes inteligentes. Como resultado, foi observado que as ASM ofereceram facilidades ao especificar tais sistemas, passando a ser mais uma opção para diminuir a distância entre o entendimento dos jogos e o aprendizado de inteligência artificial.

Ver as ASM como opção no ensino de inteligência artificial também enfatiza a importância da linguagem Machĭna. Com esta linguagem, o aprendizado das ASM pode se dar de forma interativa e prática, além de oferecer um ambiente para testes dos sistemas formalizados. Assim, o trabalho pode ser visto como mais uma justificativa, dentre outras, para que os idealizadores da linguagem Machĭna

invistam esforços no desenvolvimento de ferramentas que ofereçam um bom ambiente de trabalho para essa linguagem.

Ainda referente à linguagem Machĩna, o trabalho sugere novas estruturas a esta linguagem. Com essas possíveis mudanças, Machĩna pode se tornar mais poderosa e flexível na especificação de sistemas, como discutido no Capítulo 5.

Outra contribuição importante refere-se à avaliação das linguagens baseadas em ASM oferecidas aos desenvolvedores. Dentre as mais notórias, questiona-se a possível utilização da linguagem AsmL [22, 21] no aprendizado do formalismo de ASM. Ficou claro que algumas construções e recursos dessa linguagem permitem que suas especificações se distanciem dos conceitos do formalismo das ASM, como discutido nos Capítulos 3 e 4.

Para os interessados em aprender ASM, os Capítulos 2, 3, 4 e 5 podem ser vistos como um excelente ponto de partida, oferecendo um resumo do formalismo de ASM, uma avaliação de ferramentas de programação do paradigma e, finalmente, exemplos de sua utilização, respectivamente.

6.3 Trabalhos Futuros

Por ser pioneiro na aplicação de ASM nos jogos de computadores e inteligência artificial, o trabalho desperta atenções para uma área até então não investigada pela comunidade que estuda o formalismo de ASM. Os algoritmos genéticos surgem como a primeira técnica de IA especificada com sucesso em ASM, abrindo campo para que muitas outras técnicas possam ser formalizadas com esse poderoso método formal.

Como trabalho futuro, a especificações de outras técnicas de IA com o uso de ASM pode permitir que este formalismo ofereça material didático adequado ao ensino de inteligência artificial.

Outro possível trabalho pode ser definido para desenvolver estudos que avaliem a viabilidade do ensino de ASM em instituições de ensino superior. Além de permitir a formalização das regras de jogos de computadores em aulas e competições de IA, as ASM podem ser mais uma opção para o desenvolvimento de provas de propriedades de variados conteúdos didáticos.

Finalmente, pode-se estabelecer um estudo que priorize a avaliação das bibliotecas gráficas que oferecem suporte às linguagens de programação baseadas em ASM. Um estudo deste tipo poderia sugerir estruturas a estas bibliotecas que tornassem possível o desenvolvimento de jogos executáveis mais sofisticados em ASM como, por exemplo, os jogos em três dimensões.

Referências Bibliográficas

[1] *Tcl/Tk 8.4 Manual.*
http://www.tcl.tk/man/tcl8.4/.

[2] R. Adobbati, A. N. Marshall, A. Scholer, S. Tejada, G. Kaminka, S. Schaffer, and C. Sollitto. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the International Conference on Autonomous Agents (Agents-2001) - Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[3] M. Anlauff. Xasm – An Extensible, Component-Based Abstract State Machines Language. In *Proceedings of the ASM 2000 Workshop*, pages 1–21, Monte Veritá, Switzerland, March 2000.

[4] J. C. H. Barcellos. Algoritmos Genéticos Adaptativos: Um estudo comparativo. Master's thesis, USP - São Paulo, 2000.

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

[6] D. Bèauquier and A. Slissenko. The Railroad Crossing Problem: Towards Semantics of Timed Algorithms and their Model-Checking in High-Level Languages. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *LNCS*, pages 201–212. Springer, 1997.

[7] J. L. Bernier, C. I. Herráiz, J. J. Merelo, S. Olmeda, and A. Prieto. Solving mastermind using gas and simulated annealing: a case of dynamic constraint optimization. In *In Proceedings PPSN, Parallel Problem Solving from Nature IV, LNCS 1141*, pages 554–563, Springer-Verlag, 1996.

[8] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer, 1995.

[9] E. Börger and U. Glässer. Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In Y. Gurevich and E. Börger, editors, *Evolving Algebras - Mini-Course, BRICS Technical Report (BRICS-NS-95-4)*, pages 128–153. University of Aarhus, Denmark, July 1995.

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

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

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

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

[14] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim and J. Gruska and J. Zlatuska, editor, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic*, number 1450 in LNCS. Springer, August 1998.

[15] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[16] A. J. Champandard. *Artificial Intelligence - WAREHOUSE IA depot.* http://artificialintelligence.ai-depot.com/Essay/DeepBlue.html.

