UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

EXTENSÃO DA LINGUAGEM LARF PARA A COMUNICAÇÃO ENTRE ROBÔS JOGADORES DE FUTEBOL

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

FRANCIS MILTON JACOBSEN

BLUMENAU, JUNHO/2003

2003/1-27

EXTENSÃO DA LINGUAGEM LARF PARA A COMUNICAÇÃO ENTRE ROBÔS JOGADORES DE FUTEBOL

FRANCIS MILTON JACOBSEN

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Jomi Fred Hübner — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Jomi Fred Hübner

_____

Prof. Paulo Cezar Rodacki Gomes

_____

Prof. Marcel Hugo

Aos meus pais.

AGRADECIMENTOS

Agradeço aos meus colegas e professores que compartilharam conhecimento e companheirismo durante todos esses anos do curso.

Agradeço também aos meus amigos que estiveram presentes nos momentos mais difíceis me apoiando e incentivando, não só durante o curso, mas durante toda a minha vida.

Devo manifestar gratidão ao meu orientador que durante o período de elaboração deste trabalho foi extremamente paciente e esclarecedor quanto as minhas inúmeras dúvidas, sendo conseqüentemente de essencial importância para a conclusão do mesmo.

O meu agradecimento especial vai para as pessoas mais importantes da minha vida que são meus pais, os quais me possibilitaram ingressar no extraordinário mundo acadêmico, revolucionando minha vida para sempre.

"Parece que estamos a ponto de desenvolver grande variedade de máquinas inteligentes capazes de efetuar tarefas muito perigosas, muito onerosas ou muito aborrecidas para os seres humanos. [...] O principal obstáculo [ao desenvolvimento destas máquinas] parece ser um problema muito humano - o sentimento secreto, que se insinua furtivo e espontâneo, de que há algo ameaçador e inumano no fato de as máquinas realizarem certas tarefas melhor do que nós; ou uma sensação de repugnância ante criaturas feitas de silício e germânio, ao invés de proteínas e ácidos nucléicos. Em muitos sentidos, porém, nossa sobrevivência como espécie depende de superarmos estes chauvinismos primitivos. [...] Não há nada de inumano nas máquinas inteligentes; na realidade, elas são a expressão das incríveis capacidades intelectuais que só os seres humanos, dentre todas as criaturas deste planeta, possuem no momento".

- Carl Sagan

RESUMO

Este trabalho apresenta a extensão da Linguagem LARF adicionando a esta comandos para utilização de variáveis e comunicação entre os robôs jogadores de futebol. É utilizada como ferramenta de comunicação entre os agentes o *Simple Agent Communication Infrastructure* (SACI), que utiliza a *Knowledge Query Manipulation Language* (KQML) como linguagem e protocolo para troca de informações entre os agentes. As regras de produção da linguagem foram definidas utilizando a metalinguagem Bakus Naur form (BNF) e como gerador de *parser* o JavaCC.

ABSTRACT

This work presents the LARF language extension adding commands for use of variables and communication among the robots soccer players. It is used Simple Agent Communication Infrastructure (SACI) as the communication tool. SACI uses Knowledge Query Manipulation Language (KQML) as language and protocol for exchange of information among the agents. The production rules of the language were defined using the Bakus Naur Form (BNF) meta-language, and JavaCC as parser generator.

LISTA DE FIGURAS

Figura 1 - Tela do simulador *TBSim* 21

Figura 2 – Modelo geral de agente 24

Figura 3 – Ciclo de vida do agente 27

Figura 4 – Serviço de páginas brancas 28

Figura 5 – Serviço de páginas amarelas 29

Figura 6 – Dimensão do campo 36

Figura 7 – Área de atuação 37

Figura 8 – Interfaces da modelagem 39

Figura 9 – Diagrama das classes da área de atuação 40

Figura 10 – Diagrama da classe *RecebeRobo* 40

Figura 11 – Diagrama das classes de ação 41

Figura 12 – Diagrama da classe *Se* 42

Figura 13 – Controle de comportamento ativo 42

Figura 14 – Diagrama da classe *Rotina* 43

Figura 15 – Diagramas das expressões relacionais 44

Figura 16 – Compilação de um programa 32

Figura 17 – Visão geral da linguagem LARF 45

Figura 18 – Árvore de objetos 46

Figura 19 – Arquitetura do *AgenteJogador* 47

Figura 20 – Visão geral do protótipo 53

Figura 21 – Interface *Expressao* 54

Figura 22 – Diagrama da classe *Atribuicao* 55

Figura 23 – Diagramas das classes aritméticas 57

Figura 24 – Diagrama da classe *Imprimir* 59
Figura 25 – Diagrama da classe *CriaMsg* 60
Figura 26 – Diagrama da classe *GetCampoMsg* 62
Figura 27 – Diagrama da classe *SetCampoMsg* 63
Figura 28 – Diagrama da classe *EnviarMsg* 65
Figura 29 – Diagrama da classe *RecebeMsg* 66
Figura 30 – Execução do compilador 68
Figura 31 – Execução da implementação 69
Figura 32 – Log de execução do SACI 70

LISTA DE TABELAS

Tabela 1 – Símbolos da metalinguagem BNF 33

Tabela 2 – Símbolos terminais da linguagem 50

Tabela 3 – BNF da linguagem LARF 51

Tabela 4 – Símbolos terminais adicionados na linguagem 54

Tabela 5 – BNF do comando de atribuição 55

Tabela 6 – BNF das operações aritméticas 57

Tabela 7 – BNF do comando *Imprimir* 59

Tabela 8 – BNF do comando *CriaMsg* 60

Tabela 9 – BNF do comando *GetCampoMsg* 62

Tabela 10 – BNF do comando *SetCampoMsg* 63

Tabela 11 – BNF do comando *EnviaMsg* 65

Tabela 12 – BNF do comando *ReceberMsg* 66

LISTA DE QUADROS

Quadro 1 – Arquivo de descrição do ambiente 21

Quadro 2 – Formato de mensagem KQML 25

Quadro 3 – Anúncio de habilidade 29

Quadro 4 – Solicitação da lista de agentes 30

Quadro 5 – Resposta do facilitador 30

Quadro 6 – Utilizando o SACI para desenvolver agentes 31

Quadro 7 – Exemplo da seção controle principal 38

Quadro 8 – Exemplo da seção comportamentos 38

Quadro 9 – Exemplo da seção rotinas genéricas 39

Quadro 10 – Exemplo de declarações da linguagem 46

Quadro 11 – Exemplo de execução do *AgenteJogador* 47

Quadro 12 – Exemplo de implementação em LARF 48

Quadro 13 – Ações semânticas do comando de atribuição 55

Quadro 14 – exemplo de atribuicao em LARF 55

Quadro 15 – Ações semânticas das operações aritméticas 58

Quadro 16 – Exemplo das operações aritméticas em LARF 57

Quadro 17 – Ações semânticas do comando *Imprimir* 59

Quadro 18 – Exemplo do comando *Imprimir* 59

Quadro 19 – Ações semânticas do comando *CriaMsg* 61

Quadro 20 – exemplo do comando *CriaMsg* 60

Quadro 21 – Ações semânticas do comando *GetCampoMsg* 62

Quadro 22 – exemplo do comando GetCampoMsg 62

Quadro 23 – Ações semânticas do comando *SetCampoMsg* 64

Quadro 24 – exemplo do comando *SetCampoMsg*.....63

Quadro 25 – Ações semânticas do comando *EnviaMsg*.....65

Quadro 26 – Exemplo do comando *EnviaMsg*.....65

Quadro 27 – Ações semânticas do comando *RecebeMsg*.....67

Quadro 28 – exemplo do comando RecebeMsg.....66

Quadro 29 – Implementação da classe *Somar*.....58

Quadro 30 – Implementação da classe *Imprimir*.....60

Quadro 31 – Implementação da classe *Atribuicao*.....56

Quadro 32 – Implementação da classe *CriaMsg*.....61

Quadro 33 – Implementação da classe *GetCampoMsg*.....63

Quadro 34 – Implementação da classe *SetCampoMsg*.....64

Quadro 35 – Implementação da classe *EnviaMsg*.....66

Quadro 36 – Implementação da classe *RecebeMsg*.....67

Quadro 37 – Exemplo de implementação da comunicação em LARF.....67

LISTA DE ABREVIATURAS E SIGLAS

API *Application Program Interface*

BNF *Backus-Naur Form*

FIPA *Foundation for Intelligent Physical Agents*

IA Inteligência Artificial

IAD Inteligência Artificial Distribuída

ICMAS *International Conference on Multi-Agent Systems*

IJCAI *International Joint Conference on Artificial Intelligence*

JavaCC *Java Compiler Compiler*

KQML *Knowledge Query and Manipulation Language*

LARF Linguagem para Definição de Agentes Robôs Jogadores de Futebol

LCA Linguagem de Comunicação entre Agentes

RoboCup *Robot World Cup*

SACI *Simple Agent Communication Infrastructure*

SMA Sistemas Multiagentes

TBSim *TeamBot Simulator*

SUMÁRIO

1 INTRODUÇÃO 16

1.1 OBJETIVOS DO TRABALHO 17

1.2 ESTRUTURA DO TRABALHO 18

2 ROBOCUP 19

2.1 TEAMBOTS 20

2.2 TBSIM 20

3 SISTEMAS MULTIAGENTES 23

3.1 COMUNICAÇÃO ENTRE AGENTES 24

3.2 KQML 25

3.3 SACI 26

3.3.1 ENVIANDO E RECEBENDO MENSAGENS 28

4 LINGUAGEM LARF 36

4.1 VISÃO GERAL DA LINGUAGEM 36

4.1.1 DEFINIÇÃO DO CAMPO 36

4.1.2 DEFINIÇÃO DO JOGADOR 37

4.1.2.1 ÁREA DE ATUAÇÃO 37

4.1.2.2 CONTROLE PRINCIPAL 38

4.1.2.3 COMPORTAMENTOS 38

4.1.3 ROTINAS GENÉRICAS 39

4.1.4 ESPECIFICAÇÃO DA LINGUAGEM LARF 39

4.2 COMPILADORES 32

4.2.1 BACKUS-NAUR FORM (BNF) 33

4.2.2 JAVACC 34

4.3 COMPILADOR DA LINGUAGEM LARF 44
4.3.1 BNF DA LINGUAGEM LARF 50
5 VISÃO GERAL DO PROTÓTIPO 53
5.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 53
5.2 ESPECIFICAÇÃO 54
5.2.1 ATRIBUIÇÃO 54
5.2.2 EXPRESSÕES ARITMÉTICAS 56
5.2.3 IMPRIMIR 58
5.2.4 CRIA MENSAGEM 60
5.2.5 GET CAMPO MENSAGEM 61
5.2.6 SET CAMPO MENSAGEM 63
5.2.7 ENVIA MENSAGEM 65
5.2.8 RECEBE MENSAGEM 66
5.3 IMPLEMENTAÇÃO ..... **ERRO! INDICADOR NÃO DEFINIDO.**
5.4 TÉCNICAS E FERRAMENTAS UTILIZADAS 67
5.5 OPERACIONALIDADE DA IMPLEMENTAÇÃO 67
5.6 PROBLEMAS E DIFICULDADES 71
5.7 RESULTADOS E DISCUSSÃO 71
6 CONCLUSÕES 72
6.1 EXTENSÕES 72
REFERÊNCIAS BIBLIOGRÁFICAS 74

1 INTRODUÇÃO

O futebol de robôs, afirma David (2001), consiste de campeonatos de times de robôs móveis, cooperando com um objetivo definido (fazer gols), contra um time adversário, sem interferência humana.

