

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA
COMPUTAÇÃO**

Fernando Schütz

**Programação Orientada a Comportamentos baseada
no Modelo de Atores**

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Raul Sidnei Wazlawick

Florianópolis, Julho de 2003.

Programação Orientada a Comportamentos baseada no Modelo de Atores

Fernando Schütz

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação – Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Fernando Álvaro Ostuni Gauthier, Dr.

Banca Examinadora

Raul Sidnei Wazlawick, Dr. (orientador)

Alvaro G. R. Lezana, Dr.

Rogério Cid Bastos, Dr.

Olinto José Varela Furtado, Dr.

“... Quando trabalhais, sois uma flauta através da qual o murmúrio das horas se transforma em melodia. ... E, apegando-vos ao trabalho, estarei na verdade amando a vida. E quem ama a vida através do trabalho, partilha do segredo mais íntimo da vida. ... Disseram-vos que a vida é escuridão; e no vosso cansaço, repetis o que os cansados vos disseram. E eu vos digo que a vida é realmente escuridão, exceto quando há um impulso. E todo impulso é cego, exceto quando há saber. E todo saber é vão, exceto quando há trabalho. E todo trabalho é vazio, exceto quando há amor. E quando trabalhais com amor, vós vos unis a vós próprios, e uns aos outros, e a Deus.

O trabalho é o amor feito visível.”

Gibran Khalil Gibran

Este trabalho é dedicado a Plínio e Erenita, além de Carla, Fabiane, Carlito, Pedro e Ana. Em especial à amorosa Fabiana, que não deixou a peteca cair, e também aos incondicionais amigos Cristiano e Orlando. Obrigado também aos professores Raul, Artur, Cláudio e Anibal.

Deus seja louvado.

Sumário

<i>Índice de Figuras</i>	<i>viii</i>
<i>Índice de Tabelas</i>	<i>x</i>
<i>Resumo</i>	<i>xi</i>
<i>Abstract</i>	<i>xii</i>
1 Introdução	1
1.1 Contexto do Trabalho	3
1.2 Objetivos	4
1.2.1 Objetivo Geral	5
1.2.2 Objetivos Específicos	5
1.2.3 Justificativa.....	5
1.2.4 Metodologia.....	6
1.2.5 Resultados Esperados	7
1.2.6 Limitações do Trabalho	8
1.2.7 Estrutura da Dissertação	9
2 Modelo de Atores	10
2.1 Definições de Atores	10
2.2 Comunicação entre atores	13
2.3 Características dos Sistemas baseados em Atores	15
2.3.1 Encapsulamento.....	15
2.3.2 Herança.....	15
2.3.3 Delegação	16
2.3.4 Concorrência.....	16

2.4	Implementação.....	17
2.4.1	Estruturas de Controle	17
2.4.2	Joint Continuations	19
2.5	Linguagens Baseadas em Atores	21
2.5.1	A linguagem <i>ACT</i>	22
2.5.2	A Linguagem SALSA (<i>Simple Actor Language, System and Architecture</i>)	24
2.5.3	A Proposta Acktalk.....	28
2.5.3.1	Transformação de objetos do Smalltalk-80 em atores	29
2.5.4	A Ferramenta Mundo dos Atores	30
3	<i>Programação de atores baseada em comportamentos</i>	34
3.1	Comportamentos	34
3.2	Comportamentos como Estereótipos de Classes.....	35
3.3	Classificação de Comportamentos	36
3.3.1	Comportamentos Latentes e Evidentes.....	36
3.3.2	Comportamentos que afetam outros objetos	36
3.3.3	Comportamentos Básicos e Compostos	37
3.3.4	Comportamentos Terminais	37
3.4	Atributos.....	37
3.5	Descrição dos Comportamentos.....	39
3.5.1	Comportamentos Básicos Evidentes	40
3.5.2	Comportamentos Básicos Latentes.....	41
3.5.3	Comportamentos básicos terminais.....	42
3.6	Composição de Comportamentos.....	42
3.6.1	Herança.....	43
3.6.2	Agregação (composição)	44
3.6.3	Especialização de parâmetros.....	45
3.6.4	Criação de novos comportamentos.....	45

3.7	Condições.....	46
4	<i>Definição da Linguagem</i>	49
4.1	Componentes do ambiente de programação.....	49
4.1.1	O Palco	49
4.1.2	Os Atores	50
4.2	Definição da Sintaxe	51
4.2.1	Atributos.....	52
4.2.2	Condições Compostas	53
4.2.3	Comportamentos Compostos.....	54
4.2.4	Comportamentos Condicionados.....	54
4.2.5	Definição de Atores.....	55
4.2.6	Definição de Comportamentos Reutilizáveis	55
4.2.7	O Programa Completo	56
4.3	Um Exemplo.....	57
5	<i>Jogos Analisados</i>	59
5.1	Megamania	59
5.1.1	Atores e Comportamentos do Jogo.....	61
5.2	River Raid	66
5.2.1	Atores e Comportamentos do Jogo.....	69
6	<i>Conclusões e sugestões de trabalhos futuros</i>	73
6.1	Sugestão para trabalhos futuros	75
7	<i>Referências Bibliográficas</i>	76

Índice de Figuras

Figura 1 – Composição de um Ator	11
Figura 2 – As três ações que um ator pode executar	12
Figura 3 – Exemplo de Diagrama de Eventos em Atores.....	14
Figura 4 – A computação recursiva de um fatorial	18
Figura 5 – Uma lista de números a serem multiplicados.....	20
Figura 6 – Diagrama de Eventos da árvore de produto	21
Figura 7 - Estrutura dos atores em SALSA	27
Figura 8 - O Mundo dos Atores: tela que representa o Palco , na ferramenta.....	31
Figura 9 - Exemplos de criação de novas ações no palco desenhaPalhaço.....	32
Figura 10 - <i>Screenshot</i> do jogo Megamania: na parte superior, os alienígenas da primeira fase; logo abaixo, o canhão que representa o jogador; e na base da tela os controles de pontuação, energia e vidas.....	60
Figura 11 - Três exemplos de <i>Screenshots</i> do megamania: fase 2, fase 3 e fase 4, respectivamente.	60
Figura 12 - Ilustração do movimento do ator alien F1	64
Figura 13 - Ilustração da movimentação do ator F2.....	64
Figura 14 - Ilustração da movimentação do ator alineF3	65
Figura 15 - Ilustração da movimentação do ator alienF6	66
Figura 16 - Ilustração do movimento do ator alienMovDinamica	66
Figura 17 - Tela inicial do jogo River Raid.....	67
Figura 18 - <i>Screenshot</i> demonstrando a ponte: passagem de fases	68

Figura 19 - Ilustração do comportamento movimentoInimigo.....	70
Figura 20 - Ilustração do movimento do ator aviaoAJato	71

Índice de Tabelas

Tabela 1 - Definição dos tipos primitivos da linguagem proposta.....	38
Tabela 2 - Descrição dos atributos primitivos de cada ator.....	39
Tabela 3 - Descrição dos comportamentos básicos evidentes encontrados.....	40
Tabela 4 - Relação dos comportamentos básicos latentes encontrados.....	41
Tabela 5 - Apresentação dos comportamentos básicos terminais encontrados.....	42
Tabela 6 - Relação das condições primitivas encontradas nos jogos propostos.....	47

Resumo

SCHÜTZ, Fernando; WAZLAWICK, Raul Sidnei. Programação Orientada a Comportamentos Como Uma Extensão do Modelo de Atores. Florianópolis, 2003. 80p. Dissertação (mestrado) – Universidade Federal de Santa Catarina.

A programação de computadores é um dos principais desafios para iniciantes em computação que, normalmente utilizam linguagens imperativas na fase de aprendizagem, às quais inserem uma forma não-usual, do ponto de vista natural, de descrever situações. O presente trabalho vem de encontro a esta dificuldade, pois propõe uma sintaxe mais antropomórfica, baseada em comportamentos de objetos no *Modelo de Atores*, elaborada a partir de análises em jogos do tipo *video game*, onde pode-se descrever os comportamentos dos objetos que compunham tais jogos, bem como interações entre os objetos e também as condições que regem as ações que cada ator desempenha no ambiente. O presente trabalho baseia-se na idéias evidenciadas no *Modelo de Atores*, utilizado em disciplinas de iniciação à programação de computadores, e faz um apanhado geral das principais linguagens de programação concorrentes baseadas no modelo de atores. Há também a utilização da programação orientada a objetos da abordagem UML (Unified Modeling Process).

Palavras-chave: Modelo de Atores, Programação Orientada a Comportamentos, Jogos.

Abstract

The computer programming is one of the major challenges to beginners in the computer science that usually use imperative languages to start the learning, which introduce a non usual form, from the natural point of view, of describing situations. This research meets this difficulty, because it proposes a more anthropomorphic syntax, based on object behaviors and the Actor Model. Elaborated from games like *video games* where it is possible to describe the behavior of objects which belonged to such games, as well as interactions among the objects and also the conditions that rule the actions that each actor performs in the environment. This research is based on ideas displayed in the Actor Model, used in subjects of computer programming beginning and it englobes the main languages of concurrent programming based on the Actor Model. There is also the use of guided programming to objects and the Unified Modeling Process approach.

Key words: Actor Model, Guided programming to behaviors, Games.

1 Introdução

Acredita-se que o desenvolvimento de métodos para a comunicação entre os seres tenha sido um dos fatores predominantes da evolução terrestre. Segundo Sebesta (2000), a profundidade de capacidade intelectual do ser humano é influenciada pelo poder expressivo da linguagem utilizada para a comunicação dos pensamentos de cada um, entre os seres inteligentes que convivem no meio. Assim, pode-se dizer que os que possuem uma compreensão limitada da linguagem natural são limitados na complexidade de expressar seus pensamentos, especialmente em termos de profundidade de abstração: é difícil para as pessoas conceberem estruturas que não podem descrever, verbalmente ou por escrito. De acordo com Sebesta (2000), programadores inseridos no processo de desenvolvimento de *software* vêm-se, muitas vezes, reféns desta dificuldade. A linguagem de programação na qual desenvolve-se um software impõe limites quanto aos tipos de estruturas de controle, de estruturas de dados e de abstrações que se pode usar, limitando também as formas possíveis de conceber um algoritmo.

A programação de computadores é uma disciplina jovem. As metodologias de projeto, as ferramentas de desenvolvimento de software e as linguagens de programação ainda estão em estágio de contínua evolução (Sebesta, 2000). Apresentando-se qualquer paradigma de programação a usuários comuns, pessoas que não estão envolvidas no processo de desenvolvimento de software através de suas teorias da ciência da computação, vê-se que existe certa dificuldade para muitos em ultrapassar a grande barreira inicial (Mariani, 1998). Esta barreira é constituída entre outros aspectos, pela aprendizagem de uma linguagem de programação formal e de conceitos computacionais como variáveis, procedimentos, controle de fluxo, classes e objetos, gerando assim uma sobrecarga cognitiva em tais usuários (Pianesso, 2002).

Averiguando-se os conceitos abordados nos paradigmas computacionais imperativo e orientado a objetos, vê-se que a sintaxe e a semântica dos mesmos trabalham de acordo com a máquina projetada por John Von Neumann, na década de 40, e que se aperfeiçoa até

hoje nos aspectos imutáveis dos computadores modernos: armazenamento do tipo escreve/recupera, conteúdo de memória endereçado conforme sua posição e não pelo conteúdo, e execução de instruções de forma seqüencial (Tanenbaum, 1997). Assim, tem-se nas linguagens de programação atuais, por mais alto nível que se tente chegar, uma relação que inicia da máquina e é imposta ao ser humano.

Os paradigmas de programação poderiam, porém, ser encarados a partir do conhecimento intrínseco do ser humano, através de estruturas que realmente possam ser por ele compreendidas, presentes no seu dia a dia, e na sua linguagem.

Segundo Sebesta (2000), programadores que entendem o conceito de abstração de dados terão mais facilidade para aprender como construir tipos de dados abstratos em Java do que aqueles que não estão absolutamente familiarizados com tal exigência. O mesmo fenômeno ocorre nas linguagens naturais. Quanto mais se conhece a gramática de sua língua nativa, mais fácil será aprender uma segunda língua.

A concepção de linguagens baseadas num paradigma inverso, onde o ser humano concebesse os conceitos que fundamentassem uma situação, e que tais conceitos pudessem ser descritos através de padrões em sua própria linguagem, para então ser traduzidos a uma máquina poderia tornar a programação um ato mais espontâneo.

Segundo (Lieberman, 1987), a programação de computadores tomou um novo rumo, direcionando-se à programação distribuída, utilizando a Internet, e trabalhando com o conceito de modelos de processamento inteligente como uma sociedade de indivíduos cooperativos.. Sendo assim, o conhecimento deve ser distribuído entre os membros da sociedade, onde cada membro possui apenas o conhecimento necessário ao processamento de sua função; deve haver comunicação entre os membros de tal sociedade, com a finalidade de, por exemplo, pedir ajuda ou informar sobre seus progressos; a possibilidade de se processar diferentes tarefas em paralelo deve existir, concluindo informações mais rapidamente; e diferentes subgrupos da sociedade precisam estar aptos a compartilhar seus conhecimentos e recursos..

Este trabalho procura abordar conceitos envolvidos com a detecção, através de avaliações passíveis de serem feitas por seres humanos, de comportamentos em jogos do tipo *Video Game*, para verificar a potencialidade de abordar conceitos e comportamentos assim detectados como elementos de uma linguagem de programação. Como base para tais questões, o trabalho de (Pianesso, 2002) será o norteador inicial desta pesquisa, pois um dos pontos abordados foi o da apresentação de um formato de descrição para o conhecimento adquirido através da experimentação de projetos de software já constituídos que não se baseie apenas na descrição seqüencial de ações, mas realmente na identificação e atribuição de comportamentos aos objetos também identificados, sendo assim uma proposta de grande importância referencial.

1.1 Contexto do Trabalho

O trabalho, intitulado *Identificação e Classificação de Comportamentos de Objetos Dinâmicos* (Pianesso, 2002), será abordado como a principal fonte deste trabalho, tido como a base para a verificação e classificação inicial de comportamentos em jogos produzidos através da ferramenta Mundo dos Atores (Mariani, 1998).

Inicialmente, a concepção do projeto *Museu Virtual* (Wazlawick, Kirner et al, 2001), que é um projeto de pesquisa que propõe fornecer uma ferramenta de autoria distribuída multi-usuário, na qual estudantes, em locais diferentes possam simultaneamente construir objetos em ambientes de realidade virtual definindo comportamentos para estes objetos, levou à construção do trabalho de análise e classificação de comportamentos, pois tal pesquisa auxiliaria na construção de uma linguagem que pudesse melhorar a compreensão do visitante quanto à programação dos objetos (Pianesso, 2002).

A ferramenta *Mundo dos Atores*, proposta por (Mariani, 1998) foi utilizada como artefato de estudo, pois se apresenta como uma ferramenta onde os usuários podem praticar sua criatividade e serve para o projeto ou manipulação de elementos de *software*, como a inserção de objetos dinâmicos reativos e pró-ativos. Tal ferramenta mostrou-se satisfatória na concepção e prototipação de *software* educacional produzido por professores do ensino

fundamental e médio sem conhecimento de computação (Pianesso, 2002). Porém, várias dificuldades foram encontradas na aplicação desta ferramenta. Entre elas a necessidade do uso de uma linguagem de programação, que embora muito simples e flexível, produz uma sobrecarga cognitiva não desejável no processo (Wazlawick et al, 2001).

O trabalho desenvolvido por (Pianesso, 2002) baseou-se na criação de uma biblioteca de padrões de comportamentos predefinidos de objetos, para que o *Mundo dos Atores* possa ser adaptado para o trabalho com desenvolvedores de aplicações sem conhecimento específico em programação de computadores (Pianesso, 2002). Para a determinação dos comportamentos predefinidos, foram utilizados alguns projetos de jogos desenvolvidos no decorrer de várias disciplinas onde a ferramenta *Mundo dos Atores* (Mariani, 1998) foi utilizada.

Mesmo tendo facilitado a programação através da biblioteca de padrões construída, ainda há a necessidade de se conhecer alguns conceitos relativos à linguagem de programação *Smalltalk* (base do projeto Mundo dos Atores) e também assume-se que a programação é uma *seqüência* de ações, estando o projeto ainda baseado nas idéias de seqüencialidade e imperatividade.

Tomando-se como base uma proposta para a criação de uma linguagem que contemple a utilização de padrões de comportamentos, facilitando a criação de aplicações de maneira que o usuário não necessite utilizar comandos imperativos e a forma de programar existentes atualmente na ferramenta *Mundo dos Atores* (Mariani, 1998).

1.2 Objetivos

Tendo em vista o contexto apresentado para o trabalho, e vindo de encontro às características de modelagem e programação descritas anteriormente, os objetivos deste trabalho são apresentados a seguir.

1.2.1 Objetivo Geral

Com base na averiguação de jogos do tipo Vídeo Game, o objetivo geral deste trabalho consiste na apresentação de um modelo de programação, baseado na atribuição de comportamentos a objetos que existam em um ambiente programável, com tempo discreto, que se baseie no modelo de execução síncrono.