[17] A. R. da Silva. *Filosofia Contemporânea.* http://www.geocities.com/discursus/textos/dreyfus.html.

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

[19] V. O. Di Iorio, A. P. Oliveira, E. C. Miguel, R. S. Bigonha, and M. A. S. Bigonha. Utilização de Máquinas de Estado Abstratas em Aplicações de Inteligência Artificial e Jogos. In *XXIX CLEI - Conferência Latino Americana De Informática*, La Paz, Bolivia, September 2003.

[20] M. H. V. Emden. Relational Programming Illustrated by a Program for the Game of Mastermind. Technical Report CS-78-48, Department of Computer Science, University of Waterloo,Canada, 1978.

[21] Foundations of Software Engineering - Microsoft Research. *Introducing AsmL: A Tutorial for the Abstract State Machine Language*, December 2001. http://research.microsoft.com/fse/asml/doc/StartHere.html.

[22] Foundations of Software Engineering - Microsoft Research. *AsmL - The Abstract State Machine Language*, October 2002. http://research.microsoft.com/fse/asml/doc/StartHere.html.

[23] Gary Darby. *Mastermind.* http://www.delphiforfun.org/Programs/MasterMind.htm.

[24] D. Gemmer. AI Wars (the insect mind) - game review. Gamesdomain, 1997. http://www.gamesdomain.com/gdreview/zones/reviews/pc/jul97/bugs.html.

[25] J. Gerstmann. Unreal Tournament : Action game of the year. Gamespot, 1999. www.gamespot.com/features/1999/p3_01a.html.

[26] A. Gill. *Applied Algebra for the Computer Sciences.* Prentice Hall, Englewood Cliffs, 1976.

[27] P. Glanville. Half-Life - game review. Gamesdomain, 1998. http://www.gamesdomain.com/gdreview/zones/reviews/pc/dec98/halflife.html.

[28] U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[29] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[30] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[31] Y. Gurevich. The Sequential ASM Thesis. Technical Report MSR-TR-99-09, Microsoft Research, February 1999.

[32] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[33] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of *LNCS*, pages 266–290. Springer, 1996.

[34] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 295–328. Oxford University Press, 1995.

[35] J. Huggins and R. Mani. Evolving Algebras Interpreter v. 2.0. Technical report, MIT, 1992.

[36] J. K. Huggins. *ASM Michigan web page.* http://www.eecs.umich.edu/gasm.

[37] M. P. Jones. *Gofer Archive.* http://www.cse.ogi.edu/~mpj/goferarc/index.html.

[38] G. A. Kaminka, M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshal, A. S. Scholer, and S. Tejada. Gamebots: the ever-challenging multi-agent research test-bed. *Communications of the ACM*, January 2002.

[39] S. Li. Robocode: Advanced Robot Battle Simulation Engine. IBM developerWorks, 2002. http://www-106.ibm.com/developerworks/java/library/j-robocode/index.html.

[40] Z. Michalewicz, R. Hinterding, and M. Michalewicz. *Evolutionary Algorithms, Chapter 2 in Fuzzy Evolutionary Computation.* W. Pedrycz, Kluwer Academic, 1997.

[41] M. Norddahl and K. Gangstoe. Clanlib, a multi-platform game development library. http://www.clanlib.org/intro.html, 2004.

[42] A. P. Oliveira, A. Moreira, and V. O. Di Iorio. Uma Avaliação do Uso de Jogos no Ensino de Inteligência Artificial. In *Anais do WEIMIG 2004*, Belo Horizonte, Outubro 2004.

[43] A. M. C. Pereira. *Damas Clássicas.* http://paginas.fe.up.pt/~eol/IA/DAMASII/.

[44] F. Provo. Doom II - game review. Gamespot, 2002. http://www.gamespot.com/gba/action/doom2/review.html.

[45] P. Rosenbloom, J. Laird, and A. Newell. *The Soar Papers.* MIT Press, 1993.

[46] S. Russel and P. Norvig. *Artifical Intelligence: A Modern Approach.* Prentice Hall, New Jersey, 1995.

[47] J. Schmid. Introduction to Asmgofer. http://www.tydo.de/AsmGofer/files/AsmGoferIntro.pdf, March 2001.

[48] T. Sweeney and A. Moise. UnrealScript Language Reference. Epic MegaGames, Inc., 1998. http://unreal.epicgames.com/UnrealScript.htm.

[49] P. Sweetser. Current AI in Games: A Review, 2002. http://www.itee.uq.edu.au/~penny/Game%20AI%20Review.pdf.

[50] F. Tirelo, R. Bigonha, M. A. Maia, and V. Iorio. Machĭna: A Linguagem de Especificação de ASM. Technical Report 08/1999, Laboratório de Linguagens de Programação, Universidade Federal de Minas Gerais, 1999.