Contudo o objetivo real das partidas de futebol de robôs realizadas por todo o mundo é o desenvolvimento e aprimoramento de sistemas de inteligência artificial para colaboração entre máquinas inteligentes, sem a interferência humana. O futebol de robôs incentiva este desenvolvimento sob a forma de competição, onde cada país poderá chegar à copa mundial, com um time melhor a cada ano.

A *RoboCup* possui cinco categorias: quatro delas envolvem disputas entre times de robôs reais, pequenos (*small size league*), médios (*middle size league*), robôs que possuem quatro pernas (*Sony four-legged)*, onde se utilizam robôs Aibo desenvolvidos pela Sony, (*humanoid)* humanóides e uma quinta envolve partidas disputadas em um simulador. Esta última categoria permite que grupos de pesquisadores em IA desenvolvam times através da implementação de agentes computacionais autônomos capazes de cooperar para disputar uma partida de futebol de robôs, sem se preocupar com a parte física da construção de robôs.

Conforme Balch (2000) *TeamBots* é um conjunto de programas e pacotes Java para pesquisadores em robótica móvel na área de Sistemas Multi-Agentes. *TeamBots* é distribuído com o seu código fonte aberto. O ambiente de simulação é totalmente escrito em Java. *TeamBots* suporta prototipação, simulação e execução de sistemas que controlam sistemas de múltiplos robôs. Sistemas para controle de robôs desenvolvidos com o *TeamBots* podem ser executados no programa simulador *TBSim*. *TBSim* faz parte do pacote de programas do ambiente *TeamBots. TBSim* é um programa que tem por objetivo realizar a simulação das condições encontradas no mundo real (obstáculos, outros robôs, bola de golfe, tamanho da área de atuação, etc.) para robôs da categoria de médio porte utilizada nas competições da *RoboCup*.

Em Schlei (2002), iniciou-se o desenvolvimento de uma linguagem para definição de estratégias de controle de times de robôs jogadores de futebol baseado no ambiente *TBSim*. Naquele trabalho é apresentado o desenvolvimento de uma linguagem declarativa para a construção de times de robôs formada por agentes distribuídos. Mais especificamente, aquele trabalho procura apresentar as características mais relevantes no desenvolvimento desta lingua-

gem, concentrando-se principalmente na sua especificação e implementação para tornar a interpretação da linguagem pelo agente a mais natural possível.

Tratando-se de times de robôs formados por agentes distribuídos, não é difícil perceber a necessidade da utilização de um Sistema Multiagente (SMA), pois é através das técnicas de SMA que permite aos agentes atuarem de forma cooperativa conforme necessita uma partida de futebol.

As duas principais características de um agente num SMA são perceber e agir. A habilidade de comunicar-se faz parte da percepção (quando recebe mensagens) e da ação (quando envia mensagens), assim a comunicação entre agentes está baseada na troca de mensagens (WOOLDRIDGE, 2002). No futebol de robôs, um agente poderá, por exemplo, avisar um outro que este está com a bola, ou que está indo para o ataque e que outro agente deve ficar no lugar dele na defesa. A forma como estas mensagens são trocadas entre os agentes são definidas por uma Linguagem de Comunicação entre Agentes (LCA).

Uma ferramenta que permite a implementação da comunicação entre agentes é a *Simple Agent Communication Infrastructure* (SACI), que segundo Hübner (2001), é uma ferramenta que torna a programação da comunicação entre agentes distribuídos mais fácil, em conformidade com um padrão, rápida e robusta.

O objetivo desse trabalho é estender a Linguagem para Definição de Agentes Robôs Jogadores de Futebol (LARF), apresentada por Schlei (2002), implementando a comunicação entre os robôs utilizando a ferramenta SACI para programação da comunicação entre os agentes, para facilitar a criação de estratégias ou jogadas ensaiadas para os times.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho de conclusão de curso é especificar e implementar comandos para a comunicação entre os robôs jogadores de futebol na linguagem de definição de comportamentos dos jogadores proposta por Schlei (2002).

Os objetivos específicos do trabalho são:

a) identificar as necessidades de comunicação entre robôs jogadores de futebol para que possam cooperar mais facilmente;

b) estender a especificação da linguagem de definição de jogadores com coman-

dos de comunicação utilizando a ferramenta de programação de comunicação entre agentes SACI;

c) implementar a extensão da linguagem LARF;

d) definir e implementar um time que execute uma jogada ensaiada utilizando a comunicação para se coordenarem.

1.2 ESTRUTURA DO TRABALHO

Conforme os objetivos apresentados neste capítulo, o capítulo dois aborda a *RoboCup*. Na primeira seção daquele capítulo expõe-se sobre o *TeamBots*, e a segunda seção apresenta o simulador *TBSim*.

O capítulo três aborda a área de Sistemas Multiagentes (SMA). A primeira parte desta seção refere-se a comunicação entre os agentes. Na segunda parte trata-se da linguagem e protocolo de comunicação KQML. E a terceira parte do capítulo possui informações sobre a ferramenta de programação de comunicação de agentes SACI.

O quarto capítulo apresenta informações sobre a linguagem LARF. A segunda parte deste capítulo é referente a compiladores, a metalinguagem BNF e o gerador de *parser* JavaCC. Na terceira seção é abordado o compilador da linguagem LARF.

O quinto capítulo descreve o desenvolvimento do protótipo. Na primeira seção deste capítulo, são abordados os requisitos principais do problema a ser trabalhado, a segunda parte apresenta a especificação do protótipo, a terceira seção possui a implementação da extensão, a quarta parte aborda as técnicas e ferramentas utilizadas, a quinta seção apresenta a operacionalidade da implementação, na sexta parte são explanados os problemas e dificuldades, e a sétima seção aborda os resultados e discussão do trabalho.

O capítulo seis apresenta as conclusões do trabalho, e na primeira e única seção deste capítulo possui sugestões de extensões para trabalhos futuros.

2 ROBOCUP

A *RoboCup* é um encontro científico aberto ao grande público, na forma de várias competições de robôs e de uma conferência, promovido pela *RoboCup Federation*, instituição privada com sede na Suíça que reúne pesquisadores de todo o mundo, visando fomentar a pesquisa em Inteligência Artificial (IA), e Robótica, com o objetivo emblemático de construir até 2050 uma equipe de humanóides capaz de derrotar o campeão mundial de futebol humano. Conforme LCMI (2000), a *RoboCup*, propõe um problema padrão a ser solucionado: uma partida de futebol de robôs. Esta iniciativa permite que diversas técnicas destas áreas sejam testadas e, principalmente, comparadas. A construção de um time de futebol de robôs envolve a integração de diversas tecnologias, tais como: projeto de agentes autônomos, cooperação em sistemas multiagentes, estratégias de aquisição de conhecimento, engenharia de sistemas de tempo real, sistemas distribuídos, reconhecimento de padrões, aprendizagem, controle de processos, etc.

A *RoboCup* possui cinco categorias: quatro delas envolvem disputas entre times de robôs reais, pequenos (*small size league*), médios (*middle size league*), robôs que possuem quatro pernas (*Sony four-legged)*, onde se utilizam robôs Aibo desenvolvidos pela Sony, (*humanoid)* humanóides e uma quinta envolve partidas disputadas em um simulador. Esta última categoria permite que grupos de pesquisadores em IA desenvolvam times através da implementação de agentes computacionais autônomos capazes de cooperar para disputar uma partida de futebol de robôs, sem se preocupar com a parte física da construção de robôs.

A primeira "*Robot World Cup*" aconteceu em agosto de 1997, em Nagoya, Japão, durante a IJCAI'97 (*Fifteenth International Joint Conference on Artificial Intelligence*) e contou com a participação de pelo menos 40 times. A segunda edição aconteceu de 2 a 5 de Julho de 1998 em Paris, França, em conjunto com o ICMAS'98 (*International Conference on Multi-Agent Systems*). Em 1999 a terceira edição da *RoboCup* aconteceu durante o IJCAI'99 em Estocolmo, Suécia. Em Melbourne, na Austrália, ocorreu a quarta edição da *RoboCup* no ano 2000; a quinta foi em Seattle, em 2001 nos Estados Unidos; a sexta aconteceu em Fukuoka, no Japão em 2002 e a sétima ocorrerá em Pádua na Itália.

A *RoboCup* acontece anualmente e as competições são sempre em conjunto com congressos científicos internacionais de reconhecimento mundial, tais como IJCAI e ICMAS. Durante as competições é realizado também um *workshop* onde são apresentados os recursos

tecnológicos utilizados pelos participantes e os resultados das respectivas pesquisas. Maiores informações sobre *RoboCup* podem ser encontradas no site oficial da *RoboCup*, (http://www.robocup.org).

2.1 TEAMBOTS

TeamBots é um conjunto de programas e pacotes Java para pesquisadores em robótica móvel na área de SMA. *TeamBots* é distribuído com o seu código fonte aberto. Atualmente os robôs desenvolvidos no *TeamBots* podem ser executados nos robôs que utilizam a tecnologia Nomadic, robô Nomad 150 (BALCH, 2000).

Uma das mais importantes características do ambiente *TeamBots* é o suporte à protipação e simulação do mesmo sistema de controle que é executado em robôs móveis. O ambiente *TeamBots* é extremamente flexível. Ele suporta a execução de múltiplos robôs heterogêneos com sistemas de controles heterogêneos. Ambientes experimentais complexos (ou simples) podem ser criados com paredes, estradas, outros robôs e obstáculos circulares. Todos esses objetos podem ser criados editando-se um arquivo de configuração.

Quatro programas fazem parte da distribuição do *TeamBots*. São eles: o *TBSim*, que é o simulador do *TeamBots*. Usado para testar sistemas de controle de robôs que utilizam a API *abstractrobots*. O *TBHard*, sendo este o ambiente de execução de robôs reais para sistemas de controle desenvolvidos utilizando o *TeamBots*. O *RoboComm* que simplifica a comunicação assíncrona de robô para robô, onde qualquer objeto Java pode ser enviado de um robô para outro. E o *JCye*, para controlar robôs do tipo *Cye*. No entanto neste trabalho é utilizado o *TBSim* para simular partidas de futebol.