1.2.2 Objetivos Específicos

Para que o objetivo geral do trabalho seja atingido, algumas etapas devem ser cumpridas, e apresentam-se a seguir como objetivos específicos:

- a) Definir um conjunto de comportamentos básicos que possam ser usados para construir os comportamentos observados em jogos clássicos.
- b) Discutir a questão do nível ideal de abstração dos comportamentos básicos;.
- c) Propor um aprimoramento do Mundo dos Atores (Marianni, 1998) substituindo as ações elementares (baseadas nos comandos Logo) por comportamentos observados de nível mais abstrato;
- d) Desenvolver a sintaxe para o modelo proposto;

1.2.3 Justificativa

Partindo do fato que a programação que segue o Modelo de Atores é uma área pouco explorada, e que apresenta um ótimo potencial futuro devido à capacidade de se adequar a diferentes áreas de atuação, a elaboração de um bom referencial teórico e o aprofundamento em questões relevantes ao modelo é de crucial importância para a revitalização dos conceitos pertinentes à área, como por exemplo a discussão do nível de abstração dos comportamentos básicos, pois é importante que se evite a definição de comportamentos

demasiadamente específicos e, portanto, com pouca reusabilidade, ou demasiadamente genéricos, recaindo nas ações básicas de linguagens de programação.

Uma das áreas que pode ser citada como promissora quanto ao uso do Modelo de Atores é sua utilização como uma ferramenta de apoio ao ensino de lógica de programação, ensino este fundamentado geralmente em algoritmos estudados no papel, deixando somente a aplicação de tais algoritmos para outras disciplinas (onde também se introduz os paradigmas clássicos de programação como os paradigmas lógico, funcional, imperativo e até mesmo a programação orientada a objetos). Como um exemplo de aplicação de tais conceitos, tem-se o trabalho de (Mariani, 1998), onde apresenta a ferramenta *Mundo dos Atores*, que insere o aluno em um contexto de aprendizagem que aborda tanto as características da lógica de programação quanto da orientação a objetos (utilizando recursos computacionais). Segundo (Wazlawick & Mariani, 2001), “a ferramenta tem se mostrado adequada no processo de ensino de programação para alunos do curso de ciência da computação. Abordada na concepção e prototipação de *software* educacional por professores do ensino fundamental e médio sem conhecimento em computação, tal ferramenta também tem apresentado uma utilização satisfatória”.

1.2.4 Metodologia

Inicialmente, uma revisão bibliográfica foi elaborada a partir de artigos, livros e outros materiais disponíveis, que tratam dos assuntos relacionados a este trabalho, fundamentalmente o Modelo de Atores, as ferramentas e linguagens de programação desenvolvidas com as características do Modelo de Atores, e a própria Orientação a Objetos. A maioria dos elementos que compõem a revisão bibliográfica cita modelos de eventos assíncronos, sendo que existem poucas publicações citando linguagens baseadas em atores que trabalhem com a distribuição de eventos síncronos.

Durante a fase de pesquisa, vários comparativos entre as diversas formas de manipulação do Modelo de Atores foram realizados, com a finalidade de abordar os

aspectos característicos de cada especificação, qualificando e diferenciando todos os métodos estudados.

A interação com os jogos foi efetuada observando-se as ações exibidas pelos diferentes objetos de cada *game*. Para tal, foi utilizado um emulador de vídeo games ATARI 2600 multi-plataforma, chamado Stella (<http://stella.atari.org>).

Através da observação da execução de cada jogo no emulador supracitado, numa primeira etapa é concluída uma descrição antropomórfica dos objetos observados seguida por um relato de forma técnica, fomentando assim a comparação entre os métodos. Assim foi desenvolvida uma classificação de objetos, comportamentos, atributos e condições, tornando-se possível definir parâmetros e as principais características de tais jogos.

A linguagem UML (Unified Modeling Language) foi utilizada no processo de catalogação dos comportamentos dos objetos existentes em cada jogo observado. Tal método foi usado para facilitar a descrição dos componentes do jogo de forma hierárquica, indicando assim as relações existentes entre os objetos presentes nos jogos e servindo como base para a linguagem de definição de comportamentos.

1.2.5 Resultados Esperados

Espera-se verificar a possibilidade da criação de um paradigma de programação não imperativo, baseado na composição de comportamentos, que auxilie na definição de atores em ambientes colaborativos e competitivos.

Como inovação em relação aos paradigmas existentes que iniciaram a partir da definição de máquina para só depois criar um modelo de linguagem, espera-se inverter, pelo menos parcialmente o processo, partindo de estruturas concebidas pelos seres humanos (os comportamentos observados e descritos) e sua posterior caracterização em termos de máquina.

Este *design* antropomórfico utilizado para a definição da sintaxe é proposto para que seja possível uma diminuição na sobrecarga cognitiva dos programadores leigos.

1.2.6 Limitações do Trabalho

A primeira limitação do trabalho é apresentada quanto ao ambiente em que o jogo se apresenta inserido. Pretende-se aqui utilizar apenas jogos baseados na proposta de *Jogos de Tabuleiro*, onde apenas duas dimensões dos objetos são apresentadas ao usuário. Quaisquer características tridimensionais estão ignoradas no processo de definição dos comportamentos tanto na forma descritiva quanto sintática.

Os comportamentos de determinados atores que representam os usuários (jogadores) de tais jogos não serão discutidos, ou seja, apenas os eventos que sejam disparados por objetos controlados pela máquina serão definidos, excluindo-se assim quaisquer interações de controle que possam vir do mundo exterior. O estudo de Interferências entre Comportamentos (Pianesso, 2002) também não será alvo de estudo neste trabalho, pois tem-se que não é parte da definição da linguagem uma abordagem que normalize comportamentos definidos como anuladores, incompatíveis ou interferentes entre si.

Os jogos testados são *River Raid* e *Megamania* (ver descrição completa nas seções que seguem), e o trabalho pretende apenas discutir e propor a sintaxe para a nova linguagem proposta, excluindo-se qualquer tipo de discussão quanto à programação e construção de um compilador, sendo esta uma das sugestões de trabalhos futuros.

Além disso, para a proposta poder atingir plenamente seus objetivos de diminuição da sobrecarga cognitiva, seria possivelmente necessária a criação de uma interface para programação visual, na qual as composições de comportamentos pudessem ser feitas apenas pelo arrastar e soltar de peças em uma planilha. O trabalho atual limita-se a apresentar uma sintaxe formal para esta linguagem, merecendo o estudo das interfaces visuais uma boa dose de pesquisa futura. A descrição tanto da sintaxe quanto dos códigos dos exemplos limita-se aos comportamentos e atores que compõem o ambiente a ser apresentado, ignorando as questões relativas à inicialização e à criação do ambiente, sendo que tais critérios devem ser atribuídos ao palco, durante a implementação da linguagem.

1.2.7 Estrutura da Dissertação

O próximo capítulo (capítulo 2) aborda conceitos do modelo de atores, quanto à sua concepção e principais características, até a exemplificação e descrição de linguagens de atores já existentes. A discussão sobre as mesmas não é concluída, sendo que uma tabela comparativa mostra a ordem cronológica de desenvolvimento de linguagens baseadas no modelo de atores.

Descreve-se, no capítulo 3, a idéia de programação por comportamentos, os componentes da linguagem proposta e a discussão quanto aos itens básicos identificados nos modelos pesquisados. Assim, são definidos os conceitos de atores, ambiente e objetos diversos.

No capítulo 4 é apresentada a descrição dos itens que compõem a sintaxe da linguagem proposta, apresentando alguns conceitos quanto à caracterização de uma linguagem de programação, bem como a implementação de exemplos que demonstrem a utilização da linguagem proposta.

O capítulo 5 apresenta o coração do trabalho, definindo a sintaxe e alguns elementos da semântica para a programação dos jogos analisados, bem como a descrição e exemplificação, através de *screenshots*, dos jogos avaliados.

Têm-se, no capítulo 6, as conclusões e considerações finais do trabalho, e sugestões para a continuidade do mesmo.

2 Modelo de Atores

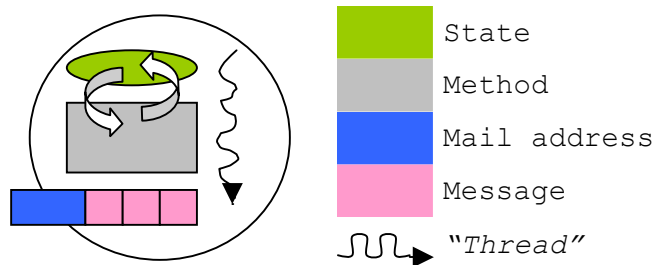
O Modelo de Atores, que teve seu desenvolvimento iniciado por Carl Hewitt (Mavadatt, 2002), é utilizado em algumas linguagens de programação como um método de programação concorrente. Inicialmente, o modelo proposto por Hewitt era uma comunidade de agentes, desenvolvido na linguagem *smalltalk*, baseada na programação orientada a objetos onde, cada objeto era considerado como uma entidade ativa, recebendo, enviando e reagindo a mensagens. Tais objetos, bem como as mensagens de interação, foram chamados de *atores* (Guy, 2003).

2.1 Definições de Atores

Segundo (Mavadatt, 2002), *atores* são objetos concorrentes e independentes que interagem pelo envio assíncrono de mensagens. Para (Guy, 2003), um ator pode ter arbitrariamente muitos “*conhecidos*”, ou seja, ele pode “*ter conhecimento*” sobre outros atores e enviar mensagens a estes. (Lieberman, 1987) coloca que atores acabam com a distinção convencional entre dados e procedimentos, criando um processamento concorrente através da alocação dinâmica de recursos em uma máquina paralela.

Na composição de tal objeto, tem-se que um ator encapsula estados, um conjunto de métodos e uma *thread* ativa (Figura 1). Cada ator tem seu único endereço de correio, servindo como um alvo para o recebimento de mensagens, que é associado com *buffers* de comunicação ilimitados, formando assim uma fila para recebimento de mensagens. (Varela, 2001).

Figura 1 – Composição de um Ator



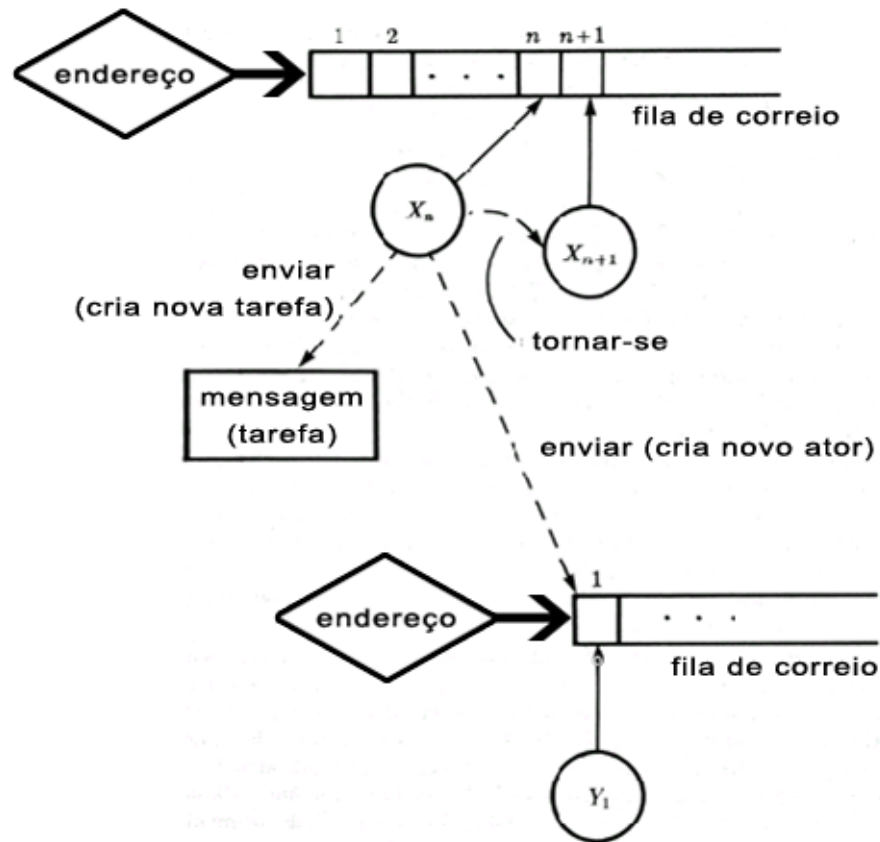
Fonte: Mavadatt, 2002

De acordo com Mavadatt (2002), o funcionamento do objeto ator é condicionado a mensagens. Por exemplo, considerando-se um ator $TESTE_n$: ao receber uma mensagem qualquer, o ator $TESTE_n$ responderá a essa mensagem pela execução de uma das três ações distintas, permitidas ao ator, que podem ser descritas como:

- a) Enviar: o operador enviar faz com que uma mensagem seja passada e colocada em uma fila de correio de outro ator. Como a comunicação é ponto a ponto, o endereço de correio do receptor precisa ser especificado.
- b) Tornar-se: a partir da operação tornar-se, o ator pode mudar seu comportamento pela criação de outro ator $TESTE_{n+1}$, com a finalidade de processar a próxima tarefa na fila.
- c) Criar: através da operação criar um ator consegue criar um novo ator com outro comportamento, porém agora especificado, com um endereço de correio único, que pode processar suas próprias mensagens.

A Figura 2 demonstra o processo que é executado no ator para desempenhar o papel das três ações possíveis, onde *mail queue* é a fila de correio representada por um vetor, um dos arcos entre os atores, *become*, é a operação tornar-se e também há a demonstração do arco *create*, que significa a criação de um novo ator. O arco *send* representa o envio de uma mensagem.

Figura 2 – As três ações que um ator pode executar



Fonte: Adaptado de Mavadatt, 2002.

Um sistema baseado em atores é composto por um grupo de atores e um conjunto de tarefas a serem executadas. Nestes sistemas, existem dois tipos de atores: um deles é conhecido como **repcionista** (*receptionist*), um ator que recebe comunicações de fora do sistema; o outro tipo é o **ator externo** (*external actor*), que não está no sistema, mas seu endereço é conhecido a um ou mais atores no sistema, permitindo assim o envio de comunicações entre eles (Agha, 1986).

A concorrência em tais sistemas é representada pelo processamento paralelo de mensagens. Segundo (Mavadatt, 2002), com o passar do tempo no sistema, novos atores

devem ser criados bem com novas tarefas devem evoluir. Tarefas que estejam completas e os atores que eram responsáveis por estas devem ser removidos do sistema.

2.2 Comunicação entre atores

Como apresentado anteriormente, a troca de mensagens é o principal meio de comunicação entre os atores em um sistema, e é por meio destas que todas as execuções concorrentes podem acontecer (Varela, 2001).

O mecanismo de passagem de mensagens é assíncrono e não-bloqueante, o que significa que um ator pode enviar uma mensagem a outro ator e continuar executando as ações a ele especificadas sem esperar que a mensagem seja recebida por outro ator. O ilimitado buffer de comunicação garante a entrega de todas as mensagens de entrada, sendo que tais mensagens chegam em uma ordem não determinística (Mavadatt, 2002).

Segundo Agha (1986), uma mensagem é representada como uma tupla contendo três elementos que são:

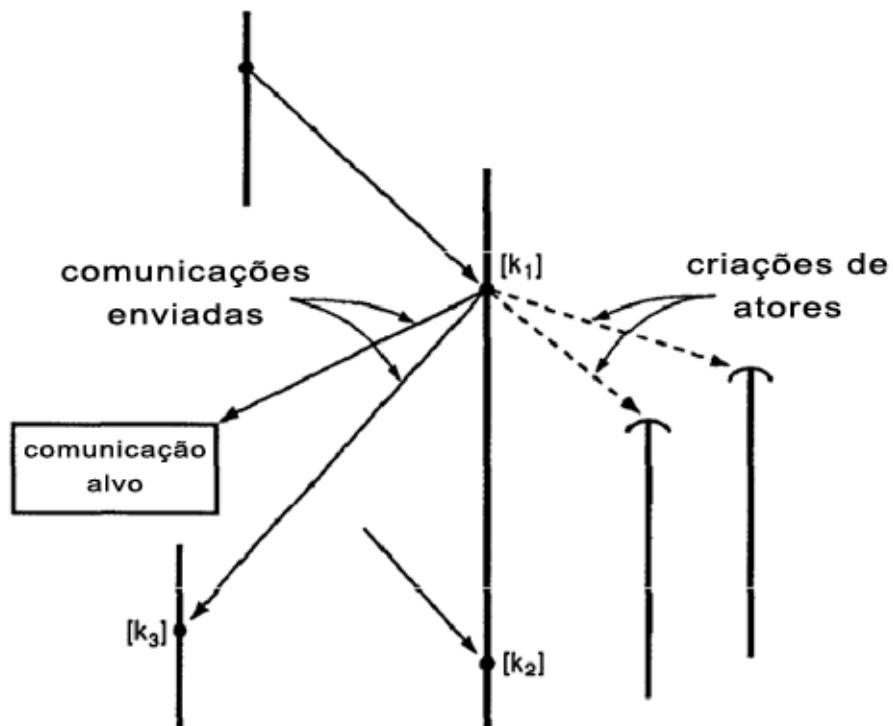
- a) TAG: que distingue cada mensagem (tarefa) no sistema;
- b) ALVO: que é o endereço de correio do ator ao qual a mensagem deve ser entregue;
- c) COMUNICAÇÃO: que contém a informação que se torna disponível ao ator no alvo. Esta informação poderá apresentar-se em qualquer formato de representação de dados.

Antes que um ator possa mandar a outro ator uma comunicação, ele precisa saber seu endereço de correio. Segundo (Mavadatt, 2002), os endereços de correio de todos os atores podem ser conhecidos (encontrados) por outros atores através da configuração inicial do sistema, de uma mensagem enviada de um ator para outro ou por meio da criação de um novo ator com um novo endereço de correio a este associado.