[51] F. Tirelo, R. Bigonha, M. A. Maia, and V. Iorio. Tutorial em Máquinas de Estado Abstratas. In *Anais do III Simpósio Brasileiro de Linguagens de Programação*, Porto Alegre, Maio 1999.

[52] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[53] S. Woodcock. Games with Extensible AI, 2004. http://www.gameai.com/exaigames.html.

Apêndice A

O MasterMind

Uma leitura mais refinada no código do jogo MasterMind exibido neste apêndice pode revelar diferenças entre o código aqui apresentado com as formalizações em ASM para o jogo discutidas no Capítulo 4.

Ao iniciar o trabalho, o MasterMind foi o primeiro sistema desenvolvido, o que deu abertura para o amadurecimento de novas idéias para sua formalização durante a escrita do texto. Outra justificativa para esse fato pode ser atribuída à linguagem AsmL que dá abertura para construções que podem se distanciar do formalismo de ASM, como discutido também no Capítulo 4

```
Classe Básica: Responsável pela configuração e
valores estáticos do jogo
Programador: Eliseu César Miguel
Orientador: Vladimir Oliveira Di Iorio
Data Inicial: 28-01-2004

enum Status_Do_Jogo
    Jogo_Em_Andamento
    Jogo_Finalizado
    Jogo_Ganho
    Jogo_Perdido

class Basica

  type Cromossomo = Map of Integer to Integer

  //estrutura utilizada para representar o resultado
  //de uma tentativa de acerto do c'odigo
  structure DuplaInt_Pegs
     PCertas as Integer
```

```
    PErradas as Integer

// Valores estaticos durante a execucão do sistema
protected Cores = {0..7}     //conjunto de cores do sistema
protected MaxJogadas = 13    //maximo de jogadas permitido
protected NCombina = 4       //tamanho do codigo secreto
// variaveis
private var aux as Map of Integer to String = {->}

//Metodo responsavel por definir quantas cores estão na
//posicão correta em relacao ao codigo
private avaliaPosCertas(tenta as Cromossomo,
                        codigo as Cromossomo) as Integer
  var indice = 0
  var PCertas = 0
  step while indice < NCombina
      if tenta(indice) = codigo (indice) then
          aux(indice) := "PosCerta"
          PCertas := PCertas + 1
      else
          aux(indice) := "Livre"
      indice := indice + 1
  step
      return PCertas

//Metodo utilizada para, dado um elemento (indice) do codigo,
//localiza um elemento de mesmo valor na tentativa e que ainda
//não tenha sido realcionado a outro elemento do codigo
private localizaCorPosErrada (i as Integer, tenta as Cromossomo,
                              codigo as Cromossomo) as Integer
        var j = 0
        var encontrou = 0
        step
          if aux(i) <> "PosCerta" then
             step while j < NCombina
                if (aux(j) = "Livre") and (tenta(j) = codigo(i)) then
                     aux(j) := "PosErrada"
                     encontrou := 1
                     j := NCombina
                else
                   j := j+1
        step
```

```
            return encontrou

  //Metodo utilizado para descobrir cores certas em posicao errada
  private avaliaPosErradas (tenta as Cromossomo,
                                codigo as Cromossomo) as Integer
    var i= 0
    var PErradas = 0
    step while i < NCombina
          PErradas := localizaCorPosErrada (i, tenta, codigo) + PErradas
          i := i+1
    step
       return PErradas

  //Metodo utilizado por outros modulos para obter o resultado completo
  //da avaliacao da tentativa em relacao ao codigo
  protected avaliaTentativa (tenta as Cromossomo,
                             codigo as Cromossomo) as DuplaInt_Pegs
    var Result as DuplaInt_Pegs = DuplaInt_Pegs (0, 0)
    step
       Result.PCertas := avaliaPosCertas (tenta, codigo)
    step
        Result.PErradas := avaliaPosErradas(tenta, codigo)
    step
       return Result
```

```
Sistema MasterMindASM
Classe Tabuleiro: Responsável pelo código secreto e controle de jogadas
Programador: Eliseu César Miguel
Orientador: Vladimir Oliveira Di Iorio
Data Inicial: 04-01-2004

class Tabuleiro extends Basica
  //inicializacao do codigo secreto
  private codigo as Cromossomo = geraCodigo()
  private var contaJogadas = 0
  private var Resultado as DuplaInt_Pegs = DuplaInt_Pegs(0,0)

  // Metodo que nao pode existir de forma publica para garantira a
  //invisibilidade do codigo secreto
  public imprimeCodigo()
```

```
    Write(" Codigo Secreto  ")
    step foreach i in [0..NCombina-1]
      Write(codigo(i)+" ")
    step
      WriteLine(" ")

  // Metodo que gera um codigo secreto aleatorio
  private geraCodigo () as Cromossomo
    var i as Integer = 0