2.2 TBSIM

TBSim faz parte do pacote de programas do ambiente *TeamBots*. *TBSim* é um programa que tem por objetivo realizar a simulação das condições encontradas no mundo real (obstáculos, outros robôs, bola de golfe, tamanho da área de atuação, etc.) para a categoria de robôs simulados nas competições da *RoboCup*. A interface gráfica do simulador é apresentada na figura 1.

Figura 1 - Tela do simulador *TBSim*

Fonte: *TeamBots*.

Para executar o *TBSim* em modo gráfico é preciso escrever a linha de comando "`java TBSim.TBSim robocup.dsc 511 300`" que passa para o *TBSim* o arquivo robocup.dsc que possui a descrição do ambiente e informa que vai desenhar o ambiente descrito numa área de 511x300 *pixels*. O *TBSim* também pode ser executado sem a utilização do modo gráfico usando a seguinte linha de comando: "`java TBSimNoGraphics robocup.dsc`". Este modo é utilizado para verificar mais rapidamente o resultado dos comportamentos criados para os robôs.

O arquivo de descrição do ambiente contém a descrição do ambiente no qual os robôs vão atuar. No quadro 1 é apresentado um exemplo comentado deste arquivo.

Quadro 1 – Arquivo de descrição do ambiente

```
// Esse arquivo de descrição configura o ambiente para simular um jogo
//de futebol da RoboCup no simulador JavaBotSim.

// Limites do campo.     esquerda     direita     abaixo     acima
             bounds         -1.47        1.47     -.8625     .8625

// Número da posição do jogador.
             seed 3

// Taxa de aceleração do tempo da partida.
             time 2.0 // acelera a partida 2 vezes em relação ao tempo real.

// Tempo máximo da partida em milisegundos.
             timeout 600000 // dez segundos
```

```
// Define o tempo máximo que pode decorrer entre duas simulações.
             maxtimestep 50 // 1/10 de segundo.

// Cor de fundo
             background x009000 // verde escuro

// Cria um objeto e define suas características
// object <objecttype> <x> <y> <theta> <forecolor> <backcolor> <visionclass>

// SocFieldSmallSim é um objeto especial que desenha o campo. Não interage com os
//robôs ou com a bola.

// Exemplo da criação do campo de futebol
object EDU.gatech.cc.is.simulation.SocFieldSmallSim 0 0 0 0 x009000 x000000 0

// A bola
object EDU.gatech.cc.is.simulation.GolfBallNoiseSim 0 0 0 0.02
       xF0B000 x000000 3

// Cria um robô e define suas características.
// robot robottype controlsystem x y theta forecolor backcolor visionclass

//======WEST TEAM======
//robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG
//------------seu sistema de controle de nomes vai aqui ^^^^^^^^
//     -1.2  0    0 xEAEA00 xFFFFFF 1
//robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG
//------------seu sistema de controle de nomes vai aqui ^^^^^^^^
//     -.5   0    0 xEAEA00 xFFFFFF 1
//robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG
//------------seu sistema de controle de nomes vai aqui ^^^^^^^^
//     -.15  .5   0 xEAEA00 xFFFFFF 1
//robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG
//------------seu sistema de controle de nomes vai aqui ^^^^^^^^
//     -.15  0    0 xEAEA00 xFFFFFF 1
//robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AIKHomoG
//------------seu sistema de controle de nomes vai aqui ^^^^^^^^
//     -.15  -.5  0 xEAEA00 xFFFFFF 1

//======EAST TEAM======
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador
       1.2  0    0 xFF0000 x0000FF 2
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador
       .5   0    0 xFF0000 x0000FF 2
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador
       .15  .5   0 xFF0000 x0000FF 2
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador
       .15  0    0 xFF0000 x0000FF 2
robot EDU.gatech.cc.is.abstractrobot.SocSmallSim AgenteJogador
       .15  -.5  0 xFF0000 x0000FF 2
```

Fonte: adaptado de *TeamBots*.

Maiores informações podem ser encontradas no site oficial do *TeamBots* (www. teambots.org).

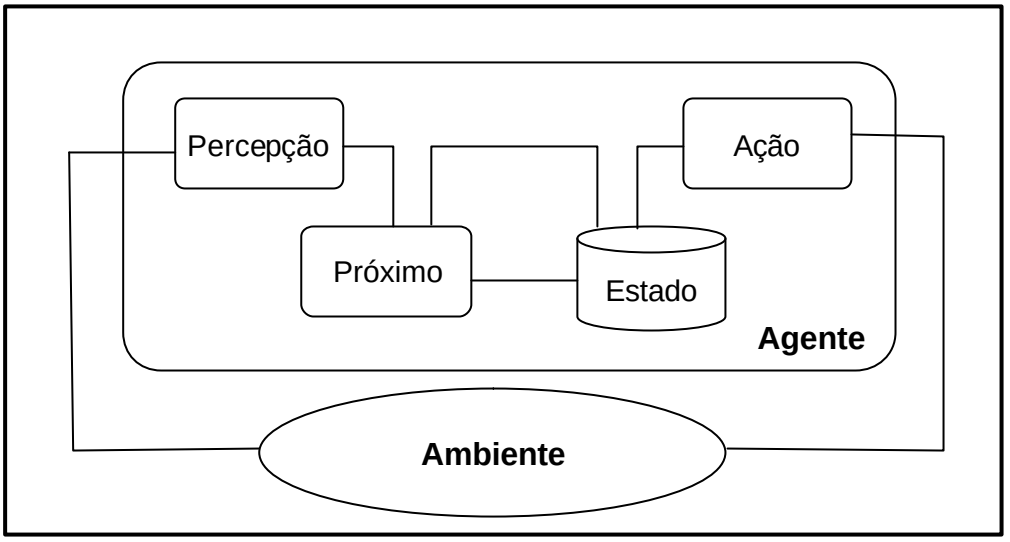
3 SISTEMAS MULTIAGENTES

O termo “Sistemas Multiagentes” está sendo bastante utilizado atualmente e tem sido aplicado por qualquer sistema que seja composto por múltiplos agentes interagindo (MENESES, 2001). Os Sistemas Multiagentes (SMA) formam uma área de pesquisa dentro da Inteligência Artificial Distribuída (IAD), que se preocupa com todos aspectos relativos à computação distribuída em sistemas de inteligência artificial (BORDINI, 2002, p. 01).

Não há uma definição universal aceita de agentes. Essencialmente, há um consenso que *autonomia* é o centro da noção de agentes. A dificuldade está na importância que diferentes características têm em diferentes domínios de aplicação. Por exemplo, em algumas aplicações o agente ser capaz de aprender é imprescindível e em outras não é importante, às vezes até indesejável. No entanto, Weiss (1999) diz que um agente é um sistema de computador que está situado em algum ambiente, a fim de satisfazer seus objetivos de projeto. Estar situado em um ambiente significa que o agente recebe entrada através de sensores do ambiente e que executa ações que mudam o ambiente de alguma forma.

Bordini (2002) afirma que o enfoque principal é prover mecanismos para a criação de sistemas computacionais a partir de entidades de software autônomas, denominadas agentes, que interagem através de um ambiente compartilhado por todos os agentes de uma sociedade, e sobre o qual estes agentes atuam, alterando o seu estado. Com isto, quer-se dizer que é preciso prover mecanismos para a interação e coordenação destas entidades, já que cada uma possui um conjunto de capacidades específicas, bem como possuem seus próprios objetivos em relação aos estados do ambiente que querem atingir, exatamente porque cada agente possui um conjunto específico e limitado de capacidades. Freqüentemente os agentes precisam interagir para atingirem seus objetivos. Desta forma, é possível, para os projetistas de sistemas computacionais, a criação de sistemas complexos de forma naturalmente distribuída e *bottom-up*. Contudo, criar mecanismos genéricos para a coordenação de tais agentes para que o sistema como um todo (em geral chamado de uma sociedade de agentes) funcione de forma adequada e eficiente é um dos grandes desafios. Outro grande desafio é a especificação interna de um agente, em que tipicamente se deseja uma representação simbólica daquilo que o agente sabe sobre o ambiente (e sobre os outros agentes naquele ambiente), bem como daquilo que o agente pretende atingir. Na figura 2 é apresentado um modelo geral de agente.

Figura 2 – Modelo geral de agente



Fonte: Adaptado de Wooldridge (1999).

Como não existe um problema definido a priori que o sistema deve resolver, o objetivo da área é estudar modelos genéricos a partir dos quais podem-se conceber agentes, organizações e interações, de modo a poder instanciar tais conceitos quando se deseja resolver um problema particular. Dito de um outro modo, o objetivo é conceber os meios a partir dos quais pode-se assegurar que agentes desejem cooperar e efetivamente o façam, com o intuito de resolver um problema específico quando este for apresentado ao sistema.

3.1 COMUNICAÇÃO ENTRE AGENTES

Comunicação é o processo pelo qual a informação é trocada entre agentes. Mensagens transmitidas possuem conteúdo e contexto. Conteúdo refere-se aos dados codificados na mensagem; contexto é como as propriedades mudam desde os dados à informação. A comunicação entre agentes é baseada em mensagens.

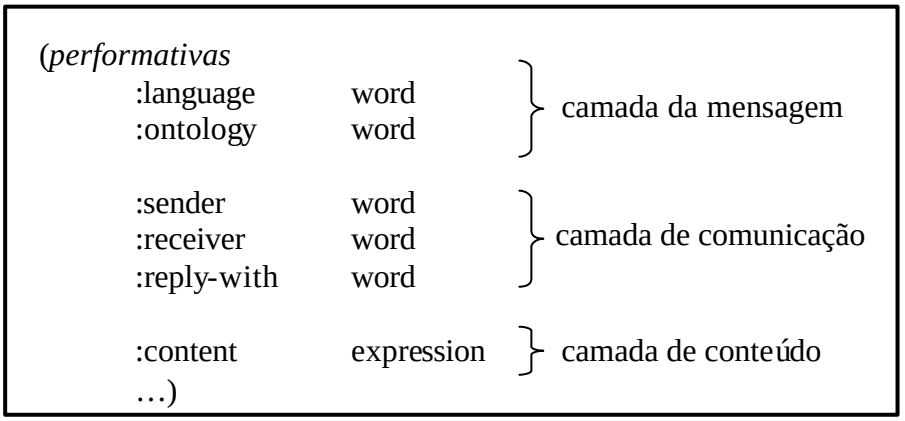
Agentes comunicam-se na busca de atingir os seus objetivos em um ambiente compartilhado. Em um SMA, as ações de agentes são ações coordenadas, seja para cooperar ou competir (negociar). Em qualquer modelo de coordenação utilizado por agentes, a comunicação tem em geral, um papel central (BORDINI, 2002, p. 11).

3.2 KQML

KQML (*Knowledge Query and Manipulation Language*) é uma linguagem e um protocolo de comunicação, de alto nível. Foi desenvolvida dentro do *"Knowledge Sharing Effort"* patrocinado pelo DARPA, em 1993. KQML é independente do mecanismo de transporte (TCP/IP, SMTP, etc), independente da linguagem conteúdo (KIF, SQL, PROLOG, etc), e independente da ontologia assumida pelo conteúdo (MENESES, 2001). Isto quer dizer que KQML é uma linguagem de comunicação entre agentes (LCA), de alto nível, onde a troca de mensagens é independente da linguagem de conteúdo e ontologia utilizada, ou seja, qualquer tipo de conteúdo pode ser enviado em uma mensagem KQML.