Segundo Mavadat (2002), a execução dos processos em um ator acontece através de eventos. Um evento pode causar o envio de uma mensagem de um ator para outro ou representar o processamento das mensagens em um ator. O processamento de eventos

ocorre de forma não determinista, pois duas mensagens enviadas a dois atores podem chegar em uma ordem que é diferente do envio; e uma comunicação enviada pelo mesmo ator pode chegar em uma ordem diferente da ordem que foi enviada. Comunicações entre atores podem ser representadas por um diagrama de eventos, ilustrado na Figura 3.

Figura 3 – Exemplo de Diagrama de Eventos em Atores



Fonte: Mavadatt, 2002

Na Figura 3, um ator é identificado pela linha vertical isolada, no topo da figura. Os pontos k_1 e k_2 representam a ordem de chegada dos eventos em um ator, onde o evento k_1 chega antes do evento k_2 . Após a ocorrência do evento k_1 , o ator deve criar um ou mais novos atores, o que é representado pelas linhas tracejadas. Como apresentado na figura, outras comunicações podem ser enviadas a outros atores (k_3 , por exemplo) (Mavadatt, 2002).

2.3 Características dos Sistemas baseados em Atores

Como o modelo de atores derivou do modelo orientado a objetos (*SIMULA* e *Smalltalk*), atores compartilham as principais características de objetos, porém derivam suas próprias características.

2.3.1 Encapsulamento

A característica do *encapsulamento* esconde todos os detalhes da implementação de um objeto do tipo ator, permanecendo apenas as interfaces visíveis ao meio externo (por exemplo, o conjunto de mensagens às quais o ator pode responder). Com o *encapsulamento* de informações em componentes autônomos, atores são capazes de incorporar diferentes pontos de vista, sem comprometer a integridade do sistema (Hewitt, 1995). Assim, através do encapsulamento as linguagens mantêm privacidade e segurança com respeito às estruturas internas dos atores.

2.3.2 Herança

De acordo com o trabalho de Mavadatt (2002), o mecanismo de *herança* permite a uma classe herdar comportamentos comuns de suas classes pai. A noção de classe não é totalmente integrada ao modelo de atores. Por exemplo, diferente de linguagens como *Smalltalk*, onde a herança impõe, a cada objeto pertencente a uma classe, restrições à mutabilidade do comportamento de um objeto filho. As linguagens baseadas em atores possuem características diferentes na implementação de suas classes, sendo que cada ator é uma entidade independente capaz de mudar seus comportamentos sem estar amarrado a nenhuma classe pai. Segundo Agha (1986), a herança no modelo de ator apenas provê a organização conceitual do sistema que é dinamicamente configurável.

2.3.3 Delegação

Delegação é descrita como outro método de compartilhamento de informações. Através do mecanismo de passagem de mensagens, sub-computações podem ser passadas de um ator para outro, o qual continua o processamento. “Delegação promove a modularidade do código por evitar a necessidade de duplicar o mesmo código no corpo de diferentes atores” (Mavadatt, 2002, pg. 6).

2.3.4 Concorrência

Segundo Agha (1996), desde a concepção do modelo de atores, Hewitt colocou os objetivos de desenvolvimento na exploração de concorrência em alta escala, afirmando que “atores são agentes que devem funcionar em paralelo”. A idéia de paralelismo reflete uma programação difícil, mas o desenvolvimento de uma sintaxe que libere o programador da preocupação com os detalhes de uma execução concorrente foi concluída através da remoção das dependências de dados desnecessárias nos comandos de atribuição.

Vários processadores podem ser utilizados na execução paralela de processos. Segundo Mavadatt (2002), o modelo de atores se baseia no comportamento de substituição (*replacement*) ao invés de atribuição, fazendo com que atores armazenem informações histórico-sensitivas (o que não pode ser feito nas linguagens ditas funcionais), e executem ações concorrentes quando não há dependência de dados. Assim, atores se apresentam apropriados para representar a evolução continuada em sistemas de tempo real, por exemplo, já que atores e as próprias ações de um único ator podem executar concorrentemente.

De acordo com Agha (1986), a concorrência é desenvolvida em um sistema de atores através do envio de várias mensagens em resposta a uma única mensagem e, o processamento é distribuído através da criação de atores consumidores (*customers*), que representam, de forma concorrente, as continuações e funções do ator que as criou. No mesmo trabalho, (Agha, 1986, p. 60) coloca um exemplo desta definição ao citar que “um fatorial recursivo, que é implementado em termos de um ator, o qual, quando recebe uma

requisição para avaliar o fatorial de n , cria um consumidor para esperar por uma mensagem dando o fatorial de $n-1$. O ator fatorial então envia a si mesmo uma requisição para avaliar o fatorial de $n-1$ junto com o endereço do consumidor. Assim, uma pilha é estruturada como uma corrente de consumidores, a fim de reduzir o gargalo seqüencial na computação”.

2.4 Implementação

Nesta etapa do trabalho, pretende-se mostrar um pouco das funcionalidades e características apresentadas na descrição do Modelo de Atores, demonstrando conceitos importantes como delegação e concorrência, e também alguns exemplos de processos tradicionais, utilizando-se as três ações primitivas imbuídas aos atores, criar, tornar-se e enviar.

2.4.1 Estruturas de Controle

As linguagens baseadas no modelo de atores não possuem, em sua definição dos comportamentos de atores, muitas estruturas de controle lógicas que são freqüentemente utilizadas na programação imperativa. Recursão, iteração, ou quaisquer outros operadores de repetição podem ser citados como exemplo. Porém, segundo Mavadatt (2002), tarefas que exijam controle concorrente de forma arbitrária necessitam de tais estruturas. As estruturas de controle representam um padrão particular de passagem de mensagens.

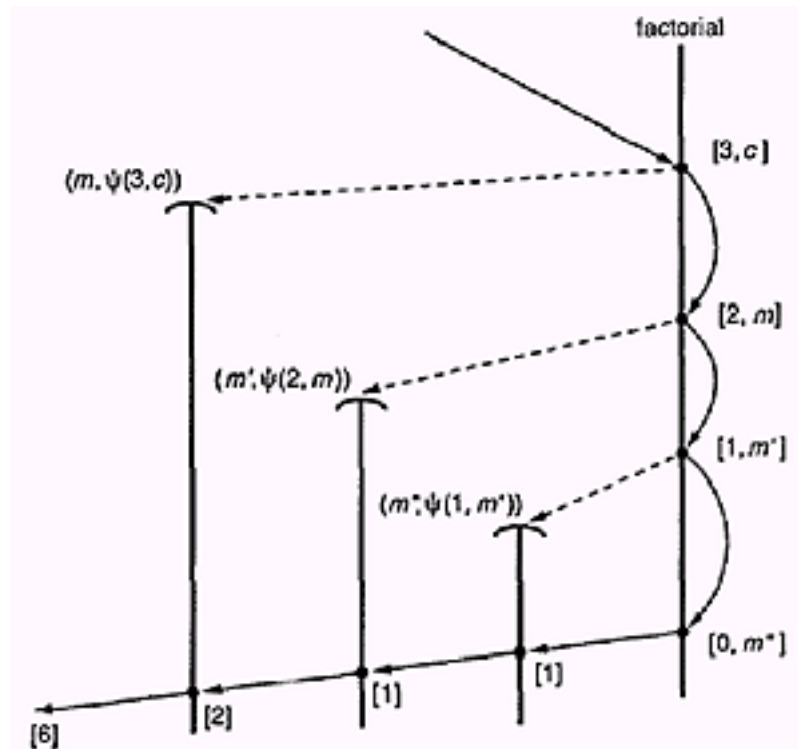
Para exemplificar o uso de recursividade em uma linguagem baseada no Modelo de Atores, abaixo tem-se o processamento da função fatorial recursivamente. Nele, primeiramente cria-se o “ator fatorial”, que aguardará pela mensagem apropriada para iniciar o processamento. Assume-se que o ator fatorial recebe uma mensagem incluindo um inteiro N diferente de zero (0) e um endereço de e-mail C para o qual o valor do fatorial deve ser enviado. Em resposta a essa mensagem, o ator fatorial:

- a) Cria um ator com o comportamento definido por $\psi(N, C)$ que multiplica N pelo inteiro recebido, enviando, ao final, o resultado ao endereço de correio C . Ao novo ator será atribuído um novo endereço de e-mail M .

- b) Envia a si mesmo uma requisição do tipo $[N-1, M]$ para calcular o fatorial de $N-1$, e para enviar o valor ao ator M que ele criou.

Em seu trabalho, Mavadatt (2002) apresenta um teste simples, efetuando uma simples requisição $[3, C]$ mostrada na Figura 4, onde “o ator fatorial cria novos atores M , M' , M'' seqüencialmente. Quando recebe uma requisição $[0, M'']$ ele apenas envia o valor 1 para o ator M'' . Depois de M'' receber um valor 1, ele multiplica 1 por 1 e envia o resultado 1 ao ator M' , o qual está baseado no seu comportamento $\psi(1, M')$. Similarmente, quando o ator M recebe o valor 2 (resultado da multiplicação) do ator M' , M multiplica 2 por 3 e envia o valor 6 ao ator C . Ao final, o valor de $3!$ estará contido no ator C ”.

Figura 4 – A computação recursiva de um fatorial



Fonte: Mavadatt, 2002.

Portanto, segundo Mavadatt (2002), uma linguagem de atores pode processar muitas requisições através da distribuição de tarefas computacionais aos atores, criados sobre os processadores distribuídos, formulando assim um processo mais rápido ao das linguagens

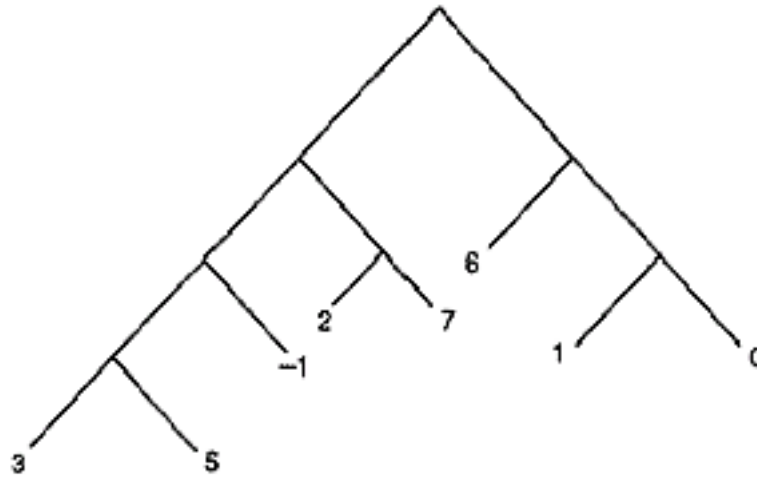
de programação imperativas, onde uma pilha é utilizada para o processamento recursivo, sem mecanismo de distribuição de tarefas computacionais e sem processamento concorrente entre duas ou mais requisições.

Segundo Agha (1986), para problemas que necessitem de pouco poder computacional e que sejam facilmente descritos, a solução recursiva apresentada se torna satisfatória, atingindo um nível aceitável de complexidade; porém quando um algoritmo propor um grande esforço computacional, outras técnicas de implementação devem ser usadas, como segue.

2.4.2 Joint Continuations

Para Varella (2001), a passagem de mensagens entre os atores é tida como seu processo de comunicação sendo implementadas como invocações de métodos em JAVA. Como o envio de mensagens é feito de forma assíncrona, um método para passar o retorno do processamento de tais mensagens é necessário para o funcionamento deste método. Assim, os termos dividir e conquistar podem ser naturalmente aplicados, usando uma função que avalie seus argumentos de forma concorrente. Como exemplo, abaixo apresenta-se um problema para a determinação do produto de uma lista de números, sendo que tal lista está representada como uma árvore (Figura 5).

Figura 5 – Uma lista de números a serem multiplicados.

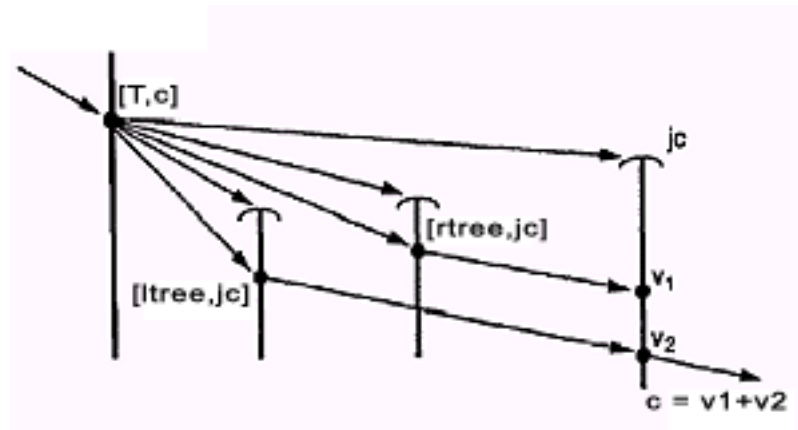


Fonte: Mavadatt, 2002

Como uma seqüência deve ser necessária, devido à natureza do problema, Mavadatt, (2002) coloca que a implementação das funções necessárias para a resolução do problema apresentado como exemplo requer a especificação de uma “continuação comum” (*joint continuation*), para sincronizar a avaliação dos diferentes argumentos que são processados concorrentemente. Os atores recebem mensagens que iniciam a computação de cada elemento; em resposta a uma comunicação com árvores não vazias T e um endereço de correio C , o ator responsável pelo produto na árvore, segundo Mavadatt (2002), efetuará os seguintes passos:

- c) Cria um novo ator, nominado jc , para a continuação comum. Seu comportamento é multiplicar dois inteiros recebidos, e enviar os resultados ao endereço de correio C .
- d) Cria um novo ator produto-árvore para avaliar o produto das T 's sub-árvores à esquerda e enviar o resultado ao ator jc .
- e) Cria um novo ator produto-árvore para avaliar o produto das T 's sub-árvores à direita e enviar o resultado ao ator jc .

Figura 6 – Diagrama de Eventos da árvore de produto



Fonte: Mavadatt, 2002

Conseqüentemente, a linguagem de atores pode delegar as tarefas computacionais a um número de atores, que podem processar suas tarefas delegadas concorrentemente. Em seu trabalho, Agha (1986) coloca que uma concorrência de granularidade fina é apresentada no modelo de atores, o que denota a preocupação com a otimização da linguagem; cita que problemas baseados em Inteligência Artificial são considerados aplicações principais às linguagens.

2.5 Linguagens Baseadas em Atores

Para ilustrar o histórico e também o estado da arte da utilização do modelo de atores mundialmente, esta seção oferece algumas considerações quanto a linguagens produzidas para trabalhar com o modelo de atores. No decorrer da seção, primeiramente será apresentada uma descrição da linguagem ACT; em seguida, apresenta-se a linguagem SALSA construída para ser utilizada principalmente com a Internet, ou a WWC (World Wide Computing – Computação Globalmente Distribuída), que provê técnicas para a reconfiguração das funções descritas na programação, em tempo de execução.

2.5.1 A linguagem *ACT*

A linguagem de programação *Act* incorpora os construtores mínimos necessários para programar sistemas arbitrários baseados no modelo de atores. O programa que descreve um ator consiste de uma seqüência de *definições de comportamentos*, seguido por um *comando*. (Agha, 1986).

A sintaxe apresentada por Agha (1986) é demonstrada por:

```
<act program> ::= <behavior definition>' (<command>
<behavior definition> ::= (define (id { (with identifier <pattern>)}')
                                <communication handler>')
<communication handler> ::= (Is-Communication <pattern> do (command))
```

A descrição de um programa, como ilustrado acima, é a definição de um comportamento, seguido por uma seqüência de comandos na linguagem. A definição de um comportamento precisa de um identificador padronizado e de um manipulador de comunicações, que são as mensagens enviadas e recebidas para o processamento das ações delegadas pelos papéis atribuídos a cada ator. A definição do manipulador de conexões também é definida por um padrão e um conjunto de comandos.

Segundo Agha (1986), na linguagem *Act* a composição concorrente dos comandos no programa também é um comando. Basicamente, comandos especificados no código são, por padrão, executados concorrentemente. Cinco comandos são apresentados para esta linguagem:

- f) **Send**: usado para enviar mensagens (comunicações). O resultado de um comando *send* é o envio de um valor da segunda expressão ao alvo especificado pela primeira expressão. Exemplo: (send <expression1> <expression2>).
- g) **Let**: amarra expressões a identificadores no corpo dos comandos aninhados em seu escopo. Exemplo: (let (<let binding>*) do <comand>*).
- h) **Conditional**: provê um mecanismo para desvio. Exemplo: (if <expression> (then do <command>) (else do <command>)).

- i) **Become:** especifica o comportamento de tornar-se. Exemplo: (become <expression>).
- j) **New:** cria novos atores. A utilização é feita descrevendo-se a palavra-chave *new* seguida por um comportamento. Exemplo: (new <behaviour>).