    var cod as Cromossomo = {->}
    step while i < NCombina
        choose j in Cores
        cod (i) := j
        i := i + 1
    step
       return cod

  // Metodo utilizado pelo tabuleiro para avaliar
  //a tentativa com o codigo
  public avaliaTentativaComCodigo (tenta as Cromossomo)
                              as Status_Do_Jogo
    initially status as Status_Do_Jogo = Jogo_Em_Andamento
    step
      if contaJogadas >= MaxJogadas then
         status := Jogo_Finalizado
      else
        step
          Resultado := avaliaTentativa(tenta, codigo)
          contaJogadas := contaJogadas + 1
        step
          if (Resultado.PCertas = NCombina) then   status:= Jogo_Ganho
          elseif (contaJogadas = MaxJogadas) status:= Jogo_Perdido
    step
      return status


  //Metodo para tornar visivel o resultado de uma tentativa
  public ForneceResultadoTentativa() as DuplaInt_Pegs
      return Resultado


Sistema MasterMindASM
```

```
Classe Servicos: Dispõe serviços básicos ao Agente Jogador
Programador: Eliseu César Miguel
Orientador: Vladimir Oliveira Di Iorio
Data Inicial: 04-01-2004
class Servicos extends Basica


  structure Individuo
     corpo as Cromossomo
     pontuacao as Integer

  structure Tentativa
     corpo as Cromossomo
     Pegs as DuplaInt_Pegs

  type CadeiaCromossomo = Map of Integer to Individuo


  type cadeiaTentiva = Map of Integer to Tentativa

  //novo Metodo para gerar novo individuo aleatorio
  protected novoIndividuo (tamanho as Seq of Integer)
                                          as Individuo
    var ind as Individuo = Individuo ({->},0)
    step
      step foreach todos in tamanho
          choose aleatorio in Cores
              ind.corpo(todos) := aleatorio
    step
      return ind
  //-------------------------------------//
  // faz a operacao matematematica de modulo
  public modulo (numero as Integer) as Integer
    if numero < 0 then
      return (-1* numero)
    else
      return numero

  //-------------------------------------//
  protected geraIndividuoVazio() as Cromossomo
     ListaNCombina = [0..NCombina]
     var individuo as Cromossomo ={->}
```

```
    step
      forall i in ListaNCombina
         individuo(i) := 0
    step
      return individuo
//-------------------------------------------------------
protected zeraPontuacao (popula as CadeiaCromossomo,
          tamanhoPop as Seq of Integer) as CadeiaCromossomo
    var pop as CadeiaCromossomo = popula

     step
      forall i in tamanhoPop
         pop(i).pontuacao:= 0
    step
      return pop

//-------------------------------------------------------
// Como trabalhar em paralelo e usar o poder dos conjuntos
//para solucionar essa implementacao
protected localizaInd (ind as Cromossomo, pop as CadeiaCromossomo,
                       tamanho as Seq of Integer) as Boolean
  var achou as Boolean = false
  step foreach contador in tamanho
      if ind = pop(contador).corpo then
         achou := true
  step
    return achou

//-------------------------------------------------------------
//auxiliar apenas para impressão da tentativa
protected imprimeTentativa (his as cadeiaTentiva, pos as Integer)
    Write (" tentativa " + (pos+1) + ":   ")
    step foreach i in [0..NCombina-1]
      Write ((his(pos).corpo(i)) + " ")
    step
      WriteLine("  "+his(pos).Pegs.PCertas +
                  " " + his(pos).Pegs.PErradas)

//-------------------------------------------------------
//metodo usado para a transposicao que sera realizada
//no fim da reproducao
protected transposicao(individuo as Cromossomo) as Cromossomo
```

```
    var ind = individuo
    step
       choose quebra in {0..NCombina-1} where quebra >= 1
          forall pos in {0..NCombina-1}
             ind(pos) := ind((pos + NCombina - quebra) mod NCombina)
    step
       return ind

Sistema MasterMindASM
Classe AgenteJogador: Agente que tentará descobri o código secreto
Programador: Eliseu César Miguel
Orientador: Vladimir Oliveira Di Iorio
Data Inicial: 04-01-2004
class AgenteJogador extends Servicos

  //Auxiliar para fornecer o valor maximo de fitness a ser
  //aproveitado e a quantidade
  structure ValorEQuantidade
     Valor as Integer
     Quantidade as Integer
  //configuracoes estáticas do agente
  private MedidaDeCorte = 100      //numero de individuos reprodutores
  private MedidaTransposicao = 10 //numero de individuos para transposicao
  private posGeracao = 0           //posiçao onde os filhos serão incluídos
  private tamanhoPopulacao = 300  //quantidade de indivíduos na populacao
  //variaveis