O formato da mensagem KQML é baseado na sintaxe da linguagem LISP, porém, os argumentos são identificados por palavras-chave precedidas por dois pontos. No quadro 2 é ilustrado o padrão do formato para mensagens KQML.

Quadro 2 – Formato de mensagem KQML



Fonte: Adaptado de Hübner (2001).

A camada de mensagem inclui informações que ajudam o receptor a entender o conteúdo da mensagem. O campo da performativa identifica a intenção do remetente com a mensagem, no valor da palavra-chave `:language` informa-se a linguagem em que a mensagem é expressa, e o valor de `:ontology` é o vocabulário usado para as palavras na mensagem. A camada de comunicação descreve os parâmetros de comunicação de baixo nível, como também a identificação do remetente e do receptor, e uma única identificação para a mensagem (palavra-chave `:reply-with`).

KQML inclui também uma arquitetura para plataformas de comunicação entre agentes. Esta arquitetura está baseada na presença de "facilitadores" que são agentes que tem conhecimento sobre quais agentes estão acessíveis e quais são as habilidades de cada um. Isto permite que sociedades de agentes funcionem de forma "aberta", ou seja, que agentes entrem e saiam da sociedade. Se um agente quiser entrar em uma sociedade, basta avisar o facilitador a respeito de suas habilidades e qual o seu endereço físico na rede. Através de consultas ao facilitador, todo agente pode localizar outros agentes com os quais seja interessante interagir (BORDINI, 2002).

A linguagem KQML foi concebida com o objetivo de se tornar um padrão de comunicação entre agentes. Algumas organizações foram criadas com o propósito de criar padrões para agentes inteligentes, como por exemplo, a *Foundation for Intelligent Physical Agents* (FIPA) e a *Agent Society*. A linguagem de comunicação entre agentes proposta pela FIPA foi intencionalmente baseada em KQML, para permitir a migração daqueles sistemas que já faziam uso de KQML, apesar de a linguagem da FIPA ter sido melhorada em versões mais recentes, ao contrário do KQML. Porém KQML possui o grande mérito de ter sido a primeira linguagem de comunicação entre agentes a ser de fato implementada e utilizada em muitos sistemas, utilizando uma abordagem própria para comunicação de agentes (BORDINI, 2002, p. 16).

3.3 SACI

A implementação de um SMA freqüentemente necessita da comunicação entre os agentes em um ambiente distribuído. Contudo, algumas vezes isso não é uma tarefa simples. Para isto é preciso algum conhecimento sobre protocolos de rede, portas TCP/IP, programação distribuída (RMI, CORBA, DCOM), e muitas outras tecnologias. Com o intuito de ajudar o desenvolvedor de SMA, existe o que é chamado de ambiente de desenvolvimento de SMA que fornece várias ferramentas para o desenvolvimento de aplicações que utilizam a abordagem de SMA.

Uma destas ferramentas é o SACI, que torna a programação da comunicação entre agentes mais fácil. Duas principais vantagens são fornecidas pelo SACI: 1) uma API para compor, enviar e receber mensagens KQML; 2) uma coleção de ferramentas que simplificam algumas dificuldades inerentes para distribuição (serviço de nome de agente, serviço de páginas amarelas, lançamento remoto, etc.). O SACI permite que agentes distribuídos se comuni-

quem de forma fácil. Possui um conjunto de classes Java que pode ser usado para ajudar o desenvolvimento de sociedades de agentes distribuídos.

O SACI foi desenvolvido com base na especificação KQML, possuindo as seguintes características principais (HÜBNER, 2001):

a) os agentes utilizam KQML para se comunicar: há funções para compor, enviar e receber mensagens KQML;

b) os agentes são identificados por um nome: as mensagens são transportadas utilizando-se somente o nome do receptor, sua localização na rede é transparente;

c) um agente pode conhecer os outros por meio de um serviço de páginas amarelas: agentes podem registrar seus serviços no facilitador e perguntá-lo sobre que serviços são oferecidos por quais agentes;

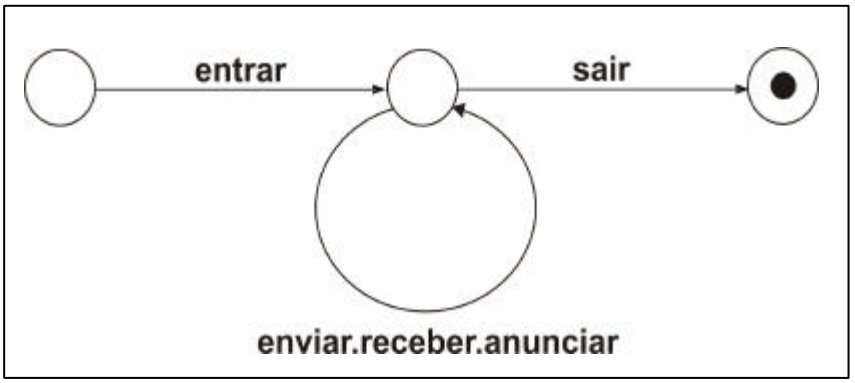
d) os agentes podem ser implementados como *applets* e terem sua interface em uma *home-page*;

e) os agentes podem ser iniciados remotamente;

f) os agentes podem ser monitorados: os eventos sociais (entrada na sociedade, saída, recebimento ou envio de mensagens) podem ser visualizados e armazenados para análise futura.

No SACI os agentes estão agrupados em sociedades, possuindo uma identificação única, onde cada agente interage com os demais utilizando uma linguagem comum. Estes agentes oferecem serviços aos demais agentes de sua sociedade, tendo os agentes um ciclo de vida, confirme descrito na figura 3:

Figura 3 – Ciclo de vida do agente

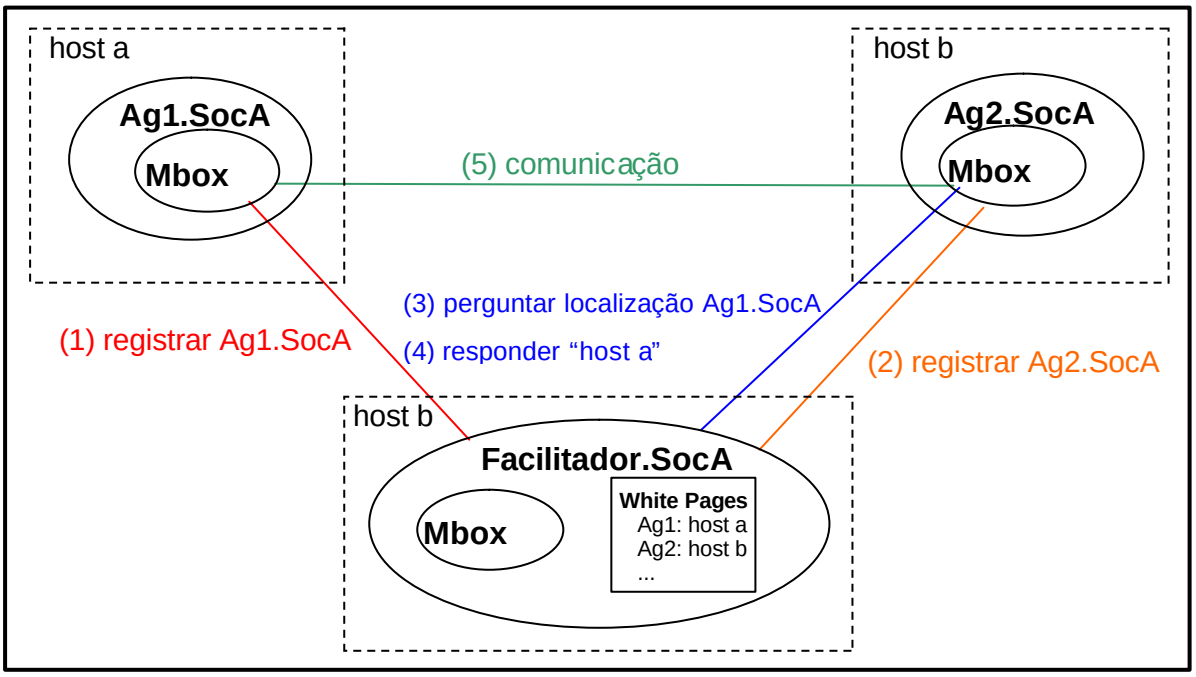


Fonte: Adaptado de Hübner (2001).

3.3.1 ENVIANDO E RECEBENDO MENSAGENS

Para o envio e recebimento de mensagens o SACI possui um componente denominado *Mbox*. A finalidade do *Mbox* é tornar transparente o envio e o recebimento de mensagens. Este componente possui funções que encapsulam a composição de mensagens KQML, o envio síncrono e assíncrono de mensagens, o recebimento de mensagens, o anúncio e a consulta de habilidades e o *broadcast* de mensagens, conforme esquema ilustrado na figura 4. Nesta figura, o agente *Ag2* da sociedade *SocA* deseja comunicar-se com o agente *Ag1* que pertence à mesma sociedade e cujo nome é conhecido pelo *Ag2*. Inicialmente, o *Mbox* do *Ag2* precisa saber a localização do *Ag1*, então, pergunta ao facilitador da sociedade *SocA* tal *localização* (seta azul (3)), que lhe responde "*host a*" (seta azul (4)). Tendo a localização, o *Ag2* inicia a comunicação com o agente *Ag1* através do seu *Mbox* (seta verde (5)) (HÜBNER, 2001, p. 10).