O exemplo abaixo mostra um algoritmo para a resolução do problema do fatorial de um número, que utiliza passagem de mensagens para implementar as estruturas de controle e a distribuição de tarefas a atores do tipo *consumidores* (*customer*), o que permite a avaliação concorrente de várias chamadas ao fatorial.

```
(define (Factorial())
  (Is-Communication (a eval (with customer ≡c)
                      (with number ≡n) do
    (become Factorial)
    (if (= n 0)
        (then (send c 1)
              (else (let (x (new FactCust (with customer c)
                                          (with number n))
                          (send Factorial (a eval (with customer x)
                                                  (with number n-1))))))

(define (FactCust (with customer ≡m)
                  (with number ≡n)
  (Is-Communication (a number k) do
    (send m n*k))
```

No exemplo de Agha (1986), a utilização de atores do tipo consumidor (*customers*) evita a criação de gargalos de execução, pois subdivide várias vezes o problema em sub-problemas, executando as diversas multiplicações necessárias de forma concorrente. Essa solução também funciona para atores cujos comportamentos são serializáveis (Mavadatt(2), 2002).

O mesmo fatorial pode ser exibido de maneira mais simples, também descrito na linguagem ACT, e apresentado abaixo.

```
(define (call Factorial (with number ≡ n)
  (if (= n 0)
    (then 1)
    (else (* n (call Factorial (with number n-1)))))
```

O exemplo acima é tido como uma construção *alto nível*, o que torna a programação em atores mais fácil. Geralmente, o compilador para uma linguagem baseada em atores prove a geração automática dos consumidores (*customers*) apropriados, podendo-se assim eliminar sua declaração (Agha, 1986).

A linguagem *Act* se apresenta eficaz na implementação básica de algoritmos concorrentes, baseados no modelo de atores. É apresentada como uma evolução da primeira linguagem de programação baseada em atores, a PLASMA, e possui as versões Act 1 (1981), Act 2 (1984) e Act 3 (1987). Segundo Liebermann (1987), a utilização desta linguagem recai principalmente na área de sistemas de Inteligência Artificial, através da implementação de problemas pela atribuição de comportamentos aos componentes do programa e à sua avaliação concorrente.

2.5.2 A Linguagem SALSA (*Simple Actor Language, System and Architecture*)

Segundo Varela (2001), aplicações rodando na Internet ou em dispositivos com recursos limitados precisam adaptar-se dinamicamente às mudanças necessárias em seus ambientes de execução, que podem ser resultados de alterações físicas ou lógicas, sem suspender o processamento de tais aplicações, ou seja, todas as adaptações necessárias devem ser feitas em tempo de execução.

Em seu trabalho, Mavadatt (2002) cita que “muitas linguagens de programação orientadas a objeto são bem sucedidas na criação de sistemas complexos, porém os níveis de adaptação, abertura e reconfigurabilidade em tempo de execução ainda não são satisfatórios”. Um aparelho celular, por exemplo, que execute um navegador WEB. Uma opção à sua pequena bateria e restrições de banda e memória seria uma reconfiguração automática, ou a abertura de uma configuração manual, para mover o *parsing* do HTML

para um servidor mais próximo, ou ainda recompor os componentes para outros dispositivos, como por exemplo, um microcomputador desktop. E é no ponto de reconfigurabilidade “on-line” que a linguagem SALSA foi idealizada.

A linguagem, SALSA (*Simple Actor Language, System and Architecture*) é introduzida para dar suporte ao modelo de atores para computação. Pode-se dizer que esta é uma extensão á linguagem Java, introduzindo assim o paradigma de programação orientada a objetos concorrentes, em comparação ao C++, que introduz o paradigma orientado a objetos na linguagem C. Segundo Varela (2001), um programa SALSA¹ pode ser facilmente pré-processado para um programa Java, onde todos os conceitos aplicados em objetos úteis em Java são preservados (p. ex. herança, generalização e polimorfismo).

De acordo com Varela (2001), um código SALSA geralmente descrito por um módulo, que é um agrupamento de comportamentos relacionados, onde podem estar contidas muitas *interfaces* e *comportamentos* de atores. A definição de um comportamento pode estender outro comportamento, o que é chamado de herança simples. Todo comportamento em SALSA estende um comportamento *top-level* chamado de *UniversalActor*, similar a Java, onde toda classe estende uma classe *Object*.. O código a seguir vem demonstrar estes conceitos.

```
module helloworld;

behavior HelloWorld {
    void act(String arguments[]){
        standardOutput ← print("Hello ") @
        standardOutput ← print("World!");
    }
}

HelloWorld helloWorld = new HelloWorld();
```

¹ Entende-se por “um programa SALSA” como sendo um programa escrito na linguagem de programação SALSA.

A partir do momento que um ator é criado e a ele é atribuído o comportamento definido no exemplo acima, uma mensagem *Act* pode ser enviada ao ator pelo sistema em execução, através de qualquer linha de comando, fazendo com que seu comportamento seja ativado. No exemplo, *StandardOutput* é um ator padrão provido pelo ambiente; uma seta (←) indica uma mensagem a outro ator; a arroba (@) indica uma continuação do tipo *token-pasing*, fazendo com que a primeira mensagem seja enviada ao ator *standardOutput* antes da segunda mensagem, mesmo sendo um ambiente de tratamento assíncrono. A criação de um ator é feita pela instanciação de um comportamento SALSA, que retorna uma referência a esse novo ator, mostrado na última linha do exemplo, onde *helloWorld* será a palavra-chave usada como referência ao ator criado.

Para o tratamento de situações onde o processamento paralelo exige um grande poder computacional, a linguagem SALSA faz a implementação de três formas de continuações:

- k) Continuação do tipo *token*: é utilizada para acomodar a passagem de retornos de mensagens, ou *tokens*, através de atores do tipo *consumidor*, ao qual o *token* é enviado depois do processamento de tal mensagem. Assim pode ser montada uma corrente de continuações de *tokens*.
- l) Continuação do tipo *join*: implementa o controle sobre um conjunto de vários *tokens* retornados por um ou múltiplos atores. Assim, um ator consumidor recebe uma matriz com todos os *tokens* de múltiplos atores, desde que tenham finalizado o processamento de suas mensagens.
- m) Continuação do tipo *first-class*: permite a um ator delegar o processamento de mensagens a uma terceira parte inserida no processo, independente do contexto no qual o processamento da mensagem estiver inserido, ou seja, independente da continuação atual para um dado *token* da mensagem.

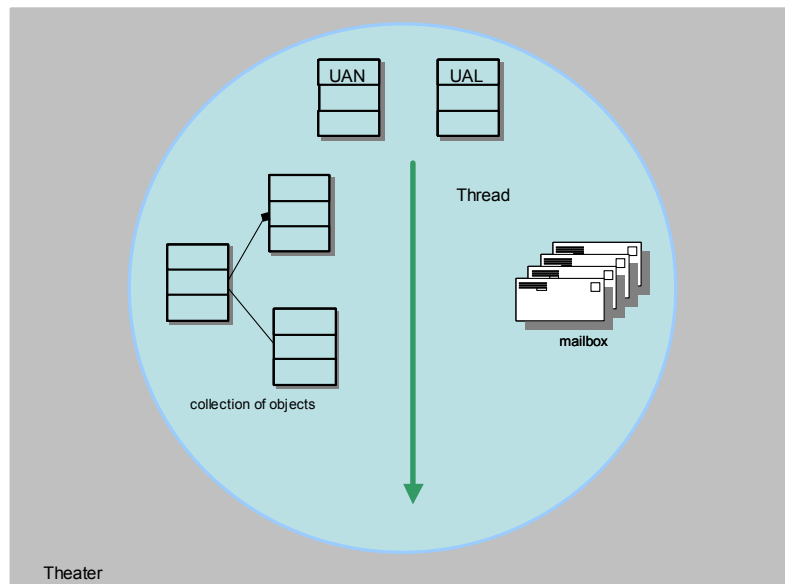
A principal aplicação da linguagem SALSA é a computação mundialmente distribuída (WWC – *World Wide Computing*), pois segundo Varela (2001) possui as

características necessárias para tal tarefa, visto sua forte característica assíncrona e o método de troca de mensagens.

A partir da Figura 7, pode-se notar a presença de dois itens adicionais à definição de atores:

- n) UAN (*Universal Actor Names*) que servem para identificar os atores unicamente na rede, estando estes atores hospedados em um conjunto de máquinas virtuais chamados de *Teatros*.
- o) UAL (*Universal Actor Location*) indica a localização do ator, ou o teatro onde tal ator está executando.

Figura 7 - Estrutura dos atores em SALSA



Fonte: Varela, 2002.

Assim, segundo Mavadatt (2002) pode-se afirmar que a linguagem SALSA torna-se uma ferramenta importante para a computação distribuída e concorrente, pois existem

características comuns entre tal linguagem e a implementação de *agentes móveis*² em várias situações, o que facilita sua aplicação na WWC e também à integração total com JAVA.

2.5.3 A Proposta Acktalk

Segundo Briot (1989), do ponto de vista da implementação, a experimentação em um projeto baseado em linguagens de atores mostra que usualmente muitos protótipos são implementados antes de se classificar todos os projetos e há também a necessidade de laboratórios e máquinas específicas para a implementação de protótipos em algumas linguagens. Assim, desenvolver códigos modulares pode demorar muito, sendo que se protótipos prévios ou mesmo outros pudessem ser reutilizados, qualquer protótipo poderia ser melhorado.

Para projetar um sistema para integrar várias linguagens de atores em um ambiente singular, Briot (1989) aponta alguns objetivos que deveriam ser atingidos:

- p) Uniformidade e modularidade: unificar várias linguagens de atores em um ambiente comum.
- q) Minimalidade e extensibilidade: moldar um *kernel* mínimo que possua uma semântica mais geral para as linguagens de atores, que possa ser estendido.
- r) Ambiente integrado: não restringir o sistema a um modelo semântico e uma implementação crua.

Assim surge o *Actalk*: um *framework* utilizado como *testbed* para linguagens baseadas no modelo de atores, o qual provê um *framework* para auxiliar na análise e na classificação de construtores de linguagens e mecanismos existentes, no projeto de novas linguagens através da derivação e da combinação de existentes, e no teste destas com

² Agentes móveis são apenas citados, pois não fazem parte do foco deste trabalho.

programas atuais sobre um rico ambiente de programação. Sua implementação baseia-se em um *kernel* para a linguagem *Smalltalk-80*, que, segundo (Briot, 1989) foi escolhida para acomodar tal projeto por ter um ambiente de programação orientado a objetos flexível, e também por possuir todas as entidades necessárias para a construção de atores: objetos, classes e mensagens, bem como processos e semáforos para dar apoio à concorrência.

2.5.3.1 Transformação de objetos do Smalltalk-80 em atores

De acordo com Briot (1997), objetos no *Smalltalk-80* são entidades passivas, ativadas pela requisição de outros objetos, através do envio de mensagens de forma síncrona, onde enviar uma mensagem representa transferir a atividade de um objeto para outro, não possuindo assim sua própria ativação.

Para permitir a concorrência, o *kernel* do *Actalk* insere o processo de múltiplas ativações, fazendo com que várias mensagens possam ativar concorrentemente objetos múltiplos, implementando também o processo *exclusão mútua* (ou simples ativação), permitindo que um objeto tenha uma ativação própria (sem a necessidade de ativações pela passagem de mensagens), tornando-se *ativo* e *autônomo*. (Briot, 1989)

Assim, como o receptor agora pode processar sua própria mensagem, o transmissor não precisa mais suspender sua atividade para processar a mensagem. Como nenhuma resposta é necessária, o transmissor pode retomar sua execução imediatamente após o envio da mensagem, que se torna um método unidirecional e assíncrono, obrigando o receptor a ter uma caixa de correio (*mailbox*) associado a ele, pra processar, a qualquer momento após a chegada, as mensagens que lhe foram enviadas. O processamento de tais mensagens é chamado de comportamento. (Briot, 1997).

Quanto à sua utilização, Briot (1995) cita que várias extensões foram definidas como subclasses das classes componentes do *kernel* do *Actalk*, em: modelos de linguagem e construtores; modelos de comunicação; e esquemas de sincronização. *Actalk* também tem sido utilizado como a base para a simulação de projetos de engenharia de software, na

construção de sistemas multi-agentes, em trabalhos que utilizam processamento de linguagem natural, a aquisição de conhecimento e em trabalho de supervisão de redes de telecomunicações.

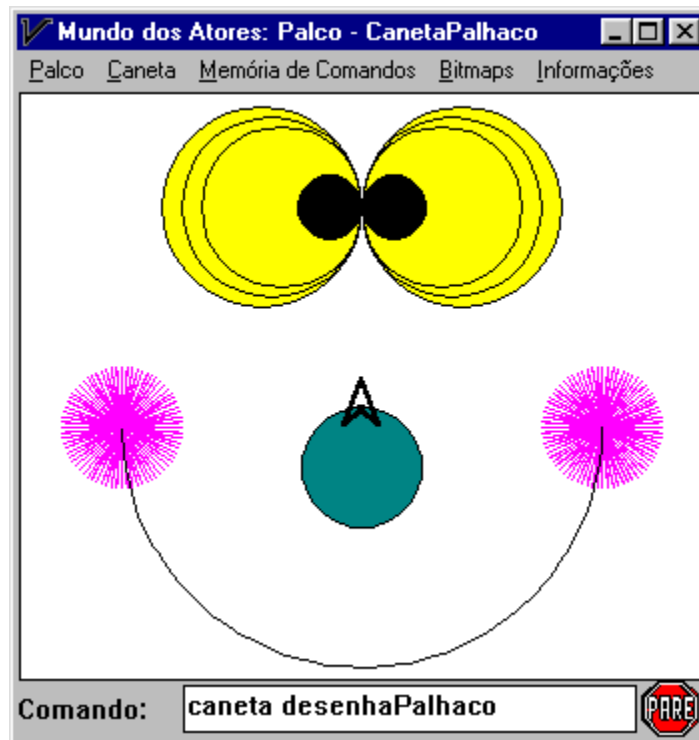
2.5.4 A Ferramenta Mundo dos Atores

A ferramenta Mundo dos Atores foi proposta por Mariani (1998) e pode ser definida como uma ferramenta baseada no Modelo de Atores, e não uma linguagem propriamente dita, tendo que sua construção foi inspirada na linguagem *Logo* (Papert, 1980). Criada para suportar a aprendizagem pelo método de ensino não seqüencial, esta pode ser utilizada em disciplinas introdutórias de programação de computadores, fazendo com que os aprendizes manipulem objetos antes mesmo do contato com a teoria de *orientação a objetos*; bem como na exploração de conceitos matemáticos e no exercício de aspectos como a criatividade (Mariani, 1998).

Segundo Pianesso (2002), para facilitar a interação com o usuário, a ferramenta apresenta a metáfora de um *Palco* (Figura 8), onde os usuários são introduzidos a uma analogia com um teatro, podendo conduzir manipulações com elementos virtuais, como a caneta, os atores e o próprio palco, sendo que têm correspondência direta com elementos concretos equivalentes.

A interação do sistema é feita através da interface da ferramenta, que apresenta o Palco, e um campo de entrada de dados. Cada ator mantém uma referência ao Palco, e este para os atores nele incluídos. Cada Palco e cada ator possuem um conjunto de ações básicas que podem ser conhecidas pelo aprendiz, designadas como uma espécie de roteiro, indicando os papéis do ator em um palco (Mariani, 1998).

Figura 8 - O Mundo dos Atores: tela que representa o Palco , na ferramenta.



Fonte: Mariani, 1998.

A descrição dos papéis que cada ator irá desempenhar no palco é feita através de comandos inseridos na própria interface. Indica-se, primeiramente o ator designado, seguido pelo nome do comportamento e, se houverem, valores que parametrizem tais comportamentos. Por exemplo:

```
caneta gira: 10.
```

O código acima descrito informa ao ator caneta que deve realizar um giro em seu próprio eixo, 10 graus no sentido horário. O comportamento indicado *gira* é tido como umas das ações básicas definidas na linguagem.

Além das ações básicas, novas ações podem ser incorporadas ao repertório do palco ou dos atores, e o palco também pode ser modificado para se adequar a uma nova situação. Novos palcos podem ser criados por especialização dos existentes, assim como atores, onde a definição de papéis possibilita o estabelecimento de comportamentos diferenciados para os atores, como por exemplo, a movimentação no palco e a interação com outros atores.

Figura 9 - Exemplos de criação de novas ações no palco desenhaPalhaço.

```
desenhaPalhaco
    "Desenha o rosto de um palhaco"

    caneta desenhaOlhos;
    desenhaNariz;
    desenhaBoca.
```

Figura 2

```
circunferencia: raio
    "Desenha uma circunferência conforme
    o raio passado como parâmetro"
    |lado|
    lado := (2 * Float pi * raio) / 36.
    caneta gira: 5.
    36 vezesRepita: [
        caneta anda: lado;
        gira: 10
    ].
    caneta gira: -5.
```

Figura 3

```
desenhaOlhos
    "Desenha os olhos do palhaço"

    caneta semRastros;
    anda: 100;
    fixaCorDosRastros: Preto.
    2 vezesRepita: [
        caneta desenhaUmOlho;
        gira: 180.
    ].
    caneta semRastros;
    anda: -100;
    comRastros.
```

Figura 4

Fonte: Mariani, 1998.

A Figura 15 subdivide-se em várias figuras, apresentadas no trabalho de (Mariani, 1998) que representam a descrição de ações que podem ser desempenhadas pelos atores no palco. Iniciado o processo de animação, cada ator passa a “desempenhar” seu papel de forma independente. O comportamento geral do sistema resulta dos comportamentos individuais de cada ator (Mariani, 1998). O controle da execução dos papéis dos atores é realizado pelo palco através do mecanismo de paralelismo por eventos discretos. Cada ator então, a cada instante, executa sem interrupções cada uma das sentenças incondicionais a ele atribuídas, às quais uma condição associada é verdadeira (Wazlawick et al, 2001).