  private var NTentativa = 0       //guarda o índice da tentativa atual
  private var populacao as CadeiaCromossomo = geraPopulacaoInicial ()
  private var reprodutores as CadeiaCromossomo = {->}
  private var historico as cadeiaTentiva = {->}
  //controla a funçao de fitness
  private var controleFitness as Cromossomo = {->}
  private tab as Tabuleiro


   //-------------------------------------------------------------
  // Metodo para funcionar como construtor em orientacao a objetos
  public inicio()
     var ind as Cromossomo = {->}
     var status as Status_Do_Jogo = Jogo_Em_Andamento
     step
       WriteLine("")
```

```
  WriteLine("--- *** MasterMind Genetico em AsmL *** ----")
  WriteLine("")
  WriteLine("           Configuracoes do Jogo ")
  WriteLine("")
  WriteLine("Conjunto de cores:     "+ Cores )
  WriteLine("Tamanho do codigo:     " + NCombina)
  WriteLine("Numero de Tentativas:  "+ MaxJogadas)
  WriteLine("")
  WriteLine("Creditos:  Professor Vladimir DPI/UFV e Eliseu DCC/UFMG")
  WriteLine("-------------------------------------------------- ")
  WriteLine(" " )
  tab.imprimeCodigo()
  WriteLine(" " )

step
   choose i in {0..tamanhoPopulacao}
      ind := populacao(i).corpo
step
   status := jogaTentativa(ind)
step
   imprimeTentativa(historico, NTentativa -1)
step while status =  Jogo_Em_Andamento
   step
      populacao := pontuaPopulacaoHistorico (historico, populacao)
   step
      controleFitness := atualizaControle (populacao)
   step
      R = defineCorte(controleFitness, MedidaDeCorte)
      geraReprodutores(R)
   step
      reproduz()
   step
      populacao := pontuaPopulacaoHistorico (historico, populacao)
   step
      controleFitness := atualizaControle(populacao)
   step
      ind := indicaTentativa(controleFitness)
   step
      status := jogaTentativa(ind)
   step
      imprimeTentativa(historico, NTentativa -1)
step
```

```
        WriteLine("---------------------------------------------------- ")
        WriteLine(" " )
        WriteLine(" Resultado: " + status)
        WriteLine(" " )

  //-------------------------------------------------------------------------
  // Metodo que pontua um individuo da populacao a partir da fun'cao de fitnes
  private funcaoFitness(ind as Individuo, falsoCodigo as Tentativa) as Integer
      peg = avaliaTentativa (ind.corpo, falsoCodigo.corpo)
      return   ind.pontuacao - (modulo(falsoCodigo.Pegs.PCertas - peg.PCertas)
               + modulo(falsoCodigo.Pegs.PErradas - peg.PErradas))

  //-------------------------------------------------------------------
  private pontuaPopulacao (falsoCodigo as Tentativa,
                       populacao as CadeiaCromossomo) as CadeiaCromossomo
     var pop as CadeiaCromossomo = populacao
     step
        step foreach indice in [0..tamanhoPopulacao]
           pop(indice).pontuacao := funcaoFitness(pop(indice), falsoCodigo)
     step
        return pop

  //-------------------------------------------------------------------
  private pontuaPopulacaoHistorico (hist as cadeiaTentiva,
                         populacao as CadeiaCromossomo) as CadeiaCromossomo
     var pop as CadeiaCromossomo = populacao
     var contador as Integer = 0
     step
        pop := zeraPontuacao (pop, [0..tamanhoPopulacao])
     step
        step while contador < NTentativa
          contador := contador + 1
          pop := pontuaPopulacao (hist(contador), pop)
     step
       return pop

  //-------------------------------------------------------------
  private jogaTentativa(individuo as Cromossomo) as Status_Do_Jogo
     var ind = individuo
     var status as Status_Do_Jogo = Jogo_Em_Andamento
     step
       status := tab.avaliaTentativaComCodigo (ind)
```

```
    step
      pegs = tab.ForneceResultadoTentativa ()
      historico(NTentativa):= Tentativa(ind, pegs)
      NTentativa := NTentativa + 1
      return status
//-----------------------------------------------------------------

//Novo Metodo usado para gerar a populacao aleatoria inicial
private geraPopulacaoInicial () as CadeiaCromossomo
   var pop as CadeiaCromossomo = {->}
   var indice as Integer = 0

   step
      step foreach coluna in [0..tamanhoPopulacao]
         ind = novoIndividuo ([0..NCombina-1])
         pop(coluna) := ind
         indice:=indice + 1
   step
      return pop

//-----------------------------------------------------------------
private atualizaControle (pop as CadeiaCromossomo) as Cromossomo
  var controle as Cromossomo = {->}
  var pontos as Integer = 0
  step foreach contador in [0..tamanhoPopulacao]
     step