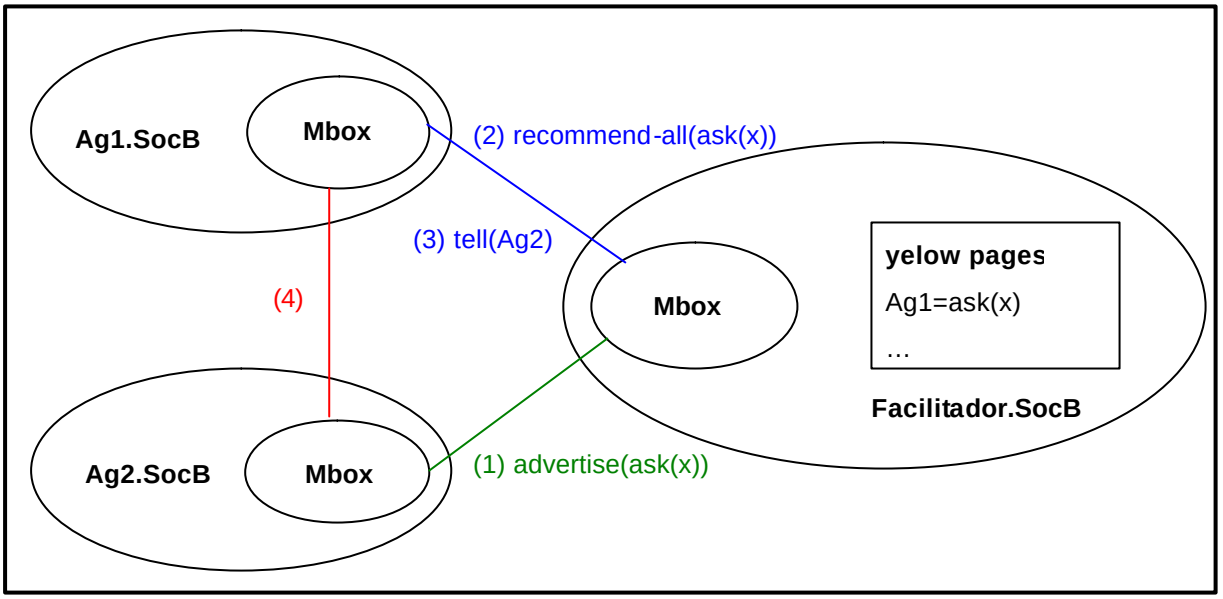
Figura 4 – Serviço de páginas brancas



Fonte: Adaptado de Hübner (2001).

Para serem conhecidos na sociedade, os agentes podem anunciar suas habilidades ao facilitador de forma análoga ao serviço de páginas amarelas. Assim, quando um agente precisa de um serviço e não conhece o nome de um agente capaz de realizá-lo, pode requisitar ao facilitador uma lista de agentes com tal habilidade, conforme ilustrado na figura 5. Na verdade, este é apenas um modelo disponível no SACI para a apresentação, sendo que outras formas podem ser utilizadas, como o *broadcasting* das habilidades (HÜBNER, 2001).

Figura 5 – Serviço de páginas amarelas



Fonte: Adaptado de Hübner (2001).

No caso da figura 5, tendo o agente *Ag*2 anunciado a habilidade *x* (*seta verde (1)*), se o agente *Ag*1 pedir ao facilitador uma lista de agentes com a habilidade *x* (*seta azul (2)*), receberá *Ag*2 como resposta (*seta azul (3)*) e poderá iniciar a comunicação com este agente (*seta vermelha (4)*).

Um exemplo de mensagem do agente *Ag*2 anunciando uma habilidade ao facilitador é ilustrado no quadro 3.

Quadro 3 – Anúncio de habilidade

```
(advertise
        :receiver       Facilitador
        :sender         Ag2
        :language       KQML
        :ontology       yp
        :content        (ask – one
                                :receive    Ag2
                                :language   alg
                                :ontology   math
                                :content    “X + Y”) )
```

Fonte: Adaptado de Hübner (2001).

No quadro 3, o *Ag2* usa uma mensagem KQML para anunciar uma habilidade para o facilitador, isso significa que o *Ag2* pode responder mensagens com a performativa `ask-one`, a linguagem `alg`, a ontologia `math` e uma fórmula com dois operandos. Então o agente Ag1

usa a mensagem descrita no quadro 4 para obter uma lista de agentes, e a resposta do facilitador no quadro 5.

Quadro 4 – Solicitação da lista de agentes

```
(recommend-all
        :receiver       Facilitador
        :sender         Ag1
        :reply-with     id1
        :language       KQML
        :ontology       yp
        :content        (ask – one
                                :language   alg
                                :ontology   math
                                :content    "X + Y") )
```

Fonte: Adaptado de Hübner (2001).

Quadro 5 – Resposta do facilitador

```
(tell
        :receiver       Ag1
        :sender         Facilitador
        :in reply-to    id1
        :language       KQML
        :ontology       yp
        :content        (Ag2))
```

Fonte: Adaptado de Hübner (2001).

O *Mbox* serve como uma interface entre os agentes e a sociedade. Seu principal objetivo é entregar uma mensagem para o receptor. Para se comunicarem no SACI, os agentes utilizam os métodos do *Mbox* que possuem tarefas de interação encapsuladas (para enviar mensagens, receber mensagens, anunciar, etc.). Um agente pode ter mais de um objeto *Mbox*, cada um destes objetos correspondem a uma única sociedade a qual eles pertencem.

Um exemplo de como esta comunicação pode ser implementada, é apresentado no quadro 6.

Quadro 6 – Utilizando o SACI para desenvolver agentes

```
01 import saci.*;
02
03 public class SampleSaciAg extends Agent {
04
05   public void run ( ) {
06     try  {
07       Message m = new Message (“(ask-one”+
08                  ” :content \”2+4\””+
09                  ” :receiver APlusServer”+
10                  ” :reply-with rAdd)”);
11       mbox.sendMsg(m);
12       boolean ok = false;
13       while ( ! ok ) {
14          Message r = mbox.polling( );
15          String irt = (String) r.get(”in-reply-to”);
16          if (irt.equals(”rAdd”)) {
17            String ans = (String) r.get(”content”);
18            System.out.println(”Answer is ” + ans ) ;
19            ok = true;
20          }
21       }
22     } catch ( Exception e ) {
23       System.err.println(”Error ” + e ) ;
24   }
14
25
26   public static void main ( String [ ] args ) {
27     SampleSaciAg a = new SampleSaciAg ( );
28     if ( a.enterSoc(”SampleSaciAg”)) {
29       a.run( );
30       a.leaveSoc( ) ;
31       System.exit(0) ;
32     }
33   }
34 }
```

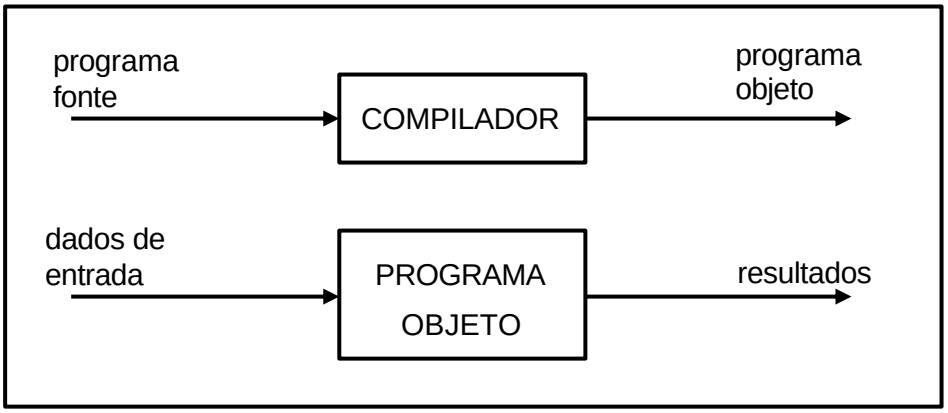
Fonte: Adaptado de Hübner (2001).

É no método *main* da linha 27 que o ciclo de vida do agente está implementado, o agente entra em uma sociedade (linha 29), envia e recebe mensagens (linha 30), através do método *run*, e deixa a sociedade (linha 31). No método *run*, o agente cria uma mensagem KQML (linha 7), a envia de forma não sincronizada utilizando sua herança da interface *mail box* (linha 11), recebe uma mensagem enviada (linha 14), e, se a mensagem é uma resposta para uma pergunta previamente enviada (linha 16), mostra a resposta (linha 18) (HÜBNER, 2001, p. 14).

4 COMPILADORES

Compiladores são tradutores que mapeiam programas escritos em linguagem de alto nível para programas equivalentes em linguagem simbólica ou linguagem de máquina. A execução de um programa escrito em linguagem de alto nível é, basicamente, um processo de dois passos, conforme é mostrado na figura 16 (PRICE, 2001).

Figura 16 – Compilação de um programa



Fonte: Price (2001).

O intervalo de tempo no qual ocorre a conversão de um programa fonte para um programa objeto é chamado de *tempo de compilação*. O programa objeto é executado no intervalo de tempo chamado *tempo de execução*. Observa-se que o programa fonte e os dados são processados em momentos distintos, respectivamente, tempo de compilação e tempo de execução (PRICE, 2001).

O compilador passa por três fases na tradução da linguagem fonte para a linguagem objeto, esta três fases estão descritas nos parágrafos abaixo.

Análise léxica – Esta é a fase em que é dividido a partir do código-fonte o tipo de palavras, como identificadores, palavras reservadas, números reais, etc. Cabe a análise léxica definir se um identificador é ou não uma palavra reservada (JOSÉ, 1987). A saída do analisador léxico é uma cadeia de *tokens* que é passada para a próxima fase, a análise sintática.

Análise sintática – A fase de análise sintática tem por função verificar se a estrutura gramatical do programa está correta, isto é, se essa estrutura foi formada usando as regras gramaticais da linguagem. O analisador sintático identifica seqüências de símbolos que constituem estruturas sintáticas (expressões, comandos), através de uma varredura ou "parsing" da representação interna (cadeia de *tokens*) do programa fonte. O analisador sintático produz

(explícita ou implicitamente) uma estrutura em árvore, chamada de *árvore de derivação*, que exibe a estrutura sintática do texto fonte, resultante da aplicação das regras gramaticais da linguagem. Em geral, a árvore de derivação não é produzida explicitamente, mas sua construção está implícita nas chamadas das rotinas recursivas que executam a análise sintática (PRICE, 2001).

As regras gramaticais que definem as construções da linguagem podem ser descritas através de produções cujos elementos incluem símbolos *terminais* (aqueles que fazem parte do código fonte) e símbolos *não-terminais* (aqueles que geram outras regras).

Análise semântica – o analisador semântico determina se as estruturas sintáticas analisadas fazem sentido, ou seja, verifica se um identificador declarado como variável é usado como tal, determina também se existe compatibilidade entre operandos e operadores em expressões.

4.1 BACKUS-NAUR FORM (BNF)

A forma normal de Backus é bastante utilizada na definição de linguagens de programação, sendo introduzida na década de 60 na descrição do ALGOL. Consiste em uma metalinguagem capaz de descrever produções gramaticais recursivamente, onde cada produção possui uma regra de substituição, que permite a associação de uma ou mais cadeias a outra cadeia de símbolos. Os símbolos da metalinguagem BNF são apresentados na tabela 1.

Tabela 1 – Símbolos da metalinguagem BNF

<*x*>	Representa um não-terminal, cujo nome é dado por um cadeia x de caracteres quaisquer. Os caracteres “<” e “>” são usados para delimitar o nome do não-terminal.
X	Corresponde a um símbolo terminal. Note que o fato de um símbolo estar fechado ou não entre "<" e ">" faz a diferença entre a representação de terminais e não-terminais de uma gramática.
::=	Representa a associação entre os dois lados da produção. Traçando um paralelo entre a notação utilizada para formalizar as gramáticas, o símbolo ::= substitui – deve ser lido como "é definido como".
|	É equivalente ao operador lógico "ou" sendo utilizado quando se deseja definir várias cadeias alternativas para uma mesma classe sintática.
e	Significa uma cadeia vazia. O artifício da cadeia vazia é empregado para representar classes sintáticas sem qualquer significado semântico.
Yz	Representa uma cadeia construída pela concatenação dos elementos *y* e *z* nesta ordem. Es-

tes dois elementos podem, por sua vez, ser símbolos de terminais, de não-terminais, de cadeia vazia, ou mesmo outras cadeias.