A partir dos modelos de linguagens apresentados tem-se que o modelo de atores é ideal para qualquer algoritmo que precise de um processamento paralelo, preferencialmente distribuído em uma Internet. Tais propostas se baseiam na interação entre os atores através da troca de mensagens de forma assíncrona, executando tais mensagens de forma não

determinística, a não ser que continuações sejam utilizadas. Todas as linguagens se mostram eficazes na implementação de sistemas, porém percebe-se que os usuários devem possuir amplos conhecimentos em programação para sua utilização.

Focando o mesmo paradigma de programação concorrente, o trabalho de (Mariani, 1998) apresenta uma abordagem baseada no *Smalltalk*, porém implementando um processamento síncrono dos comportamentos pertencentes aos atores e seus objetos. O framework *Actalk*, proposto por Briot (1989) também utiliza a ferramenta *Smalltalk*, e serve como um *testbed* para várias linguagens. Ambas abordagens utilizam todas as facilidades da Programação Orientada a Objetos já encapsulada na linguagem.

O próximo capítulo apresenta as características da linguagem proposta, seguida pela sintaxe proposta e exemplos de sua utilização. Assim pode ser feito um comparativo entre as apresentações feitas na revisão bibliográfica e a proposta aqui inserida: a de criar uma sintaxe mais antropomórfica para o modelo baseado em atores, independente de ferramentas existentes que são baseadas nos paradigmas clássicos de programação.

3 Programação de atores baseada em comportamentos

Com o intuito de compor uma forma de programação mais antropomórfica, em contraposição aos modelos apresentados no capítulo anterior, é apresentada neste trabalho uma proposta de um modelo de programação de atores por composição de comportamentos. A base para uma possível implementação de tal linguagem seria a ferramenta *Mundo dos Atores* (Mariani, 1998), pois se tem que esta proposta deva ser implementada seguindo um modelo de eventos síncrono que herda algumas características do paradigma orientado a objetos.

3.1 Comportamentos

Esta proposta se baseia na atribuição de **comportamentos** a atores pertencentes ao escopo do problema que se deseja resolver com programação, onde se pode apresentar um comportamento como uma ação contínua, que tal ator irá executar durante todo seu tempo de vida no programa, podendo este ser modificado por outro ator, dependendo das atribuições que cada um possuir. O comportamento que cada ator terá define seu papel no palco, durante a execução do programa. Cada comportamento é definido e programado através do Modelo de Atores, mas o usuário final da linguagem de comportamentos não precisaria descrever seu código, pois seu código ficaria encapsulada no próprio modelo.

Uma vez que o ator é criado, aparecendo ou não³ no ambiente de programação, este já possui um comportamento a ele designado, o qual será executado durante toda a sua presença no ambiente do programa.. Tais comportamentos também podem ser condicionados a situações que definem os momentos em que os atores irão desempenhar as ações referentes a tais comportamentos, sendo estas condições regidas por tempo,

³ A percepção do usuário quanto ao comportamento não é o fator predominante para a existência de comportamento neste objeto, pois os comportamentos não necessariamente precisam estar visíveis ao usuário.

proximidade com outros objetos, presença no palco, contato com outros atores e até mesmo por modificações ocorridas em atributos de outros atores.

3.2 Comportamentos como Estereótipos de Classes

Na definição de comportamentos pode-se afirmar que estes possuem todas as características de uma classe, podendo assim ser definidos. Comportamentos podem ter atributos, instâncias, associação uns com os outros, generalização e agregação.

As classes, em UML podem estar estereotipadas de acordo com vários critérios, como por exemplo, seu objetivo dentro de um programa. Segundo Booch (2000) *estereótipo* se referencia a uma classificação de alto nível, de um objeto que proporciona uma idéia do tipo de objeto que se trata, sendo assim uma extensão para a própria linguagem. Considera-se um estereótipo como um metatipo, pois cada um cria o equivalente a uma nova classe no metamodelo da UML.

Um estereótipo pode ser usado para especificar diferenças de utilização ou de sentido entre dois elementos com uma estrutura semelhante. Sua composição possui a indicação da sua classe base e, opcionalmente, uma representação gráfica (ícone) correspondente. Segundo (Furlan, 1998), existem certas restrições na construção de estereótipos:

- a) Devem estar baseados em certas categorias existentes no metamodelo.
- b) Podem apenas estender essas categorias de certos modos predefinidos.
- c) Todo elemento na UML pode ser, no máximo, um estereótipo que, por sua vez, é omitido quando a semântica predefinida da UML for suficiente.

Na seção seguinte são apresentadas algumas classificações decorrentes do processo de atribuição aos objetos, da apresentação destes no ambiente de programação e da própria criação de comportamentos.

3.3 Classificação de Comportamentos

Primeiramente, uma classificação descritiva se mostra interessante para uma definição mais clara do que é um comportamento, e como este se relaciona com os objetos durante o ciclo de programação a ser proposto.

3.3.1 Comportamentos Latentes e Evidentes

Todo objeto, assim que criado, possui um comportamento, mas não é necessário que este comportamento seja perceptível ao usuário. Assim, a primeira classificação a ser apresentada está ligada à percepção destes pelo usuário, dividindo-se em evidentes e latentes (não evidentes).

Comportamentos são ditos evidentes quando são percebidos pelos usuários no programa, ou seja, enquanto os comportamentos estiverem ativos a ação que tal objeto executa pode ser percebida a todo instante.

A classificação tida como *latente* pode ser definida como um comportamento que não se encontra necessariamente evidenciado no objeto mesmo que esteja ativo, ou seja, a execução aparente deste comportamento não é necessariamente visível ou perceptível pelos usuários no ambiente em qualquer momento, mas apenas em certos momentos oportunos, que podem ocorrer em função da interação com outros objetos ou mesmo em função do tempo. Por exemplo, o comportamento de “virar a direita quando chegar a uma esquina” só pode ser percebido quando o objeto efetivamente chegar a uma esquina, embora esteja sempre ativo de forma latente mesmo quando não existem esquinas no palco.

3.3.2 Comportamentos que afetam outros objetos

Uma classificação envolvendo objetos que afetam o comportamento de outros objetos poderia ser proposta, como, por exemplo, no comportamento “repelir”, fazendo com que outros objetos fujam do objeto que possui este comportamento ativo, porém, após uma breve análise, pode-se perceber que este tipo de comportamento que afeta outros objetos na verdade pode ser interpretado de outra forma. Ao invés de implementar o comportamento “repelir Mosquito” na classe “Citronela”, poderia ser implementado o comportamento

simétrico “fugir de Citronela” na classe “Mosquito”, o que parece bem mais sensato quando se fala em “comportamento” em oposição a “comando”.

3.3.3 Comportamentos Básicos e Compostos

Outra questão que pode ser apresentada está ligada diretamente à existência de comportamentos básicos (ou atômicos) e compostos. Pode-se definir um comportamento como básico quando este não for composto por nenhum outro comportamento, ou seja, é tido como um componente atômico do ambiente, pois é o elemento primitivo para a construção de outros comportamentos. Segundo (Pianesso, 2002), comportamentos básicos são definidos como comportamentos simples, que não necessitam de outros comportamentos para serem explicados. Estes, normalmente, são mais genéricos, e servem como ponto de partida para a geração de outros comportamentos mais complexos, com finalidade mais específica. Por exemplo, o comportamento básico poderia ser “andar”, e um comportamento composto mais específico poderia ser “andar em círculos de raio 10”.

Por outro lado, pode-se definir um comportamento como composto quando este for formado pela junção de dois ou mais comportamentos básicos, para atingir determinado objetivo, necessário como comportamento para determinada ação a ser desempenhada pelo objeto durante sua existência no ambiente de programação.

3.3.4 Comportamentos Terminais

Podem ser definidos ainda os comportamentos terminais. Um comportamento é considerado como terminal quando a ação por ele descrita, por sua natureza, só pode ser executada uma única vez durante a existência do ator no ambiente. Um exemplo seria o comportamento “destruir-se”, que quando ativado faz com que o ator desapareça do palco, ou o comportamento “tornar-se X” que faz com que o ator mude de classe, e portanto, mude todos os comportamentos que originalmente possuía.

3.4 Atributos

Atributos podem ser definidos como elementos que permitem a valoração das características dos comportamentos, fazendo parte da lógica envolvida no processo de definição de cada um dos comportamentos analisados. São utilizados para diferenciar a

forma como o ator desempenhará o papel a ele definido através da descrição do comportamento.

Cada vez que um objeto é criado, e um comportamento a ele é designado, os atributos devem ser ajustados de acordo com a situação. Os valores de tais atributos podem ser alterados durante o ciclo de vida do objeto, dependendo de como foram definidos, seja através da interferência de outros objetos ou até mesmo de elementos padrões, inerentes ao ambiente, como por exemplo, o tempo de execução.

A tipificação dos atributos é apresentada na tabela 01.

Tabela 1 - Definição dos tipos primitivos da linguagem proposta.

Tipo	Valores possíveis
Inteiro	Conjunto dos valores inteiros, positivos e negativos.
Real	Conjunto dos números Reais.
Caractere	Caracteres alfanuméricos e símbolos.
Lógico	Conjunto com conteúdo binário, representando verdadeiro ou falso.
Classe	Referencia uma classe de objetos, podendo ser definida na descrição do programa desenvolvido ou estar encapsulada no próprio modelo. Pode ser utilizada para indicar um conjunto de objetos.
Objeto	Pode ser definido como a instância de uma classe, sendo assim um exemplar de um conjunto de objetos definidos através de uma classe.

O uso de classes como atributos serve para que o programador possa desenvolver condições e comportamentos cuja lógica define a interação com mais de um objeto ao mesmo tempo. Por exemplo, se um comportamento fugir tivesse que ser atribuído a um ator do tipo zebra, todas as zebras pertencentes a um grupo deveriam fugir de um ator do tipo leão, que também pode possuir mais de um exemplar (que pode ser definido como uma instância). Assim, seriam definidas as classes Zebra e Leão, e que qualquer zebra teria como comportamento “fugir de Leão”, ou seja, fugir de qualquer leão e não de um específico.

Os comportamentos e condições definidos (e previamente encapsulados) no ambiente de programação possuem atributos que, assim como os atributos do palco, precisam apenas

ser ajustados pelo programador. Podem existir outros atributos, definidos pelo próprio programador, que pertençam tanto aos atores, comportamentos e condições desenvolvidas, quanto aos atributos tidos como gerais, que servem ao controle da interação dos objetos no ambiente.

A tabela 02 possui a definição de atributos sugeridos para os problemas propostos.

Tabela 2 - Descrição dos atributos primitivos de cada ator

Atributos	Definição
Direção	Informa o sentido em que o ator estiver andando no ambiente, possuindo assim um valor que informa o ângulo que o objeto está posicionado em relação ao "norte" absoluto do ambiente (aponta para a parte superior da tela, se visto pelo usuário). Assim que criado, o objeto possui a direção do "norte" do ambiente no programa. Pode ser tipificado como um valor constante ou como uma função dependente de tempo ou interação com outros objetos.
Velocidade	Quantidade de passos/unidade de tempo discreto do sistema a que o ator (objeto) se movimenta no ambiente. Pode ser tipificado como um valor constante ou como uma função dependente de tempo ou interação com outros objetos.
Imagem	Figura que o ator apresenta como sua aparência no ambiente
Posição	É designado como um par de coordenadas $\langle x, y \rangle$, que apresenta a posição relativa do ator, sendo $\langle 0, 0 \rangle$ o centro do palco.
TempoDeVida	Indica à quantos instantes de tempo o ator encontra-se no palco.

3.5 Descrição dos Comportamentos

Nesta fase do trabalho, apresenta-se a descrição dos comportamentos pré-analisados no projeto, pontuando as diferentes conotações inferidas a cada um de acordo com anteriormente. Todas as definições de comportamentos a seguir estão descritas segundo a sintaxe definida por:

```
nome_do_comportamento(nome_do_parâmetro1 : tipo_do_parâmetro1;
...; nome_do_parâmetron : tipo_do_parâmetron);
```

onde:

- nome_do_comportamento: indica a designação a que o comportamento deve ser referido no momento da programação do problema proposto;
- nome_do_parâmetro_n: denota a identificação do parâmetro a que o comportamento estará sujeito;
- tipo_do_parâmetro_n: tipifica o parâmetro quanto à valoração aceitável pelo mesmo. Os tipos podem ser conhecidos como inteiros, reais, caracteres e lógicos, bem como funções definidas pelo usuário, no caso dos parâmetros serem variáveis e não valores pré-determinados.

3.5.1 Comportamentos Básicos Evidentes

A tabela 3 apresenta os componentes classificados como básicos e evidentes, de acordo com a observação feita nos jogos analisados. Tais comportamentos são assim classificados, pois, a partir do momento que o ator adquirir tais comportamentos, as ações características serão perceptíveis no ambiente.

Tabela 3 - Descrição dos comportamentos básicos evidentes encontrados

Comportamento	Descrição
andar(velocidade : Numero);	Movimenta-se uniformemente em frente, na velocidade indicada pelo parâmetro repassado. Esta velocidade pode ser informada através de uma constante, equação ou função dependente do tempo, podendo, por exemplo, definir uma aceleração em relação ao tempo decorrido do programa.
girar(velocidade_radial : Numero);	O objeto permanece girando em torno de seu próprio eixo, na velocidade indicada pelo parâmetro.
emitir_som(som : Som);	O objeto emite determinado som, definido pelo parâmetro <i>som</i> .
gerar(classe : Classe)	Gera um novo objeto da classe passada como parâmetro, na mesma posição em

	que se encontra seu gerador.
modificar(atributo : Atributo, valor : Numero);	Modifica atributos do próprio objeto, proprietário do comportamento, sendo que este pode receber um valor constante ou ser definida uma função para tal atributo.

No caso do comportamento “Modificar” assume-se que seja evidente mesmo que os valores dos atributos não sejam potencialmente visualizados no ambiente. Estima-se, sim, que, caso fossem visualizados os atributos, sua alteração seria evidente a cada instante de tempo em que o comportamento estivesse ativo.

3.5.2 Comportamentos Básicos Latentes

A característica de um comportamento ser básico e latente recai sobre a necessidade de haver uma relação com outro objeto do ambiente ou condição específica, e também porque estes podem pertencer a um ator que já possui outro comportamento, permanecendo escondidos até que algum evento faça com que as características deste comportamento se tornem ativas.

Tabela 4 - Relação dos comportamentos básicos latentes encontrados

Comportamento	Descrição	Quando se torna visível
bater(classe : Classe)	Caso o objeto esteja se movendo e toca em outro objeto da classe especificada, o objeto proprietário do comportamento muda de direção.	É percebido apenas quando houver o toque de dois objetos no palco.
fugir(classe : Classe; velocidade : Numero)	Se houver pelo menos um objeto do tipo definido por “classe” no palco, o objeto proprietário do comportamento foge dele na velocidade indicada.	É percebido apenas quando um objeto da classe passada como parâmetro se encontra no palco.
perseguir(classe : Classe; velocidade : Numero)	Se houver pelo menos um objeto do tipo definido por “classe” no palco, o objeto	Depende também da presença de um objeto da classe indicada

	proprietário do comportamento o persegue na velocidade indicada.	como parâmetro.
derrapar(permanência : Inteiro)	Quando a velocidade linear ou a direção do movimento do objeto for alterada, o objeto permanecerá no movimento anterior durante o tempo de permanência, dando a impressão de estar derrapando.	Só é perceptível quando o movimento do objeto muda de direção ou velocidade.
considerarPalcoCircular()	Se o ator sair por um lado do palco, imediatamente surge no lado oposto, dando a impressão de que o palco é circular.	Só pode ser percebido se o objeto sair momentaneamente do palco.

3.5.3 Comportamentos básicos terminais

O comportamento `destruirSe()` só poderá ocorrer uma vez para o objeto pois o mesmo, após a execução, não existirá mais no palco; e o comportamento `TornarSe(classe : Classe)` quando ativado faz com que o objeto mude de classe, perdendo portanto todos os seus comportamentos anteriormente definidos (inclusive o comportamento `TornarSe`).

Tabela 5 - Apresentação dos comportamentos básicos terminais encontrados

Comportamento	Descrição
<code>destruirSe()</code>	O objeto se destrói, não fazendo mais parte do ambiente.
<code>tornarSe(classe : Classe)</code>	O objeto proprietário passa a ser um objeto de outra classe, adotando os comportamentos descritos para o tipo informado como parâmetro.

3.6 Composição de Comportamentos

Para determinadas situações apresentadas nos problemas analisados, faz-se necessário o uso de comportamentos que não estão definidos nos comportamentos classificados como

básicos. Entretanto, é possível descrever novos comportamentos a partir destes, os quais são chamados de comportamentos compostos. Os elementos que participam dessa composição devem ser obrigatoriamente comportamentos básicos e/ou outros comportamentos compostos já definidos.

Qualquer papel que possa ser desempenhado por um ator e que precise, simultaneamente, desenvolver as ações características de dois comportamentos básicos, é considerado um comportamento composto. Pode-se citar, como exemplo, um ator que represente uma ambulância, à qual permanece por toda sua existência no ambiente andando em linha reta no palco, e emitindo o som característico de uma sirene. Ambos comportamentos são caracteristicamente básicos, os quais podem ser agregados em um único comportamento composto, como segue:

```
COMP andaEmAlerta ← {
    anda(10); emitirSom('sirene.wav').4
}
```

O exemplo acima faz apenas a composição simples de dois comportamentos, porém conceitos da Programação Orientada a Objetos são utilizados para indicar algumas composições de comportamentos, sendo apresentados nas próximas seções.