        pontos := pop(contador).pontuacao
     step
        if (pontos in Indices(controle)) then
            controle(pontos):= controle(pontos) + 1
        else
            controle(pontos):=1
  step
     return controle

//-----------------------------------------------------------------
//Metodo responsavel por decidir ate qual valor maximo de
// fitness sera aporveitado para gerar
//individuos reporodutores e qual a quantidade maxiam de
//individuos com esse fitnes vingara
private defineCorte (controle as Cromossomo,
                     corte as Integer) as ValorEQuantidade
```

```
    var indiceAuxiliar as Integer = 0
    var indice as Integer = 0
    var quantidade as Integer = 0
    var soma as Integer = 0
    step while soma < corte
      indiceAuxiliar := indiceAuxiliar - 1
      if (indiceAuxiliar in Indices(controle)) then
        step
            soma := soma + controle(indiceAuxiliar)
            indice:=indiceAuxiliar
        step
         if (soma >= corte)
            quantidade := controle(indiceAuxiliar) - (soma - corte)

    step
      R as ValorEQuantidade = ValorEQuantidade (indice, quantidade)
    step
      return R

  //-------------------------------------------------------------
  //gera reprodutores e inicia a populacao com os reprodutores
  private geraReprodutores (parametros as ValorEQuantidade)
     var quantValMax as Integer = parametros.Quantidade
     var valorMaximo as Integer = parametros.Valor
     var contador as Integer = 0
     pontoInicial = 0
     step
        reprodutores := {->}
     //escolhe os individuos
     step foreach todos in [0..tamanhoPopulacao]

       if populacao (todos).pontuacao > valorMaximo  then
           reprodutores(contador):=
                              Individuo(populacao (todos).corpo, pontoInicial)
           contador := contador + 1
       elseif  populacao(todos).pontuacao =
                                valorMaximo and quantValMax > 0 then
           reprodutores(contador):=
                              Individuo(populacao (todos).corpo, pontoInicial)
           contador := contador + 1
           quantValMax := quantValMax - 1
```

```
    //realiza a transposicao nos elementos restantes
    //para completar o conjunto da populacao
    step while contador < MedidaTransposicao + MedidaDeCorte
       choose alguem in {0..NTentativa-1}
          reprodutores(contador):=Individuo(transposicao(
                                    historico(alguem).corpo),pontoInicial)
          contador := contador + 1
    step
       populacao := reprodutores

//-------------------------------------------------------------------------
//Metodo responsavel por realizar a reproducao
private reproduz()
    var novoIndividuo as Cromossomo = {->}
    PontoInicial = 0
    //utilizado para setar valores aleatorios no individuo novo
    var indice as Integer = 0
    var indiceDoNovo as Integer = MedidaDeCorte + MedidaTransposicao
    step while indiceDoNovo <= tamanhoPopulacao
       //gera um individuo a partir dos reprodutores
       step while indice < NCombina
            choose aleatorio in {0..MedidaDeCorte-1}
                novoIndividuo(indice):= reprodutores(aleatorio).corpo(indice)
            indice:=indice + 1

       step
          //insere individuo caso não haja outro na populacao
          if not (localizaInd (novoIndividuo, populacao,
                  [0..indiceDoNovo - 1])) then
             populacao (indiceDoNovo) := Individuo(novoIndividuo, PontoInicia
             indiceDoNovo := indiceDoNovo + 1
          novoIndividuo := {->}
          indice:=0

//-------------------------------------------------------------
//Indica Tentativa
private indicaTentativa (controle as Cromossomo) as Cromossomo
  var valorFitness = (max x|x in Indices(controle))
  // guarda indices dos candidatos
  var possiveisCandidatos as Set of Integer = {}
  var candidato as Cromossomo = {->}
```

```
    //monta o conjunto com indices dos possiveis candidatos
    step foreach indices in [0..tamanhoPopulacao]
       if populacao (indices).pontuacao = valorFitness then
           possiveisCandidatos := possiveisCandidatos union {indices}
    step
       // escolhe um candidato aleatoriamente
       choose alguem in possiveisCandidatos
           candidato := populacao (alguem).corpo

    step
        return candidato

//-----------------------------------------------------------------
Main()
  var tab = new Tabuleiro()
  var agJogador = new AgenteJogador(tab)
  step
     agJogador.inicio() //inicio(agJogador)
     WriteLine("Fim")
```