Maiores informações sobre a notação BNF podem ser encontradas em Aho (1988) e José (1987).

4.2 JAVACC

Java Compiler Compiler (JavaCC) é um gerador de *parser* para uso com aplicações em Java. Esse gerador de *parser* é uma ferramenta que lê uma especificação de gramática e converte-a em um programa Java que possa reconhecer se um determinado texto pertence a gramática especificada. Além do gerador de *parser*, o JavaCC fornece outras potencialidades relacionadas a geração do *parser* tal como as ações da árvore de derivação sintática, eliminação de erros, etc.

O JavaCC, que já foi chamado de *Jack*, foi criado pela SUN para a comunidade de programadores Java. Constitui uma ferramenta poderosa, flexível e simples de usar. JavaCC aproveita todas as características da linguagem Java para prover facilidades na construção de analisadores sintáticos. Principalmente o fato de Java ser orientado a objetos tornando o uso do JavaCC proveitoso, facilitando a geração e adaptação dos códigos que avaliam os nós da árvore sintática. Uma outra característica da linguagem Java, o tratamento de exceções, torna o gerenciamento de erros sintáticos de fácil implementação e leitura. Além dessas facilidades, o pacote JavaCC dispõe de duas ferramentas, além do próprio JavaCC: o *jjtree* e o *jjdoc* (TAVARES, 2001).

O *jjdoc* é um utilitário que lê uma especificação JavaCC e gera um arquivo HTML com toda a gramática da linguagem na notação BNF. Aproveitando-se de *links* internos, pode-se facilmente navegar entre os símbolos não-terminais. O *jjdoc* é útil para documentação e para se verificar se a gramática foi corretamente descrita no arquivo JavaCC (TAVARES, 2001).

O *jjtree* é um poderoso utilitário para criação e manipulação de árvores sintáticas. Este trabalha sobre um arquivo ligeiramente modificado de JavaCC, com extensão *.jjt*, com a adição de opções para o programa e de ações semânticas sobre a árvore sintática. Ele gera uma especificação para JavaCC, que não é muito diferente do arquivo original acrescido de melho-

res definições das ações semânticas, além de gerar os arquivos com as classes que implementam a árvore sintática (TAVARES, 2001).

5 LINGUAGEM LARF

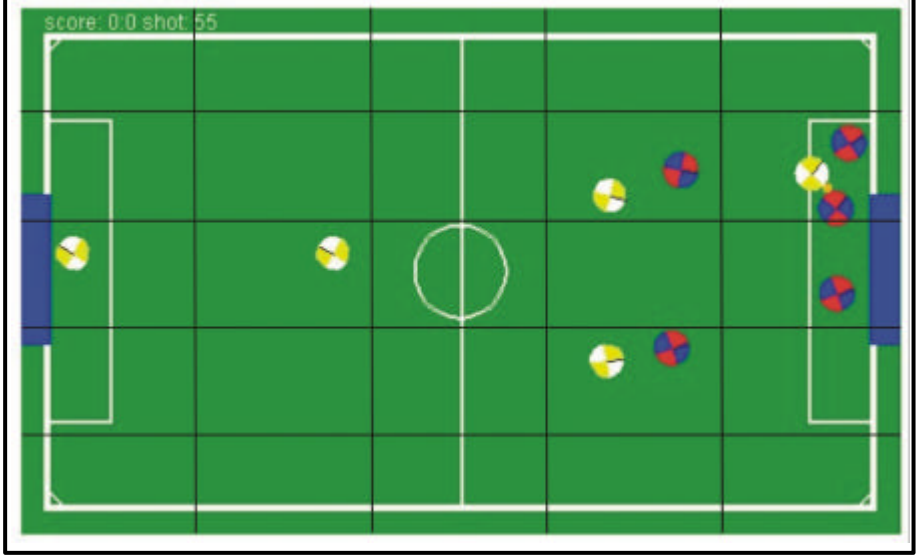
5.1 VISÃO GERAL DA LINGUAGEM

A Linguagem para Definição de Agentes Robôs Jogadores de Futebol (LARF) foi divida em três partes. A primeira parte define em quantas seções o campo vai ser dividido, na segunda parte do arquivo é definido o comportamento dos robôs e a terceira parte é reservada para a escrita de rotinas genéricas aos agentes (robôs) do time (SCHLEI, 2002).

5.1.1 DEFINIÇÃO DO CAMPO

Esta seção foi reservada para criar uma matriz de pequenas seções lógicas sobre o campo de futebol, estas seções existem para facilitar a definição da área na qual o robô vai atuar no campo. Para definir a quantidade de seções foi criada a declaração *DimensaoDoCampo*. A declaração *DimensaoDoCampo* recebe como parâmetro dois valores separados por uma vírgula, para facilitar a implementação indica-se o uso de valores na faixa de 1 até 10. A figura 6 ilustra o resultado da aplicação do exemplo descrito em seu rodapé.

Figura 6 – Dimensão do campo




```
DefinicaoDoCampo(5,5); Define o campo com 5 linhas e 5
                       colunas totalizando 25 seções.
```

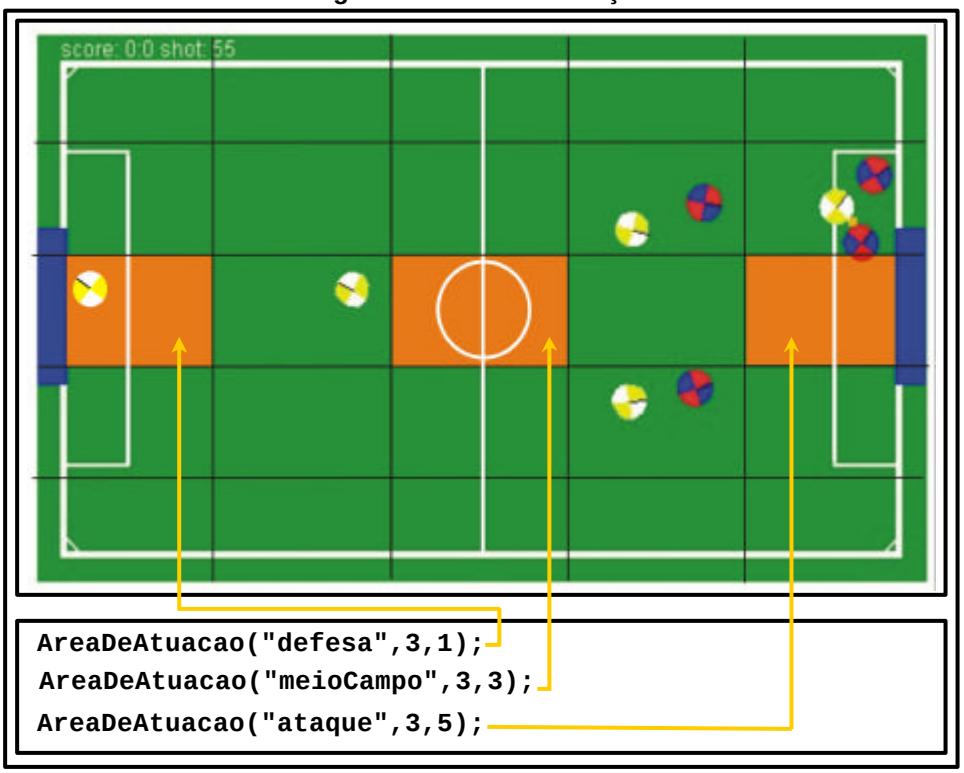
5.1.2 DEFINIÇÃO DO JOGADOR

Para definir os comportamentos dos cinco robôs, utiliza-se a segunda seção do arquivo. A descrição do comportamento dos jogadores inicia-se com a declaração *DefinicaoDoJogador* seguido de um parâmetro que indica o número do jogador, este parâmetro deve estar no intervalo de 1 até 5 sendo que a declaração de cada jogador é exclusiva. Dentro desta seção existem 3 subseções: área de atuação, controle principal e comportamentos.

5.1.2.1 ÁREA DE ATUAÇÃO

Esta seção tem a finalidade de nomear as seções lógicas criadas com o comando *DimensaoDoCampo* no começo do programa. Após o nome da seção deverá ser informado dois valores que indicam as coordenadas da matriz lógica (estes valores devem estar dentro dos limites definidos pelo comando *DimensaoDoCampo*). A declaração *AreaDeAtuacao* pode ser utilizada várias vezes seguidas usando o mesmo nome para várias seções lógicas diferentes ou nomes diferentes para criar várias áreas de atuações diferentes para o respectivo robô. A utilização do comando *AreaDeAtuacao* é apresentada na figura 7.

Figura 7 – Área de atuação



Fonte: Adaptado de Schlei (2002).

5.1.2.2 CONTROLE PRINCIPAL

Esta seção tem o objetivo de fazer o controle do comportamento ativo no robô, para isso é utilizada a declaração de verificação *SE* (a classe *SE* é apresentada com mais detalhes na seção 4.1.4). A declaração *SE* recebe como parâmetro uma expressão lógica que caso seja verdadeiro executa a declaração *Ativa* (a classe *Ativa* é apresentada com mais detalhes na seção 4.1.4). A declaração *Ativa* recebe como parâmetro o nome de um comportamento que é descrito na seção seguinte do arquivo. No quadro 7 é apresentado um exemplo de implementação da seção *ControlePrincipal*. Neste exemplo verifica-se se a bola está na área do jogo, e se estiver, será ativado um comportamento previamente definido para o robô.

Quadro 7 – Exemplo da seção controle principal

```
ControlePrincipal
Inicio
      Se ( BolaNaArea(“jogo”)) entao
      Ativa ( Comp_Novo );
Fim;
```

Fonte: Schlei (2002).

5.1.2.3 COMPORTAMENTOS

A seção de comportamentos é destinada para a definição de rotinas especiais que vão ser executadas repetidas vezes enquanto este comportamento continuar ativo pela seção de controle principal do robô (SCHLEI, 2002, p. 31). No exemplo do quadro 8, o robô vai andar a velocidade de 0.8, virar para a direção da bola e, se ele estiver com a posse da bola irá girar para a direita num ângulo de 10 graus.

Quadro 8 – Exemplo da seção comportamentos

```
Comportamento Comp_Novo
Inicio
      Andar(0.8);
      VirarParaBola();
      Se ( ComPosseDaBola ) entao
            inicio
                  GirarParaDireita(10);
            Fim;
Fim;
```

Fonte: Schlei (2002).