3.6.1 Herança

O primeiro método de composição de comportamentos apresentado é a herança que, segundo (Furlan, 1998) é o mecanismo para expressar a similaridade entre classes, simplificando a definição de Classes similares a outras que já foram definidas, tornando atributos e serviços comuns em uma hierarquia de Classe. Para (Larmann, 2000), herança é um mecanismo de software para implementar aderências dos subclasses às definições de superclasses .

A herança entre comportamentos pode ocorrer de forma simples (ou hierárquica), onde o comportamento herda as propriedades de sua superclasse em uma linha hierárquica (Pianesso, 2002). Assim, um comportamento definido como filho (subclasse) do

⁴ sirene.wav é somente um exemplo de arquivo que contém o som característico de uma sirene.

comportamento pai (superclasse) possui todos os comportamentos e atributos do pai, além de ter os seus próprios.

Um comportamento pode herdar as propriedades de várias superclasses não relacionadas hierarquicamente, ou seja, a herança múltipla, ou utilizar a herança seletiva, que permite a seleção de características de uma ou mais superclasses para compor o novo comportamento (Rumbaugh, 1991).

A reutilização de componentes é um ponto forte da herança, pois segundo (Kamienski, 1996), a herança viabiliza a construção de sistemas a partir de componentes reusáveis facilitando assim extensibilidade em um mesmo sistema, diminuindo a quantidade de código para adicionar características a sistemas já existentes. Também facilita a manutenção de sistemas, pois provê maior legibilidade e diminui a quantidade de código a ser acrescentado (Kamienski, 1996).

O método de composição de comportamentos pode fazer a utilização de outros artefatos para a definição de novos comportamentos que não apenas a herança, pois muitas vezes a definição de herança múltipla não é suficientemente aproveitada para a junção de dois comportamentos para a criação de outro. A seguir outros métodos são apresentados.

Como um exemplo de herança nesta linguagem, pode-se citar o comportamento *gerar*(classe:Classe), que simplesmente gera um novo elemento na posição em que se encontra o gerador, e o comportamento *criar*(classe:Classe; coordX, coordY : Número), que herda as ações desempenhadas pelo comportamento *gerar*, porém faz com que o elemento se posicione nas coordenadas definidas nos parâmetros passados na função, onde *coordX* é a coordenada horizontal e *coordY* é a coordenada vertical para tal ator.

3.6.2 Agregação (composição)

Segundo (Pianesso, 2002), a relação de composição permite que objetos sejam compostos pela agregação de outros objetos ou componentes, ou seja, a formação de uma nova classe como um agregado de classes preexistentes, fazendo com que um comportamento composto possa ser criado por meio de uma relação *parte_de* ou

componente_de. A composição é utilizada neste trabalho na definição de novos comportamentos como, por exemplo, o comportamento *consumirCombustível()*, que é definido como a agregação dos comportamentos básicos *andar((velocidade))* e *modificar((combustível,combustível-1))*, apresentando um comportamento resultante que faz o objeto andar e a cada passo reduzir a quantidade corrente do atributo “combustível”.

3.6.3 Especialização de parâmetros

Para se definir realmente o que um ator vai executar durante seu tempo de vida no ambiente de programação, a definição de seu comportamento deve ser especificada ao máximo, valorando seus atributos. Uma forma possível para a realização desta definição é a utilização de um certo tipo de especialização.

De acordo com Pianesso (2002), um novo comportamento herda suas propriedades de suas superclasses, entretanto, se a propriedade já se encontra definida em nível de subclasse, é esta a definição que prevalece. Esta característica denomina-se *overriding*, porém só pode ser aplicada quando a propriedade especificada na classe herdeira possui o mesmo nome da propriedade herdada.

O comportamento *AndarADezPorHora()*, definido como *Andar(10)* pode se considerado um exemplo de especialização por fixação de parâmetro, pois mantém o comportamento original, porém fazendo a especialização de um parâmetro através da fixação do valor 10 para ele.

3.6.4 Criação de novos comportamentos

Através da experimentação dos comportamentos básicos definidos anteriormente nos jogos propostos, observou-se que apenas a simples aplicação destes não foi eficaz para ter-se a definição completa dos papéis desempenhados por atores e outros objetos. Assim, novos comportamentos devem ser criados, a partir de situações singulares ou não, que

referenciem os comportamentos básicos, utilizando os artifícios mencionados anteriormente como métodos de composição.

A utilização dos métodos de composição, pode se tornar uma atividade complicada, que acabe por gerar incertezas na descrição de comportamentos, e até definições equivocadas de composições. Este trabalho não se propõe a analisar a semântica da composição de comportamentos em profundidade, como fez Pianesso (2002), mas apenas considerar sua possibilidade como ferramenta de programação de atores. Assim, a linguagem terá três formas básicas de composição de comportamentos, a saber:

- a) Composição simples de comportamento. Exemplo: pode-se definir um comportamento como *andaEmCirculos(passo,ângulo)*, sendo a junção dos comportamentos básicos *anda(passo)* e *gira(ângulo)*.
- b) Condicionamento de comportamento: ex. colocar uma condição no comportamento, como *andaSeTocado(velocidade,classe)*, que corresponde a junção da condição *aoTocar(classe)* com o comportamento *andar(velocidade)*.
- c) Fixação de parâmetro. Um exemplo seria o novo comportamento definido como *andaRapido()*, que poderia ser o comportamento *andar(passo)* com o parâmetro *passo* fixado em 50⁵.

Os comportamentos compostos, assim definidos, também podem ser classificados como latentes ou evidentes, em função de estarem ou não visíveis todo o tempo ou não.

3.7 Condições

Desde os tempos de Charles Babbage, com sua máquina analítica, condições são tidas como a principal característica da programação, apresentando a possibilidade de desvios

⁵ O valor 50 pode ser considerado rápido, pois se movimentaria 50 posições na tela por unidade de tempo discreto. O tempo discreto do ambiente deveria também ter valores próximos ao tempo real.

condicionais (Henessy, 1997). Para que certos tipos de comportamentos possam ser criados, a descrição de condições se torna necessária.

Como parte da composição da linguagem, condições são definidas como estereótipos, pois se apresentam como um conjunto de classes especiais. As condições básicas aqui descritas também foram identificadas a partir da experimentação com os jogos envolvidos.

As condições irão restringir o período de ativação dos comportamentos, trabalhando como eventos de acionamento, de temporização, de decisão e de interação com outros objetos do palco. Segue abaixo um exemplo de condição aplicada a um comportamento:

```
aoTocar(parede) ⇒ emitirSom('ping.wav').
```

O exemplo acima deve ser escrito dentro da definição de um ator, que será o ator proprietário de tal comportamento. A condição *aoTocar(parede)* restringe o comportamento *emitirSom('ping.wav')*, fazendo com que o ator proprietário emita o som indicado por parâmetro (ping.wav) no momento em que tocar um objeto do tipo *parede*.

A tabela abaixo apresenta algumas condições obtidas a partir dos projetos analisados.

Tabela 6 - Relação das condições primitivas encontradas nos jogos propostos

Condição	Descrição
<code>sempre()</code>	Condição que é sempre verdadeira, ou seja, enquanto o ator estiver ativo no ambiente, executará o comportamento associado a esta condição. Tem equivalência direta a uma condição vazia.
<code>aDistância(distancia : Número; alvo : Ator)</code>	Esta condição é verdadeira se um objeto da classe informada estiver a uma distância menor que o parâmetro "distância" do objeto dono do comportamento.
<code>aoTocar(objeto : Ator)</code>	Esta condição é verdadeira quando o objeto dono do comportamento estiver tocando um objeto da classe indicada.
<code>sempreQue(valor1 : Número; comparação : Comparação; valor2 : Número);</code>	Esta condição é verdadeira se valor1 for comparável a valor2 pelo elemento do tipo "Comparação" (>, <, <=, >=, =, <>).
<code>noIntervaloDeTempo(início, fim : Número)</code>	Esta condição é verdadeira quando o tempo de vida do ator estiver entre

	início e fim, inclusive.
aCada(tempo : Número)	Esta condição se torna verdadeira quando o tempo de vida do ator for múltiplo do parâmetro "tempo".
comProbabilidade(percentual : Número)	Esta condição se torna verdadeira a cada instante com probabilidade dada por "percentual" (entre 0 e 1).
ForaDaTela()	Condição verdadeira cada vez que o ator ultrapassar os limites da tela.

Além da tabela apresentada acima, pode-se definir novas condições, de acordo com a necessidade do programador ao definir um novo palco. O relacionamento destas condições com atributos e comportamentos, e a sintaxe para a definição das mesmas, serão descritos nas próximas seções deste trabalho.

4 Definição da Linguagem

Este capítulo apresenta os componentes, as definições e as principais características da linguagem de programação, baseada em comportamentos, que se pretende validar neste trabalho. Tais definições baseiam-se nos conceitos abordados nos capítulos anteriores, apresentando definições e relacionamentos entre os comportamentos, as condições e os atributos já classificados, bem como a construção de novos itens para a adequação aos jogos analisados.

4.1 Componentes do ambiente de programação

Para que se possa trabalhar com a linguagem a ser proposta, abaixo se descreve os principais elementos que compõem o ambiente de programação proposto.

4.1.1 O Palco

Numa analogia com um teatro, o lugar onde os atores atuam é o *Palco* (Mariani, 1998). Utilizando-se da conceituação de um modelo baseado em atores, o *Palco* serve como uma metáfora para a designação do local onde tais objetos podem “atuar”, ou seja, executar as ações correspondentes aos seus respectivos comportamentos, sendo que estes envolvem a participação de outros atores, fazendo com que interajam entre si, durante a execução do programa.

O palco também é o local onde objetos que não são identificados como atores estão dispostos, caracterizando assim o ambiente em que tal programa deverá estar inserido. No caso de um ambiente envolvendo a natureza, atores como onças, macacos e insetos em geral são considerados os atores, objetos ativos no ambiente; e árvores, *canyons*, montanhas e cavernas, são considerados objetos passivos, tendo finalidade apenas decorativa, ou servindo como obstáculos aos atores.

Sendo assim, o palco também delimita a ação dos atores no ambiente, restringindo a área de locomoção dos objetos ativos, bem como a posição dos objetos passivos, tornando-se assim uma espécie de coordenador dos objetos que interagem no ambiente de execução do programa.

O palco possui algumas propriedades que permitem a relação do mesmo com os atores, propriedades estas que funcionariam como se fossem “sensores”, identificando, por exemplo, a quantidade de atores, de determinado tipo, que estão ativos no palco; implicando assim no controle da criação e destruição de atores pertencentes a tal palco.

4.1.2 Os Atores

Descritos como os objetos ativos da linguagem, os atores proporcionam a ação no ambiente de programação. Cada ator deve ser definido na linguagem caso participe do programa, e a ele é atribuído um ou mais comportamentos.

Tais atores, assim que criados, aparecem no palco e passam a desempenhar a ação definida pelos comportamentos a ele atribuídos, interagindo assim com outros atores, e até mesmo com elementos definidos no ambiente, limitados pelo palco onde estão inseridos. O ator poderá permanecer executando seus comportamentos até a finalização do programa, ou até que se destrua, fazendo com que desapareça do palco, não podendo assim interagir mais com os outros componentes do palco.

Assim que criados, os atores se tornam objetos independentes, respeitando apenas as definições dos comportamentos a estes impostos durante a sua programação. A única dependência de todos os atores definidos no ambiente é a do Palco, pois tal elemento do ambiente tem total controle sobre todos os objetos que estejam em sua área de atuação, fazendo um monitoramento constante, e agregando controles importantes quanto à funcionalidade do programa.

Por possuir uma representação visual no ambiente, alguns atores podem permanecer “escondidos”; eles possuem papéis importantes para a execução do programa no ambiente, mas não precisam aparecer na tela. Tais atores possuem características de controle quanto ao comportamento global do sistema, relacionando atores entre si e também impondo as

regras inerentes ao ambiente em que estão inseridos. Um bom exemplo de tal tipo de ator são os aqui definidos como *Controladores do Jogo*, que servem para inicializar o jogo, definindo inicialmente todos os atores e posicionando-os de acordo com as regras do jogo, e também para o controle da mudança entre as diversas fases e níveis que o programa possui.

4.2 Definição da Sintaxe

Para a definição do ambiente de programação, que é composto por atributos, atores, comportamentos e condições, certas regras devem ser definidas. Isso é feito através da descrição da *sintaxe* de programação. A sintaxe ou a forma como os elementos de uma linguagem devem ser dispostos e escritos, tem efeito significativo sobre a legibilidade dos programas, e um dos problemas para descrever uma linguagem, é a diversidade de pessoas que devem entender tal descrição (Sebesta, 2000).

Por ser um padrão popular para descrição de linguagens de programação, será utilizada a forma de Backus-Naur, ou simplesmente BNF, uma metalinguagem para as linguagens de programação. Uma descrição BNF é chamada de gramática, sendo simplesmente um conjunto de regras.

A seguir são listados os elementos básicos da notação utilizada:

- a) O símbolo ::= designa a igualdade entre os componentes da sintaxe, ou seja, corresponde ao símbolo de “definição”;
- b) O caractere | (*pipe*) indica a divisão entre as diversas possibilidades que tal definição poderá assumir;
- c) O símbolo [...]* indica uma lista formada pela repetição de zero ou mais vezes o conteúdo dos colchetes (os colchetes também são símbolos da meta-linguagem, não aparecendo nas sentenças da linguagem objeto);
- d) <nome> indica os elementos não terminais, ou seja, elementos cuja construção é feita de combinações de outros elementos.

- e) *nome* indica o elemento terminal, ou literal. Sebesta (2000) apresenta estes elementos pelo nome de lexemas, que são unidades sintáticas de nível mais baixo (identificadores, literais, operadores e palavras reservadas, nesta linguagem).

Alguns literais usados nesta linguagem são:

- a) O símbolo = (sinal de igualdade) indica uma atribuição de valores, sendo que o elemento que estiver a direita será inserido no elemento à esquerda do símbolo.
- b) O ponto final (.) designa o fim de qualquer estrutura definida na sintaxe.
- c) O caractere ; (ponto e vírgula) indica a quebra entre definições de comportamentos (divisão), porém não encerra a descrição proposta.
- d) ==, >, <, >= <= são símbolos utilizados para a comparação entre dois termos.

4.2.1 Atributos

Como descrito anteriormente, atributos devem ser valorados durante a descrição do ambiente para que se possa caracterizar os comportamentos de acordo com o papel de cada ator. A sintaxe para tal tarefa é apresentada como:

<valoração do atributo> ::= <nomeDoAtributo> = <expressão>.

A sintaxe acima é dividida em:

- s) <nomeDoAtributo>: seqüência de caracteres alfanuméricos, iniciados por letra, que identifique unicamente cada atributo descrito no ambiente.
- t) *valor*: elemento que pode ser indicado por uma expressão, outro atributo ou um literal alfanumérico designando o valor do atributo.

A declaração de um atributo não possui uma sintaxe definida pois não há necessidade nesta proposta, bastando apenas que o programador coloque qualquer literal para indicar um atributo, ou fazendo o processo de valoração (descrito acima). Outra forma de definição pode ser concluída através da declaração de um parâmetro em comportamentos. O tipo que cada atributo assume depende da definição do parâmetro (ver sintaxe abaixo) ou pelo tipo de conteúdo a este atribuído, durante a valoração.

Para acessar um atributo que pertença a algum ator, utiliza-se a sintaxe:

`<acessar atributo do ator> ::= [<nomeDoAtor>:]<nomeDoAtributo>`.

Portanto, três componentes são fundamentais para tal descrição, como segue:

- `<nomeDoAtor>`: indica o nome do ator proprietário do atributo que se deseja acessar. Este componente se torna facultativo caso o acesso esteja sendo feito dentro da declaração de um ator, ou de um comportamento que tal ator possua.
- conectivo : (dois pontos): utilizado para separação entre o nome do ator e o nome do atributo, e indica também a intenção da operação.
- `<nomeDoAtributo>`: literal que denota o atributo que se deseja utilizar. Tal atributo pode ser definido pelo programador, ou utilizar algum dos atributos primitivos de cada ator, previamente definidos pela linguagem (Tabela 2).

4.2.2 Condições Compostas

Uma condição composta é descrita como uma junção de várias condições simples, ou de várias outras condições compostas já definidas na programação. Sua sintaxe é definida por:

`<condição composta> ::= <condição> | <condição composta> E <condição composta> | <condição composta> OU <condição composta> | NAO <condição composta>`

Assim, uma condição composta pode ser descrita como abaixo:

- `<condição>`: é uma condição simples (ver sessão 3.7);
- conectivo *E*: a condição composta resultante da combinação de duas condições por *E* será verdadeira quando ambas forem verdadeiras;
- conectivo *OU*: a condição composta resultante da combinação de duas condições por *OU* será verdadeira quando pelo menos uma delas for verdadeira;

- operador *NÃO*: funciona como um inversor da condição, fazendo com que esta opere de forma contrária à sua definição, sendo que se torna verdadeira quando a condição a que foi imposto tal operador for falsa.

4.2.3 Comportamentos Compostos

Um comportamento composto é descrito como a junção de comportamentos simples, ou outros comportamentos compostos, para a formação de um novo comportamento que descreva o papel de um determinado ator no ambiente.