Apêndice B

O Mundo Wumpus

```
//-------------------------------------------------------------------
/*CavernaBase: Esse módulo provê informações de configuração
comuns ao ambiente e aos aventureiros do Mundo Wumpus.
Programador: Eliseu C. Miguel
Orientador: Vladimir Oliveira Di Iorio
Data :02/08/2004
*/

module CavernaBase

 initial states:


  type
     public Cavidade is (x: Int; y: Int);//Defini uma posição na caverna
     public Direcao is public enum { Norte, Leste, Sul, Oeste };

  static


  dynamic
     //retorna a posição de linha de uma cavidade
     public getLinha (c: Cavidade):Int := c.x;
     //retorna a posição de coluna de uma cavidade
     public getColuna(c: Cavidade):Int := c.y;
//-------------------------------------------------------------------

/* Machina: Módulo obrigatório que dispara as transições
```

```
Programador: Eliseu C. Miguel
Orientador: Vladimir Oliveira Di Iorio
Data :02/08/2004
*/
module Machina

import:

MundoWumpus(login);

init:

create MundoWumpus,

create x : agent of AventureiroSimples do
  login (x, "Pirata"),
        create y : agent of AventureiroSimples do
  login (x, "Cavaleira"),
end,

end

//-------------------------------------------------------------------
/*
MundoWumpus: Módulo de gerência do ambiente para o Mundo Wumpus
Programador: Eliseu C. Miguel
Orientador: Vladimir Oliveira Di Iorio
Data: 02/08/2004
*/

module MundoWumpus

  import CavernaBase(getLinha, getColuna, Direcao, Cavidade);

  initial state:

    type

      //informações sobre cada agente aventureiro
      AvDescriptor is (
        nome : String;
        direcao : Direcao;
```

```
        posicao : Cavidade;
      );

      //ocupantes das cavidades da caverna no jogo Wumpus
      Ocupante is enum
      {
         Wumpus,
         MauCheiro,
         Buraco,
         Brisa,
         Ouro,
         Brilho
      };

      // controle para geração da Caverna
      SetupCaverna is enum
      {
         Inicial,
         InsereElementos,
         InicioJogo
      }
      // define quem está na cavidade
      Caverna is Cavidade -> set of Ocupante;
      // usado para montar o mundo inicial
      TCavidades is set of Cavidade;

//- - - -
    static
        nBuracos     : Int := 3;  // define quantos buracos haverá na caverna
        limiteLinha  : Int := 10; // limite máximo de linha
        limiteColuna : Int := 10; // limite máximo de coluna

//- - - -

    dynamic
        // ambiente do jogo
        caverna : Caverna;
        // conjunto de cavidades para montar o mundo
        todasCavidades : TCavidades
        // conjunto de agentes para escolher a direção
        aventureirosJogam : set of Agent ={};
        // conjunto de agentes para mover na caverna
```

```
    aventureirosMovem : set of Agent ={};
    // informações do aventureiro
    avInfor : Agent -> AvDescriptor;
    // lista de elementos a compor a caverna
    vencedores : Set of Agent = {};
  listaOcupantes : list of Ocupante =
       [Wumpus, Ouro] + listaBuracos(nBuracos);


//- - - -

  derived
    //funções públicos. Podem ser utilizados pelos agentes aventureiros
    public minhaPosicao : Cavidade := avInfor(self).posicao;
    public minhaDirecao : Direcao  := avInfor(self).direcao;
    public meuNome      : String   := avInfor(self).nome;

    posicaoAgente(ag: Agent) : Cavidade := avInfor(ag).posicao;
    direcaoAgente(ag: Agent) : Direcao  := avInfor(ag).direcao;

    /*
          Essas funções não recebem a posição onde procura-se
          Brisa ou Brilho ou MauCheiro. Isso ocorre visto que
          o aventureiro, ao usar essa função, somente poderá
          saber o que existe onde ele está.
    */
    public temBrisaAqui : Bool   := Brisa     in caverna(minhaPosicao);
    public temBrilhoAqui: Bool   := Brilho    in caverna(minhaPosicao);
    public temMauCheiro : Bool   := MauCheiro in caverna(minhaPosicao);


    //funções não públicas.
    //Não podem ser utilizados pelos agentes aventureiros
    /*
          É necessário passar a cavidade onde procura-se o
       Wumpus ou um Buraco ou o Ouro pois o MundoWumpus é o único
       agente a usar esses testes e o aplica a todos agentes aventureiros.
    */
    temWumpus(cav: Cavidade):= Wumpus in caverna(cav);
    // passa a cavidade pois avalia para todos agentes
    temBuraco(cav: Cavidade):= Buraco in caverna(cav);
    // passa a cavidade pois avalia para todos agentes
```

```
    temOuro  (cav: Cavidade):= Ouro   in caverna(cav);


    listaBuracos(quantos: Int): list of Ocupante :=
        if quantos = 0 then []
        else cons(Buraco, listaBuracos(quantos-1));


    //sorteia uma cavidade para montar o mundo
    sorteiaCavidade :cavidade :=
       choose y in todasCavidades
          y
    end;


    //pega as características do ocupante
    getCaracteristica(elemento: Ocupante) : Ocupante :=
        if elemento = Wumpus then MauCheiro
        elseif elemento = Buraco then Brisa
           elseif elemento = Ouro then Brilho
    end;


//-  -  -  -
    action

       //inicia uma caverna vazia
       inicializaCaverna :=