5.1.3 ROTINAS GENÉRICAS

A seção *rotina* é reservada para a escrita de procedimentos genéricos aos robôs, sendo assim possível escrever um certo comportamento uma única vez e este ser usado para todos os robôs, sendo este comportamento automaticamente adequado às condições de cada robô. Esta adequação é possível pela forma como foi implementado o compilador. Uma rotina é constituída de um bloco de declarações da linguagem. Estes blocos de controle são compostos de ações primitivas do robô, controles de fluxo (*SE*) e chamadas de outras rotinas possibilitando chamadas recursivas. O quadro 9 exemplifica uma rotina para o robô chutar a bola para o gol. Através do comando condicional *SE*, faz-se a verificação se o jogador está virado para o lado do gol, e se caso estiver, chuta a bola para o gol.

Quadro 9 – Exemplo da seção rotinas genéricas

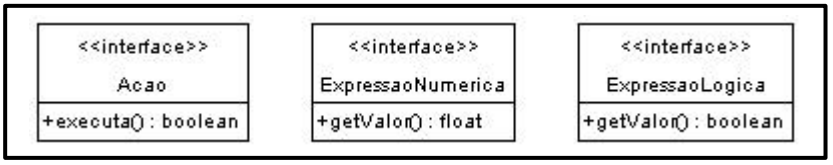
```
Rotina ChutarProGol
Inicio
  Se ( JogadorNoAngulo(AnguloDoGol()) ) entao
  Inicio
      ChutarBola();
  Fim;
Fim:
```

Fonte: Schlei (2002).

5.1.4 ESPECIFICAÇÃO DA LINGUAGEM LARF

Na especificação da linguagem LARF, Schlei (2002) definiu três tipos de interfaces para atender as necessidades gerais da implementação das classes da linguagem. A interface *Ação*, utilizada para generalizar a chamada para classes instanciadas. A interface *Expressao-Numerica* serve para implementar classes que retornam algum valor numérico. E a interface *ExpressaoLogica* para implementar classes que retornam um valor booleano. A figura 8 ilustra as interfaces da modelagem.

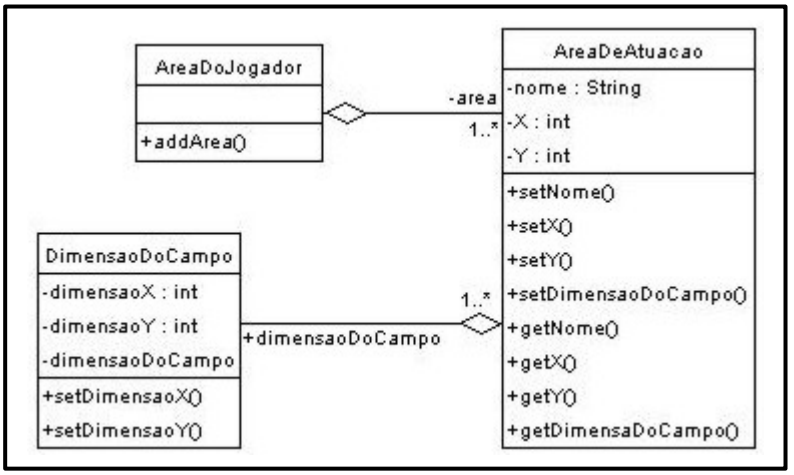
Figura 8 – Interfaces da modelagem



Fonte: Schlei (2002).

Para o controle da área de atuação do jogador foram desenvolvidas três classes. A classe *DimensaoDoCampo* possui a configuração das seções lógicas do campo. A classe *Área de Atuação* guarda as coordenadas X,Y de uma das seções lógicas criadas com base na dimensão do campo. Por fim a classe *AreaDoJogador* detém o conjunto de várias classes do tipo *AreaDeAtuação* que foram criadas. Na figura 9 é apresentado o diagrama das classes da área de autação do jogador.

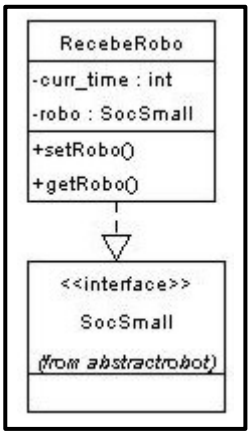
Figura 9 – Diagrama das classes da área de atuação



Fonte: Schlei (2002).

Com exceção das classes, *AreaDoJogador, DimensaoDoCampo, AreaDeAtuacao,* todas as classes da linguagem estendem a classe *RecebeRobo*. Esta por sua vez, possui métodos para obter e retornar valores para o robô. Ela possui também a variável *curr_time*, para garantir que mais de uma ação não seja realizada num mesmo tempo de execução. A interface *SocSmall* faz parte do pacote *TeamBots.* A figura 10 ilustra o diagrama da classe *RecebeRobo*.

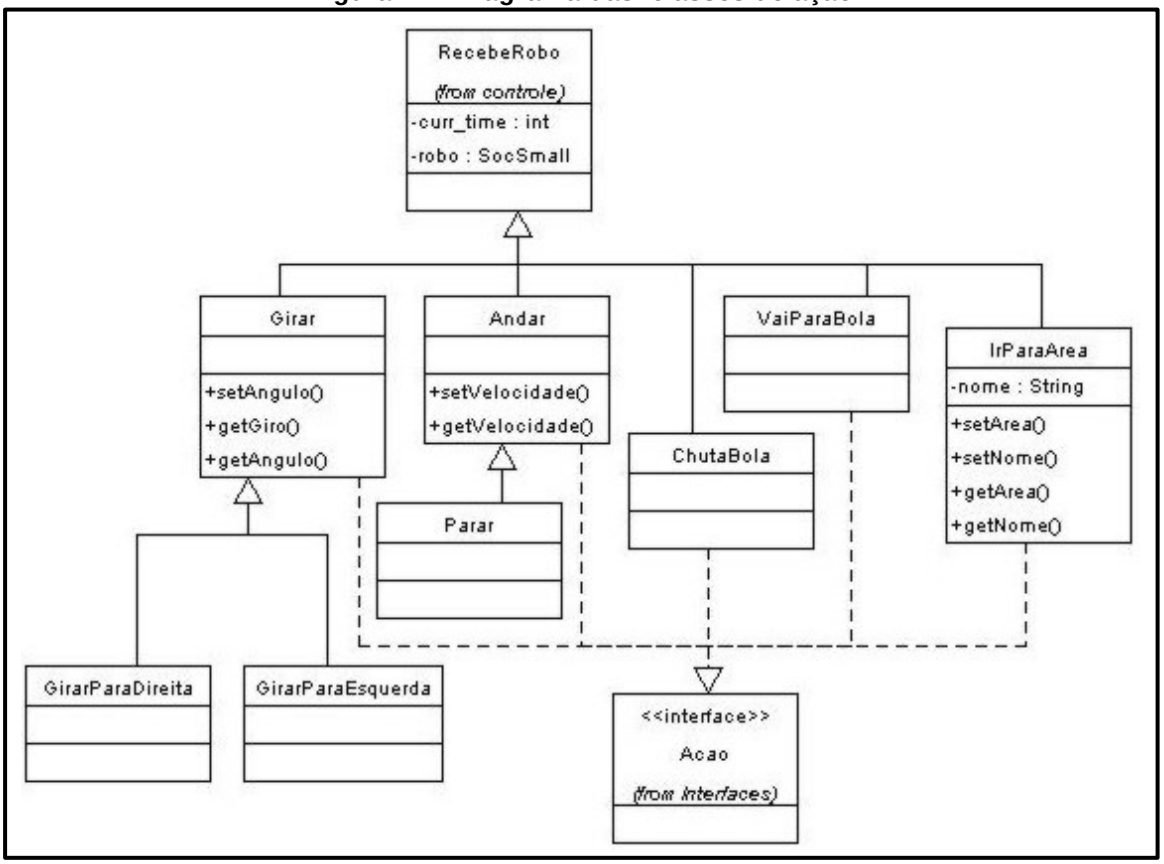
Figura 10 – Diagrama da classe *RecebeRobo*



Fonte: Schlei (2002).

As classes que possuem ação direta sobre o comportamento dos robôs são as classes de ação. Estas classes apresentam as ações primitivas dos robôs, como andar, virar para a bola, ir para a área, chutar a bola, girar e parar. Classes que necessitam de retorno de valor numérico para fazer comparações com outros valores como distância do robô até o gol, distância da bola, distância do jogador, entre outras, fazem a implementação da interface *ExpressaoNumerica*. Outras classes necessitam de retorno com valor lógico implementando, neste caso a interface *ExpressaoLogica*. Utilizando esse tipo de expressão foram implementadas classes para retornar para o robô se a bola está na área, se ele está com posse da bola, etc. O diagrama das classes de ação é apresentado na figura 11.

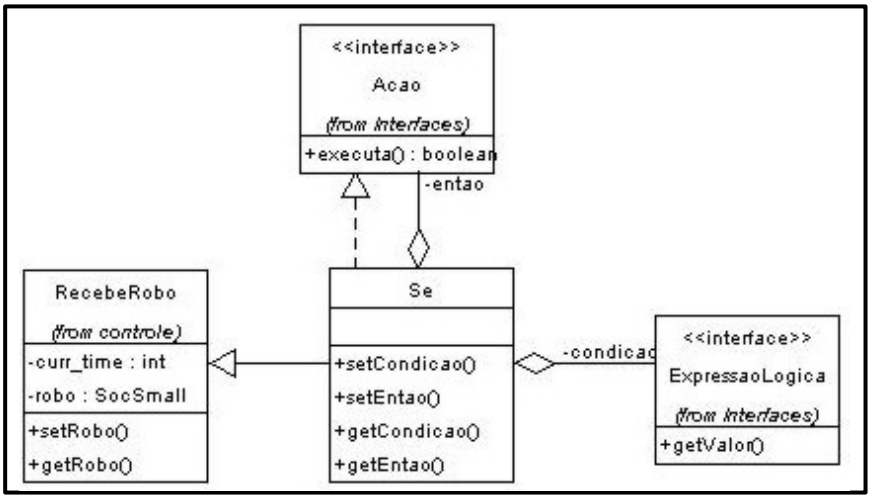
Figura 11 – Diagrama das classes de ação



Fonte: Adaptado de Schlei (2002).

O comando declarativo que faz o controle de fluxo disponível nesta linguagem é o *SE*. A partir da avaliação de uma expressão lógica usando as expressões de retorno numérico e lógico o comando *SE* ativa comportamentos ou faz chamadas a classes que implementam a classe Ação (SCHLEI, 2002, p. 41). Na figura 12 é apresentado o diagrama da classe *SE*.