<comportamento composto> ::= <comportamento> | <comportamento composto> ; <comportamento composto>

Assim, os elementos sintáticos são:

- <comportamento>: um comportamento simples, que pode ser considerado como um elemento primitivo da linguagem.
- conectivo ; (ponto e vírgula): indica a composição de dois comportamentos. O comportamento resultante poderá depender dos tipos de interferência de combinação definidos por (Pianesso, 2002)

4.2.4 Comportamentos Condicionados

A construção de comportamentos condicionados serve para restringir a ativação dos comportamentos compostos especificados no sistema.

<comportamento condicionado> ::= <condição composta> => <comportamento composto>.

Na definição de um comportamento condicionado, devem constar:

- <condição composta>: que definirá as situações nas quais o comportamento é ativo.
- conectivo =>: indica a imposição da condição definida ao lado esquerdo do conectivo, ao comportamento composto, que deve ser definido à direita do conectivo.

- <comportamento composto>: indica o comportamento ou conjunto de comportamentos original que fica então condicionado pela condição do lado esquerdo da implicação;
- . (ponto final): indica o término da definição.

4.2.5 Definição de Atores

<ator> ::= *ATOR* nomeDoAtor { [<comportamento condicionado>]* }

A definição do ator se baseia em:

- Palavra-chave *ATOR*: designa o início da descrição de um ator;
- nomeDoAtor: é a informação que designará unicamente a classe do ator durante o programa.;
- Caracteres { } : limitam as referências aos comportamentos condicionados, atribuídos ao ator;
- [<comportamento condicionado>]*: indica uma lista dos comportamentos condicionados atribuídos a este ator.

4.2.6 Definição de Comportamentos Reutilizáveis

Se for necessário reutilizar determinadas definições de comportamentos compostos ou condicionados, pode-se usar a estrutura definida abaixo:

**<definição> ::= *COMP* <nomeComportamento>(<parâmetro>[;<parâmetro>]*)
← { <comportamento condicionado> }**

<parâmetro> ::= <nomeDoParametro>[,<nomeDoParametro>]* : <tipo>;

Os itens podem ser descritos como:

- palavra-chave *COMP*: indica a criação de um novo comportamento.
- <nomeComportamento>: este item pode ser composto por qualquer seqüência de caracteres alfanuméricos, iniciados por letras, designa a informação que será utilizada como chave para indicar o comportamento criado.

- símbolo \leftarrow : indica definição, sendo que os comportamentos condicionados que estiverem ao lado direito do símbolo definirão o novo comportamento cujo nome é dado pelo identificador que estiver ao lado esquerdo da definição;
- caracteres { }: limitam o escopo da definição do comportamento.

Com relação à definição de <parâmetro>, esta estrutura possui os seguintes componentes:

- <nomeDoParâmetro>: pode ser definido como a identificação única de um parâmetro, sendo que a denominação deve conter apenas valores alfanuméricos, iniciada por uma letra.
- <tipo>: indica qual o conjunto de valores que podem ser atribuídos a tal parâmetro, devendo este ser um dos tipos primitivos já apresentados na Tabela 1, ou declarado como uma classe de Ator no ambiente.

4.2.7 O Programa Completo

Apresenta-se assim a definição de programa da linguagem proposta:

<programa> ::= <definição>* <ator>*

Os elementos na definição acima têm os seguintes significados:

- <definição>: declaração das definições de todos os comportamentos reutilizáveis que serão usados no programa;
- <ator>: declaração de todas as classes de atores que serão utilizadas no programa.

Propositalmente foi excluída desta definição a questão da inicialização do ambiente. Tal inicialização seria necessária e teria que ser efetuada pelo Palco e não pelos atores. Entretanto, o estudo do Palco e suas funções foge ao escopo deste trabalho. Assume-se assim que, objetos definidos pela linguagem serão convenientemente colocados em suas posições pelo palco, mas o detalhamento deste processo não será aqui definido.

4.3 Um Exemplo

Para ilustrar a sintaxe apresentada anteriormente, segue a descrição de um código de programa exemplificando um ambiente composto por dois atores, um objeto *Lançador* de bolas, que permanece todo o tempo andando em círculos no palco; e o objeto que representa a *Bola* lançada que, uma vez criada, anda em frente sempre na mesma direção até sair do palco. Se o ator Bola sair do palco ele se destrói; se o Lançador sair, ele para momentaneamente de lançar bolas.

```
#inicio PrgExemplo

COMP andaEmCirculos(passo) ← {
    anda(passo);
    gira(1).
}

COMP lancaBolas(tempo) ← {
    cada(tempo) ⇒ gera(Bola).
}

ATOR Bola {
    anda(10).
    foraDoPalco() ⇒ destroi().
}

ATOR Lançador {
    andaEmCirculos(2).
    NAO foraDoPalco() ⇒ lancaBolas(10).
}

#fim PrgExemplo.
```

Abaixo é apresentada a tabela comparativa ao código, utilizado para descrição do algoritmo proposto para o problema, nesta tabela as condições compostas e reutilizáveis são decompostas em seus elementos definidores.

Ator: Lançador	
condição	comportamento
sempre	anda(10)
sempre	gira(1)
(NAO foraDoPalco() E cada(10))	gera(Bola)

Ator: Bola	
Condição	comportamento
sempre	anda(10) .
foraDoPalco()	destroi() .

Note que a aplicação de condições a comportamentos já condicionados implica a prática na realização de um E lógico entre as condições aninhadas, como no caso de "(NAO foraDoPalco() E cada(10))".

5 Jogos Analisados

Os comportamentos e atributos descritos nas seções anteriores foram criados a partir da observação de alguns jogos clássicos de videogame. Nesta seção demonstra-se a viabilidade em reconstruir partes de tais jogos a partir da linguagem definida no capítulo anterior.

5.1 *Megamania*

O Megamania é caracterizado como um jogo de ação, onde o objetivo principal é destruir alienígenas que invadem o planeta terra. Este jogo foi criado na década de oitenta por Steve Cartwright, e posteriormente produzido e distribuído pela empresa *Activision*. Assim, o jogador é representado no palco como um canhão, que possui movimentações apenas na horizontal (esquerda e direita), e pode disparar balas contra os alienígenas, de acordo com comandos guiados pelo usuário jogador (Figura 10).

Os alienígenas podem se comportar de várias formas. Desde o simples comportamento de permanecer apenas trafegando horizontalmente em velocidades variáveis (atirando, às vezes, contra o jogador), ou fazendo movimentações variáveis em direção ao canhão do jogador, sendo que, caso o objeto que representa o alienígena encoste-se ao canhão que representa o jogador, ambos são “*desintegrados*”, sendo assim eliminados do jogo.

Figura 10 - *Screenshot* do jogo Megamania: na parte superior, os alienígenas da primeira fase; logo abaixo, o canhão que representa o jogador; e na base da tela os controles de pontuação, energia e vidas.



Fonte: <http://www.gamescreenshots.com/>

O jogo possui vários níveis, sendo que cada nível é dividido em várias fases, representadas pelos tipos de alienígenas. Passadas oito fases, os níveis começam a se repetir, porém os alienígenas mudam seu comportamento principal, especializando-se nos termos de maior velocidade, menor tempo de resposta a ataques e no lançamento de um maior número de tiros contra o canhão que representa o jogador.

Figura 11 - Três exemplos de *Screenshots* do megamania: fase 2, fase 3 e fase 4, respectivamente.



Fonte: <http://www.gamescreenshots.com/>

O jogador permanece no jogo enquanto tiver vidas. Inicialmente possui duas vidas de reserva, podendo este número ser aumentado em uma unidade, toda vez que o jogador chegue a atingir um número múltiplo direto de dez mil pontos, sendo que estes são conseguidos pelo jogador através da destruição de alienígenas e pela quantidade de energia que possuir ao finalizar uma das fases do jogo, concluída assim que todos os alienígenas da

respectiva forem eliminados. As vidas podem ser perdidas, ou seja, o jogador *morre* por motivos que podem variar entre acabar a sua energia, que varia de acordo com o tempo decorrido do jogo, ser atingido por um tiro alienígena, ou ser atingido pelo próprio alienígena.

5.1.1 Atores e Comportamentos do Jogo

Dos atores identificados no jogo apenas aqueles que não são controlados diretamente pelo usuário estão aqui descritos, seguindo as especificações impostas pela sintaxe definida anteriormente.

Um dos atores, identificado como objeto de defesa do usuário, é a Bala do Canhão. Assim que criada passa a andar para frente, seguindo sempre na mesma direção. Quando sair do Palco (extrapolar os limites definidos pelo ambiente ao Palco), ela se destrói. Se antes de sair do Palco tocar em alguma nave alienígena, transforma-se em um ator do tipo bala explodida.

```
ATOR BalaDeCanhao {
    anda(10).
    foraDoPalco() => destroiSe().
    aoTocar(Alien) => tornaSe(BalaExplodida).
}
```

As naves alienígenas, que serão definidas a seguir, atacam o canhão através de balas, comportando-se de forma idêntica à bala do canhão, porém transformando-se em um objeto do tipo Bala Explodida agora ao tocar o canhão.

```
ATOR BalaDeAlien {
    anda(10).
    foraDoPalco() => destroiSe().
    aoTocar(Canhao) => tornaSe(BalaExplodida).
}
```

Ambas as definições dos atores acima demonstram uma igualdade na definição de parte dos atores, o que sugere a utilização de um mecanismo que possa fazer com que partes da descrição de atores e de comportamentos sejam reutilizados para a definição de novos objetos para o ambiente. Assim, pode-se definir um comportamento com as definições comuns a ambos os atores, como segue.

```
COMP movimentoBala(alvo:Classe) ← {
    anda(10).
    foraDoPalco() ⇒ destroiSe().
    aoTocar(alvo) ⇒ tornaSe(TbalaExplodida).
}
```

Assim, o comportamento *movimentoBala* pode fazer parte da descrição de todos os atores que possuírem o mesmo comportamento. Os atores *BalaDeCanhao* e *BalaDeAlien* podem ser redefinidos, utilizando o comportamento *movimentoBala* através das condições originais Os atores pode ser então descritos como:

```
ATOR BalaDeCanhao {
    movimentoBala(Alien).
}

ATOR BalaDeAlien {
    movimentoBala(Canhao).
}
```

Embora a quantidade de linhas na descrição tenha aumentado, a otimização poderia ser mais clara se houvessem mais atores que utilizassem tal comportamento, porém a reutilização de código é definida como um fator predominante na redução da carga de trabalho durante a produção de software (Pianesso, 2002).

Com o intuito de ser o elemento de indicação de que um objeto foi destruído no jogo, a bala explodida é um ator que, ao ser criado permanece no palco por alguns instantes de

tempo, destruindo-se após este intervalo pré-definido. Como o tempo de permanência deve ser curto⁶, define-se a destruição após 2 instantes de tempo.

```
ATOR Bala_Explodida {
    noIntervaloDe(2,2) => destroiSe().
}
```

Os atores que representam os alienígenas no jogo têm comportamentos diferentes em cada fase. O alienígena da primeira fase permanecerá, após criado, andando em uma única direção. Como todo ator, assim que criado, é direcionado para frente (topo do Palco), este alienígena deve girar a 90 graus sentido horário para depois começar a andar. Caso saia do palco, passa a executar o comportamento *consideraPalcoCircular()*, fazendo com que este se desloque novamente ao início de sua trajetória. Se tocar um ator do tipo bala explodida, destrói-se, e a cada 3 instantes de tempo, gera um ator do tipo BalaDeAlien. O ator e alguns comportamentos compostos são definidos abaixo.

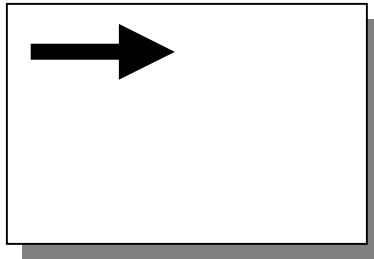
```
COMP destrutivelPorBala() ← {
    aoTocar(BalaExplodida) => destroiSe().
}

COMP disparaBalas() ← {
    aCada(3) => gera(BalaDeAlien).
}

ATOR alienF1 {
    noIntervaloDe(1,1) => gira(90).
    anda(5).
    consideraPalcoCircular().
    destrutivelPorBala().
    disparaBalas().
}
```

⁶ Este tempo curto foi observado durante a fase de análise do jogo em questão, sendo que a permanência de tal ator no palco se deu de forma quase imperceptível.

Figura 12 - Ilustração do movimento do ator alien F1

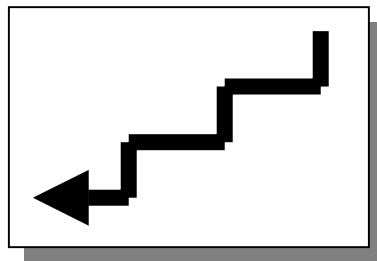


O ator que representa o alienígena da fase dois é em quase tudo semelhante ao ator da fase 1. Porém este ator anda em zigue-zague, movimentando-se no sentido vertical e horizontal, ao invés de andar em linha reta. Este comportamento é conseguido ao se adicionar um comportamento periódico de girar 90 graus a direita ou à esquerda, dependendo do valor do atributo "sentido", que é invertido também a cada giro.

```

ATOR alienF2 {
  noIntervaloDe(1,1) => gira(180); modifica(sentido,1).
  anda(5).
  consideraPalcoCircular().
  destrutivelPorBala().
  disparaBalas().
  aCada(5) => gira (sentido*90); modifica(sentido,-1*sentido).
}
  
```

Figura 13 - Ilustração da movimentação do ator F2



Também tendo um comportamento muito semelhante ao alienígena da fase 01, o ator da fase 03 é diferente apenas na forma de andar no palco. O movimento é na mesma

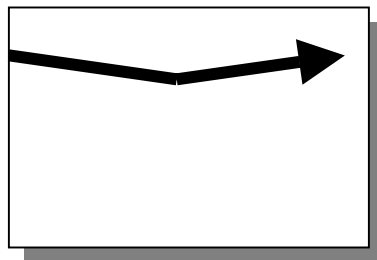
direção porém efetuando um zigue-zague mais “suave” se comparado ao mesmo comportamento do alienígena da fase 02.

```

ATOR alienF3 {
    noIntervaloDe(1,1) => gira(95); modifica(sentido,-10).
    anda(5).
    consideraPalcoCircular().
    destrutivelPorBala().
    disparaBalas().
    aCada(15) => gira(sentido); modifica(sentido,-1*sentido).
}

```

Figura 14 - Ilustração da movimentação do ator alineF3



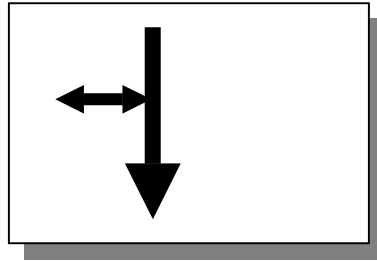
O alienígena da fase 6 possui as mesmas partes comuns a todos apresentados anteriormente, modificando também apenas a forma como se desloca no palco. Tal ator anda em linha reta verticalmente, modificando esporadicamente sua direção, sem que se possa definir, de forma fixa, o momento exato desta mudança de direção. Assim, a condição *comProbabilidade()* foi utilizada neste exemplo, e como sua direção principal é a vertical, periodicamente muda o valor de seu atributo direção.

```

ATOR alienF6 {
    noIntervaloDe(1,1) => gira(180).
    anda(5).
    consideraPalcoCircular().
    destrutivelPorBala().
    disparaBalas().
    aCada(15) => modifica(direcao,180).
    comProbabilidade(0,05) => gira(90).
}

```

Figura 15 - Ilustração da movimentação do ator alienF6



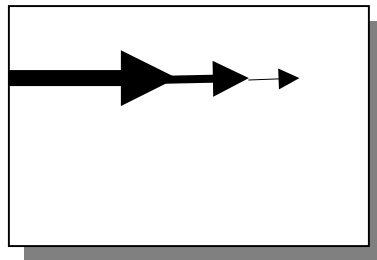
O comportamento dos atores que pertencem à fases futuras se repete aos já citados, porém fazendo uma mudança periódica entre três valores de velocidade predefinidos.

```

ATOR alienMovDinamica {
  noIntervaloDe(1,1) ⇒ gira(90); modifica(velocidade,30).
  aCada(5) ⇒ modifica(velocidade,velocidade-10).
  sempreQue(velocidade,'=',0) ⇒ modifica(velocidade,30).
  anda(velocidade).
  consideraPalcoCircular().
  destrutivelPorBala().
  disparaBalas().
}

```

Figura 16 - Ilustração do movimento do ator alienMovDinamica



5.2 *River Raid*

Criado também na década de oitenta (80) por Carol Shaw, e distribuído pela empresa *Activision*, o *River Raid* é um jogo de ação, onde o principal objetivo do jogador, representado por um avião, é destruir os inimigos através de tiros, sendo que tais inimigos

são representados por objetos com formas de helicópteros, navios e aviões a jato, sobrevoando um imenso rio, controlados pela máquina de forma aleatória (Figura 17).

Figura 17 - Tela inicial do jogo River Raid



Fonte: <http://www.atariage.com/>

O jogo é dividido em fases, sendo que cada uma é separada pela metáfora de uma ponte (

Figura 18), que deve ser destruída pelo jogador, podendo assim mudar de fase.