           forall y  in {1..LiminteLinha}
               forall x  in {1..LimiteColuna}
                   caverna((y,x)):= {};
       end;

       //inicia conjunto com todas cavidades menos a (1,1)
       inicializaCavidades :=
           forall y in {1..LiminteLinha}
               forall x in {1..LimiteColuna}
                   if (x!=1) and (y!=1) then
                      todasCavidades ((y,x)):= true;
        end;

        //Coloca um elemento ocupante na caverna
```

```
insereOcupante(elemento: Ocupante):=
   let cavidade := sorteiaCavide(todasCavidades);
   in
       caverna(cavidade):= caverna(cavidade) + {elemento};
       todasCavidades(cavidade):= false;
       if (elemento != ouro) then
           montaVizinhos(getCaracteristica(elemento), cavidade);
       else
           caverna(cavidade):= caverna(cavidade) + {Brilho};

    end;
end;

//monta a vizinhaça para um novo elemento na caverna
montaVizinhos(vizinho:Ocupante, cavi : Cavidade):=
   let x  := getLinha(cavi);
       y  := getColuna(cavi);
   in
     if  y < limiteColuna then
         caverna((x ,y+1))  := caverna((x ,y+1)) + {vizinho}

     if  x < limiteLinha then
         caverna ((x+1 ,y)) := caverna ((x+1 ,y)) + {vizinho}

     if  y > 1 then
         caverna ((x ,y-1)) := caverna((x ,y-1)) + {vizinho}

     if  x > 1  then
         caverna ((x-1 ,y)) := caverna((x-1, y)) + {vizinho}
   end
end;


//inclui um agente aventureiro no jogo
public login (a: Agente, nome: String):=
   aventureirosJogam(a):= true;
   avInfor(a) := AvDescriptor(nome, Norte, Cavidade(1,1));
end;


//monta a caverna colocando todos os Ocupantes
loop posicionaElementos :=
```

```
      if listaOcupantes = [] then return
      else
         let (a:x) = listaOcupantes
         in
            insereOcupante(a);
            listaOcupantes := x;
         end
      end
   end;

//permite que o agente aventureiro decida sua direção
public novaDirecao(d:Direcao)  :=
   if aventureirosJogam(self) then
     avInfor(self).direcao := d;
     aventureirosJogam(self):= false;
     aventureirosMovem(self):= true;
end;


// ação para gerar a configuração inicial
//da caverna e iniciar o jogo
loop constroiCaverna(fase: SetupCaverna):=

   case fase of


      Inicial =>
           inicializaCaverna;
           inicializaCavidades;
           fase:= insereElementos;
      ;

      InsereElementos =>
           posicionaElementos
           fase:= inicioJogo;
      ;

      InicioJogo=>
           return;
      ;

   end; //case
```

```
end;//constroiCaverna


// ação para gerir o jogo
ciclo:=
   if aventureirosJogam = {}
       forall y in aventureirosMovem
          let cavidade_y = posicaoAgente(y)
          in
              aventureirosMovem(y):=false;

                 if  temOuro(cavidade_y) then
                       vencedores(y):=true;
                       stop;

                 elseif not(temWumpus(cavidade_y)) and
                           not(temBuraco(cavidade_y)) then

                       aventureirosJogam(y):= true;
                       caminha(y);

                 end //if

           end  //let

end;//ciclo


// ação para caminhar um aventureiro desde que este
//queira ir para uma cavidade válida para a caverna
caminha (av: Agent):=

   case av.minhaDirecao of

        Norte =>
           if getColuna(av.posicao) <  limiteColuna then
               av.posicao := Cavidade(getLinha(av.posicao),
                                      getColuna(av.posicao) + 1 )

        Sul =>
           if getColuna(av.posicao) >  1 then
```

```
                    av.posicao := Cavidade(getLinha(av.posicao),
                                           getColuna(av.posicao) - 1 )

            Leste =>
               if getLinha(av.posicao) <  limiteLinha then
                    av.posicao := Cavidade(getLinha(av.posicao) + 1,
                                           getColuna(av.posicao))

            Oeste =>
               if getLinha(av.posicao) >  1 then
                    av.posicao := Cavidade(getLinha(av.posicao) - 1,
                                           getColuna(av.posicao))


         end;//case

      end; // caminha


//-----------
  init

      constroiCaverna(Inicial)

//-----------

  transition

      ciclo

end

//-------------------------------------------------------------


/* AventureiroSimples: Os competidores implementarão
   agentes que utilizem técnicas de IA para decidir qual
   melhor caminho para caminhamento. Esse módulo exemplo
   cria um agente sem inteligência capaz de escolher o
   caminho com base  no sorteio da direção.
```

```
   Programador: Eliseu C. Miguel
   Orientador: Vladimir Oliveira Di Iorio
   Data :10/08/2004
*/

module AventureiroSimples

  import:

MundoWumpus(novadirecao);
        CavernaBase(Direcao);

  action

     // decide a direção por não determinismo
     decideDirecaoAleatoria :=
        choose y in Direcao
            novaDirecao(y)
     end;

  transition

      decideDirecaoAleatoria;

end
//-------------------------------------------------------------
```