A área de mobilidade do avião que representa o usuário, bem como dos inimigos do tipo helicóptero e navio são limitados pelo leito do rio e pela metáfora de ilhas colocadas no meio do rio, sendo que sua posição define cada fase. Caso o objeto representado pelo jogador se choque com o leito ou com ilhas se destrói, não acontecendo com os inimigos, que apenas mudam a direção de movimentação, tornando-se a aposta de antes do choque. O objeto inimigo jato não é limitado por essas barreiras. A movimentação permitida ao usuário é horizontal (esquerda e direita), e também aumentar e diminuir a velocidade do

avião, fazendo com que ultrapasse mais rapidamente os obstáculos e inimigos, bem como cada fase.

Figura 18 - Screenshot demonstrando a ponte: passagem de fases



Fonte: <http://www.atariage.com/>

O jogador possui inicialmente três vidas, sendo que estas representam a presença do jogador no *game*. As vidas podem ser perdidas tanto pelo choque no leito do rio, como pelo choque aos inimigos e pontes, ou quando acabar o combustível, utilizado como metáfora ao tempo de vida do jogador, podendo este ser repostado através da passagem do avião jogador sobre objetos que representam postos de gasolina, sendo que os mesmo também podem ser destruídos, através de tiros do jogador.

O jogo conta também com uma marcação de pontos, que são obtidos pela eliminação de inimigos, postos de combustível e pontes, sendo que estes variam de acordo com o tipo de objeto destruído.

5.2.1 Atores e Comportamentos do Jogo

Como no jogo Megamania, o RiverRaid possui um jogador que representa o item de ataque aos inimigos: a Bala. O ator que representa a bala move-se sempre no sentido norte do jogo, iniciando a trajetória na posição de seu gerador. Ao sair do palco a bala se destrói e ao tocar atores do tipo Inimigo, ela se torna uma bala explodida.

```
COMP movimentoBala() ← {
    anda(10).
    foraDoPalco() ⇒ destroiSe().
    aoTocar(Inimigo) ⇒ tornaSe(BalaExplodida).
}
```

O comportamento *movimentoBala* fixa sua velocidade em 10, e indica o que deve ser feito quando estiver fora do palco e quando tocar atores pertencente à classe Inimigo. Assim, o ator bala pode ser definido com o código abaixo.

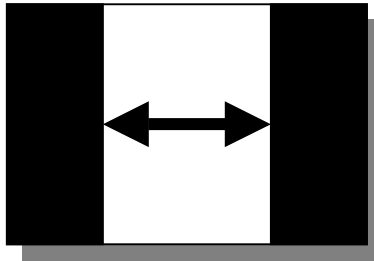
```
ATOR Bala {
    movimentoBala().
}
```

Com o intuito único de servir como elemento de indicação de destruição, a *balaExplodida* é um ator que permanece por um curto espaço de tempo no jogo, utilizada pelos atores *BalaDoAviao* e *BalaDeInimigo*.

```
ATOR BalaExplodida {
    noIntervalo(2,2) ⇒ destroiSe().
}
```

A movimentação dos inimigos do jogador, representados por helicópteros, e feita horizontalmente no palco, sendo que ao atingirem o leito do rio ou objetos que representem ilhas no jogo (pertencente à classe *ObjetosImóveis*), os mesmos mudam sua direção, passando a percorrer o caminho oposto ao que percorriam, representado pelo comportamento *bater*. Ao tocarem o ator do tipo *AviaoJogador*, tornam-se atores do tipo *InimigoExplodido*, que servirá como elemento de destruição do jogador, permanecendo pouco tempo no palco. Ao saírem do palco destroem-se, e são destrutíveis pelo ator *balaExplodida*.

Figura 19 - Ilustração do comportamento movimentoInimigo



```

COMP destrutivelPorBala() ← {
    aoTocar(BalaExplodida) ⇒ destroiSe().
}

COMP movimentoInimigo ← {
    noIntervaloDe(1,1) ⇒ gira(90).
    anda(10).
    destrutivelPorBala().
    bater(ObjetosImoveis).
    aoTocar(AviaoJogador) ⇒ tornaSe(InimigoExplodido).
    foraDoPalco ⇒ destroiSe().
}

ATOR Helicoptero {
    noIntervaloDe(1,1) ⇒ modifica(imagem,'helicoptero.jpg').
    movimentoInimigo().
}

ATOR Navio {
    noIntervaloDe(1,1) ⇒ modifica(imagem,'navio.jpg').
    movimentoInimigo().
}

ATOR InimigoExplodido {
    noIntervaloDe(2,2) ⇒ destroiSe().
}

```

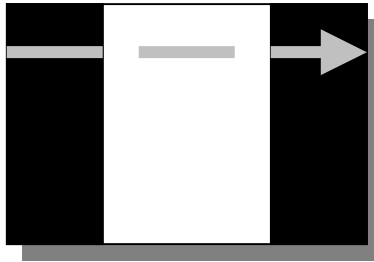
Além dos inimigo descritos acima existe outro que não possui o comportamento acima, pois além do considerar o palco como circular (não é destruído ao sair do palco), também não possui o comportamento *bater* quando toca nos objetos imóveis do palco.

```

ATOR AviaoAJato {
    noIntervaloDe(1,1) => gira(90).
    anda(14).
    destrituivelPorBala().
    consideraPalcoCircular().
    aoTocar(AviaoJogador) => tornaSe(InimigoExplodido).
    foraDoPalco => destroiSe().
}

```

Figura 20 - Ilustração do movimento do ator aviaoAJato



Apesar de ser um objeto imóvel no palco, o ator que representa a ponte no jogo possui certa interação visual com o ambiente, indicando, no momento do toque por um ator do tipo *AviaoDoJogador* sua transformação a um ator *PonteExplodida*, que é mais um dos objetos de destruição do jogo.

```

ATOR Ponte {
    aoTocar(AviaoDoJogador) => tornaSe(PonteExplodida).
}

ATOR PonteExplodida {
    noIntervaloDe(2,2) => destroiSe().
}

```

E o último objeto observado que tem interação direta com o jogador é o posto de combustível, que permanece imóvel no palco porém serve de elemento “recarregador de energia” dos atores da classe *AviaoDoJogador*. Assim, ao tocarem atores do tipo *AviaoDoJogador* o ator que representa o posto de combustível emite um som, e ao tocar atores do tipo *BalaExplodida* se destroem. Não há a necessidade da declaração de um ator

postoExplodido, por exemplo, pois a destruição deste não implica na eliminação de nenhum outro ator componente do palco.

```
ATOR PostoDeCombustivel {  
    aoTocar(AviaoDoJogador) => emitirSom('sompuesto.wav').  
    aoTocar(BalaExplodida) => destroiSe().  
}
```

Assim, tem-se que a descrição dos jogos pode ser feita de maneira descritiva e antropomórfica, cabendo a um compilador verificar a sintaxe e a semântica dos elementos envolvidos na definição de comportamentos e condições. A reutilização dos código também se mostra um fator muito favorável para a construção e novos algoritmos.

6 Conclusões e sugestões de trabalhos futuros

Através da análise das linguagens baseadas no modelo de atores existentes, tem-se que o processo de aprendizado de suas sintaxes e a utilização dos conceitos inseridos nas implementações deste paradigma não são processos triviais, desmotivando assim a utilização do modelo para a construção de programas baseados na orientação a objetos concorrentes. A ferramenta *Mundo dos Atores* (Mariani, 1998) apresenta-se eficaz no ensino de programação e orientação a objetos para alunos aspirantes em computação, porém exige que tais estudantes possuam algum tipo de conhecimento computacional para interagir com a ferramenta.

O processo de observação baseou-se em programas que descrevem jogos pois possuem elementos que interagem de forma visível ao observador (entre si e com o ambiente). Pode-se notar então ações que caracterizam comportamentos, pois repetem-se durante toda a existência do ator (objeto) no jogo. Assim, definiu-se um conjunto de comportamentos tido como básicos para a linguagem proposta, permitindo sua reutilização para a descrição de vários outros ambientes. Esta reutilização pode ser feita pela forma primitiva ou através da construção de novos comportamentos, chamados de compostos.

Todos os comportamentos definidos são tidos como condicionados, ou seja, dependem da realização de alguns eventos pertencentes ao ambiente, como o tempo transcorrido e o toque entre os mesmos; ou por situações criadas por atributos dos próprios atores. Mesmo que não seja declarado durante a construção do comportamento este estará condicionado, pois uma condição nula indica que tal comportamento será *sempre* desempenhado pelo ator. Assim, construiu-se um conjunto de condições básicas para delimitar os papéis dos atores.

Além das condições, a caracterização de cada comportamento para desempenhar os papéis específicos de cada ator, pode ser realizada pela valoração de atributos nativos a qualquer ator criado, ou ainda pela definição de novos atributos.

A sintaxe definida para a linguagem apresenta uma proposta baseada em uma linguagem coloquial, fazendo com que usuários que não estejam inseridos nos conceitos de programação convencionais possam descrever ambientes existentes, devido às características antropomórficas incluídas, embora ainda presos à descrição textual de código.

O desenvolvimento de uma classificação para os comportamentos também foi uma tarefa importante, pois assim pode-se definir atores que executam alguns comportamentos que não são visíveis a todo momento, mas que pertencem ao papel desempenhado pelo ator durante o decorrer do processo, sendo que pode ser evidenciados através de qualquer uma das condições definidas.

Uma restrição imposta pela linguagem é que os atores podem apenas alterar seus próprios atributos, podendo apenas ler as características de outros atores pertencentes ao palco. Algumas situações precisam, portanto de uma solução menos trivial, impondo ao programador uma forma síncrona de ativação de comportamentos. Pode-se citar como exemplo os atores que representam o avião e o posto de combustível no jogo *River Raid*. Assim que o avião toca o posto, o atributo combustível deve ser modificado e ao mesmo tempo, o atributo combustível do próprio posto de gasolina também deve ser alterado, impondo que um dos dois devesse realizar tal alteração no outro. Assim, o comportamento de um dos dois atores estaria parcialmente descrito em outro ator, o que eliminaria a questão atômica da atribuição de comportamentos, uma característica interessante para iniciantes. Pode-se eliminar tal situação fazendo com que ambos tenham o comportamento de modificar seus atributos, condicionados ao toque entre si. Assim, apenas a função que alteração deveria ser diferente (um diminuindo e outro aumentando os valores).

A utilização de tal linguagem pode substituir alguns dos conceitos abordados em classes de ensino de linguagem de programação somente imperativas, e pode auxiliar em projetos que dependam da descrição de comportamentos a objetos existentes, através da conclusão de ferramentas visuais para a construção dos ambientes. Cita-se aqui o projeto *Museu Virtual* (WAZLAWICK et al., 2001).

6.1 Sugestão para trabalhos futuros

De acordo com as limitações impostas a este trabalho, tem-se como propostas para trabalhos futuros os itens a seguir:

- a) Desenvolver um compilador para a linguagem, que utilize componentes texto e gráficos, facilitando ainda mais a descrição de ambientes. Assim a linguagem se tornaria mais antropomórfica, permitindo a exploração dos conceitos por pessoas que não possuem conhecimentos em programação.
- b) Construir uma biblioteca padrão de comportamentos, baseados em mais jogos a serem analisados. Assim a gama de atores e de comportamentos poderia ser aumentada, fazendo com que houvesse maior reaproveitamento de conhecimentos desenvolvidos pela observação.
- c) Descrever uma forma de controle aos atores que interajam diretamente com o usuário do computador, ou seja, que obedecem a comandos de periféricos, fazendo com que seja possível a descrição de tais comportamentos, e construir uma caracterização mais fiel de ambientes observados.
- d) Avaliar a viabilidade real da linguagem proposta através da experimentação em ambientes de ensino a leigo em programação de computadores, através de métodos de pedagogia.

7 Referências Bibliográficas

(AGHA,1986) AGHA, Gul. **An Overveiw of Actor Languages**. MIT: The Artificial Intelligence Laboratory. Massachusetts: June, 1986.

(AGHA, 1986) AGHA, Gul. **Actors: A Model of Concurrent Computation in Distribuided System**. Cambridge: MIT Press, 1986.

(AGHA, 1997) AGHA, Gul, MASON, I.A., Smith, S.F. e TALCOTT, . C. L. **A foundation for actor computation**. Journal of Functional Programming, 7:1-72, 1997.

(BOOCH, 2000) BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML, guia do usuário**. Tradução de Fábio Freitas da Silva. Rio de Janeiro: Campus, 2000.

(BRIOT, 1997) BRIOT, Jean-Pierre. **Actalk: A Framework for Object-Oriented Concurrent Programming – Design and Experience**. Université Paris 6 – CNRS. 1997.

(BRIOT, 1989) BRIOT, Jean-Pierre. **Actalk: a Testbed for Classifuing and Designing Actor Languages in the Smalltalk-80 Environment**. Proceedings of European Conference on Object-Oriented Programming, British Computer Society Workshop Series: Cambridge. July 1989.

(FURLAN, 1998) FURLAN, José Davi. **Modelagem de Objetos através da UML – the Unified Modeling Language**. São Paulo: Makron Books, 1998.

(GUY, 1993) GUY, L., STEELE, Jr. & GABRIEL, Richard P. **The Evolution of Lisp**. ACM SIGPLAN Notices, vol 28, núm. 3 – Março – 1993, pp 231 – 270. <http://citeseer.nj.nec.com/steele93evolution.html>

(LARMAN, 2000) LARMAN, Carig. **Utilizando UML e Padrões: Uma introdução à análise e ao projeto Orientados a Objetos**. Tradução Luiz^a Meirelles Salgado. Porto Alegre: Bookman, 2000.

(LIEBERMAN, 1987) LIEBERMAN, Henry. **Concurrent Object-Oriented Programming in Act 1**. MIT Artificial Intelligence Laboratory, Cambridge USA, 1987. <http://agents.www.media.mit.edu/people/lieber/Lieberary/OOP/Act-1/Concurrent-OOP-in-Act-1.html>

(LIU, 2002) LIU, Jie & LEE, Edward A. **Timed Multitasking for Real-Time Embedded Software**. Invited paper in *IEEE Control System Magazine*, special issue on "Advances in Software Enabled Control", to appear in December, 2002.

(MARIANI, 1998) MARIANI, Antonio Carlos. **O Mundo dos Atores**. Universidade Federal de Santa Catarina, Depto. de Informática e Estatística. Florianópolis, 2001. <http://www.inf.ufsc.br/poo/atores>.

(MARIANI, 1998) MARIANI, Antonio Carlos. **O Mundo dos Atores: uma perspectiva de introdução à programação orientada a objetos**. Anais do Simpósio Brasileiro de Informática na Educação (SBIE). Fortaleza, Ceará.

(HENESSY, 1997) HENNESSY, John L & PATTERSON, David A. **Computer Organization and Design: the hardware/software interface**. 2nd edition. São Francisco, CA: Morgan Kaufman Publishers, 1997.

(PIANESSO, 2002) PIANESSO, Ana Cláudia Fiorin. **Identificação e Classificação de Comportamentos de Objetos Dinâmicos**. Dissertação de Mestrado. Florianópolis: UFSC, 2002.

(PHILIPPSEN, 1995) PHILIPPSEN, Michael. **Imperative Concurrent Object-Oriented Languages: An Annotated Bibliography**. Berkeley, California: International Computer Science Institute, 1995. <ftp://ftp.cs.york.ac.uk/reports/YCS-94-220.ps.Z>

(SEBESTA, 2000) SEBESTA, Robert W. **Conceitos de Linguagens de Programação**. 4ª edição. Porto Alegre: Bookman, 2000.

(SHINOUDA, 2002) SHINOUDA, Maher. **Actor Languages and Component-Based Design**. Canada: University of Waterloo, 2002. www.cs.uwaterloo.ca/~mshinouda/Actor.pdf

(SOUZA, 1997) SOUZA, Patrícia C.; WAZLAWICK, Raul S. An Authoring System for the Creation of Educational Adventures in Virtual Reality. Revista GRAPH&TEC 1(2):39-53. Editora da UFSC. Julho, 1997. ISSN 1413-6481

(TANENBAUM, 1992) TANENBAUM, Andrew S.; COSTA, Luiz Fernando; MARQUES SOBRINHO, Hélio. **Organização estruturada de computadores**. Rio de Janeiro: Prentice-Hall, 1992.

(VARELA, 2001) VARELA, Carlos. **Worldwide Computing with Universal Actors**: Linguistic Abstractions for Naming, Migration, and Coonlination. Tese de Doutorado, Universidade de Illinois, Abril 2001. <http://osl.cs.uiuc.edu/Theses/varela-phd.pdf>.

(VARELA, 1999) VARELA, C. & AGHA, G. **Linguistic Supports for Actors, First-Class Token-Passing Continuations and Join Continuations**. Proceedings of the Midwest Society for Programming Languages and Systems Workshop, October, 1999. <http://osl.cs.uiuc.edu/~cvarela/mspls99/>

VARELA, C. & AGHA, G. **Programming Dynamically Reconfigurable Open Systems with SALSA**. <http://osl.cs.uiuc.edu/salsa/>

RUMBAUGH, James et all. **Object-Oriented Modeling and Design**. 1ª edição. Prentice Hall.

WAZLAWICK, Raul S. et all. **Providing More Interactivity to Virtual Museums**: A Proposal for a VR Authoring Tool. Presence Teleoperators And Virtual Environments, MIT Press, Cambridge, USA. 10(6):647-656. December.

Screen-shots do atari. <http://www.atariage.com/>

Screen-shots do atari. <http://www.gamescreenshots.com/>