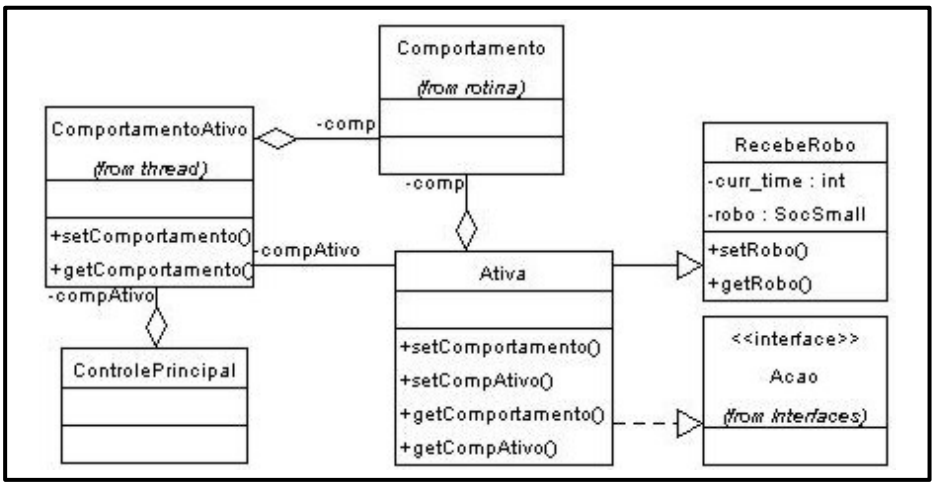
Figura 12 – Diagrama da classe *Se*



Fonte: Adaptado de Schlei (2002).

O conjunto de classes que fazem o controle do comportamento ativo do *AgenteJogador* é constituído de três classes. A classe *Ativa* ativa um comportamento, ou aborta o comportamento atual e ativa um novo comportamento. A classe *ComportamentoAtivo* é estendida da classe *Thread* e faz com que o comportamento atual ativo do robô seja executado repetidamente até que a instância da classe *Ativa* altere para um outro comportamento. E a classe *ControlePrincipal* possui obrigatoriamente uma agregação de um objeto da classe *ComportamentoAtivo* sendo utilizada para determinar qual o comportamento que deverá ser ativado. A modelagem das classes do controle de comportamento ativo do robô é ilustrada na figura 13.

Figura 13 – Controle de comportamento ativo

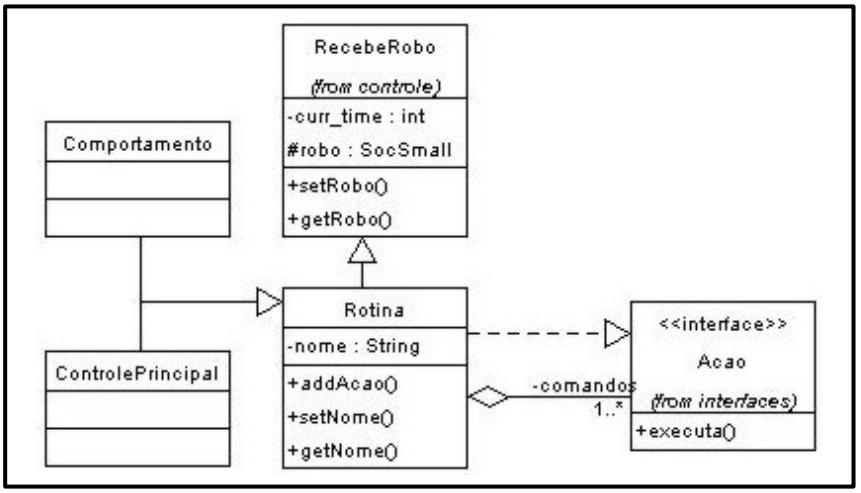


Fonte: Adaptado de Schlei (2002).

Foi implementada na linguagem a classe *Rotina* que possui uma agregação da interface *Ação*, este atributo é definido como uma pilha de objetos que implementam a interface *Ação.*

As ações são executadas na ordem em que aparecem nas declarações da implementação do comportamento do robô. A figura 14 apresenta o diagrama da classe *Rotina*.

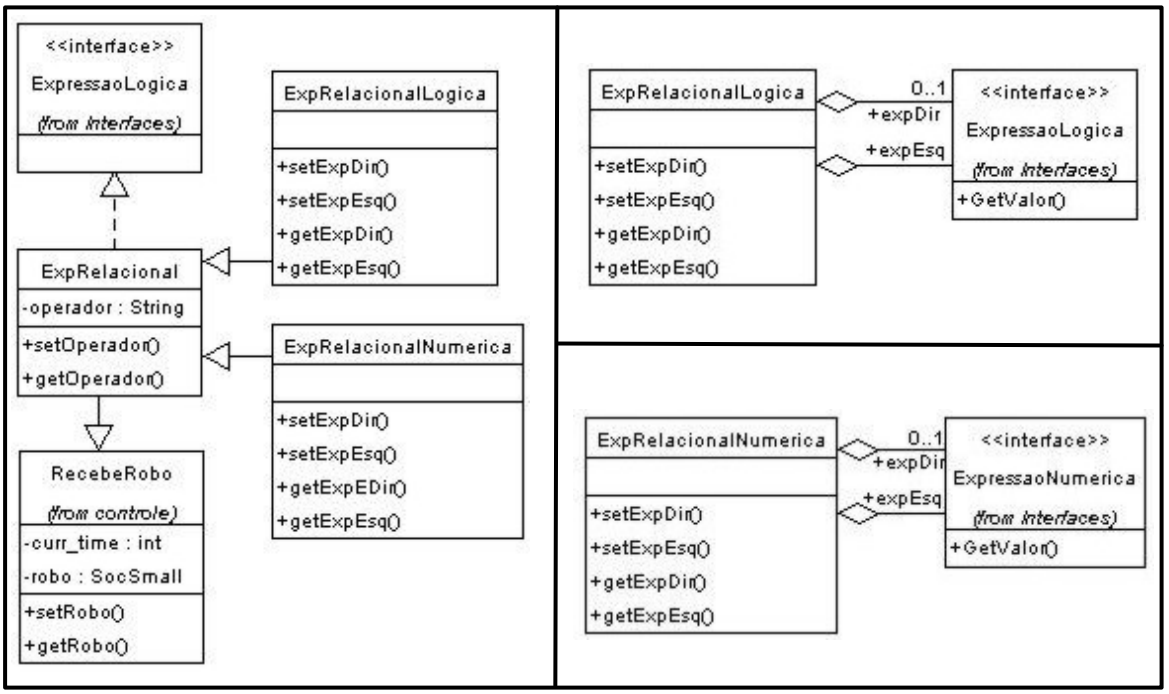
Figura 14 – Diagrama da classe *Rotina*



Fonte: Adaptado de Schlei (2002).

Com a finalidade de realizar comparações de valores das expressões numéricas e lógicas foram criadas duas classes (*ExpRelacionalLogica*, e *ExpRelacionalNumerica*), as duas classes estendem a interface *ExpressaoLogica*, assim retornando valores boolianos. Para isto a classe abstrata *ExpRelacional*, que estende a classe *RecebeRobo* e implementa a classe *ExpressaoLogica*, indica o operador a ser utilizado na avaliação lógica das classes que estendem dela. A *ExpRelacionalLogica*, estende a classe *ExpRelacional* e faz a comparação entre expressões lógicas usando operadores do tipo “E”, “OU” e “NÃO”. E a classe *ExpRelacionalNumerica* que também estende a classe *ExpRelacional*, faz comparações entre expressões numéricas utilizando os operadores “= =” (igual), “<” (menor), “<=” (menor ou igual), “>” (maior), “>=” (maior ou igual) e “<>” (diferente). Na figura 15 apresentam-se os diagramas das expressões relacionais explicitando a relação das classes *ExpressaoLogica* e *ExpressaoNumerica* com suas interfaces.

Figura 15 – Diagramas das expressões relacionais

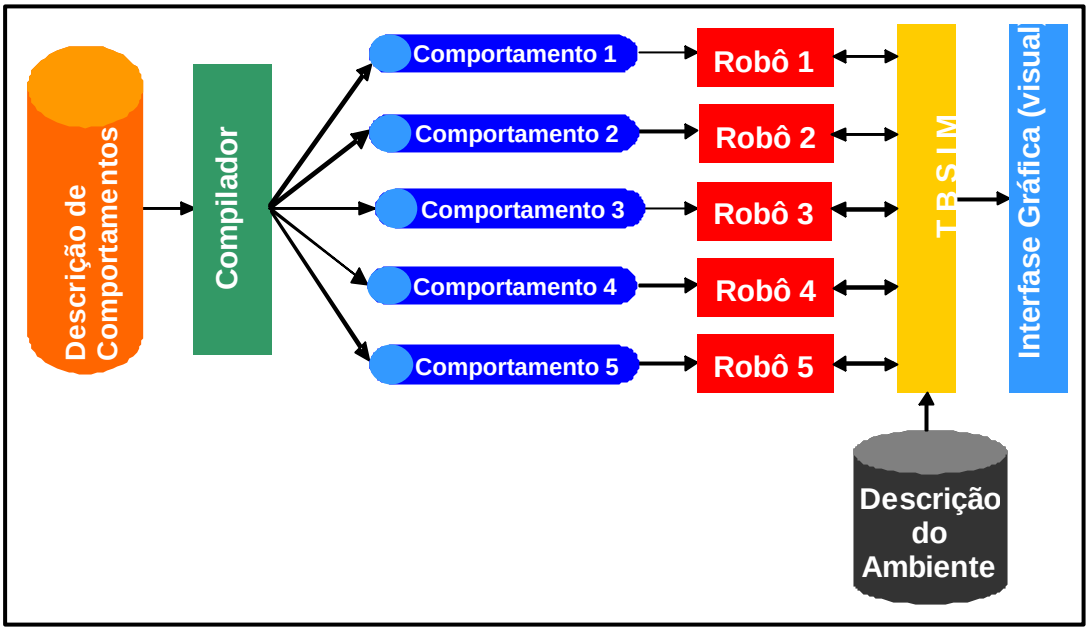


Fonte: Adaptado de Schlei (2002).

5.2 COMPILADOR DA LINGUAGEM LARF

O compilador da Linguagem LARF foi desenvolvido com o objetivo de traduzir um arquivo de descrição de comportamentos para ser executado pelo *TBSim*. Este compilador foi criado utilizando-se o software JavaCC (consulte a seção 4.2.2 para maiores informações sobre o software JavaCC). Após sua execução, se não for encontrado erros no arquivo de comportamentos de entrada, o compilador cria um arquivo de saída para cada jogador definido no arquivo de entrada, podendo o número de arquivos de saída variar de 1 até 5. Uma visão geral da linguagem é apresentada na figura 17.

Figura 17 – Visão geral da linguagem LARF



Fonte: Adaptado de Schlei (2002).

Cada arquivo de saída tem a descrição dos comportamentos individuais de cada agente jogador que foi definido no arquivo de entrada. O arquivo gerado como saída da compilação serve como entrada para a classe *AgenteJogador* (na figura 17 esta classe está representada pelas caixas "Robô"). O *AgenteJogador* é criado pelo programa *TBSim* que faz a leitura do arquivo de descrição de ambientes e, a partir deste, mostra a execução do *AgenteJogador* com o auxílio de uma interface gráfica. Primeiramente o *AgenteJogador* faz a leitura do arquivo de entrada que contem a descrição do comportamento, após a leitura do arquivo, faz a execução deste comportamento (SCHLEI, 2002).

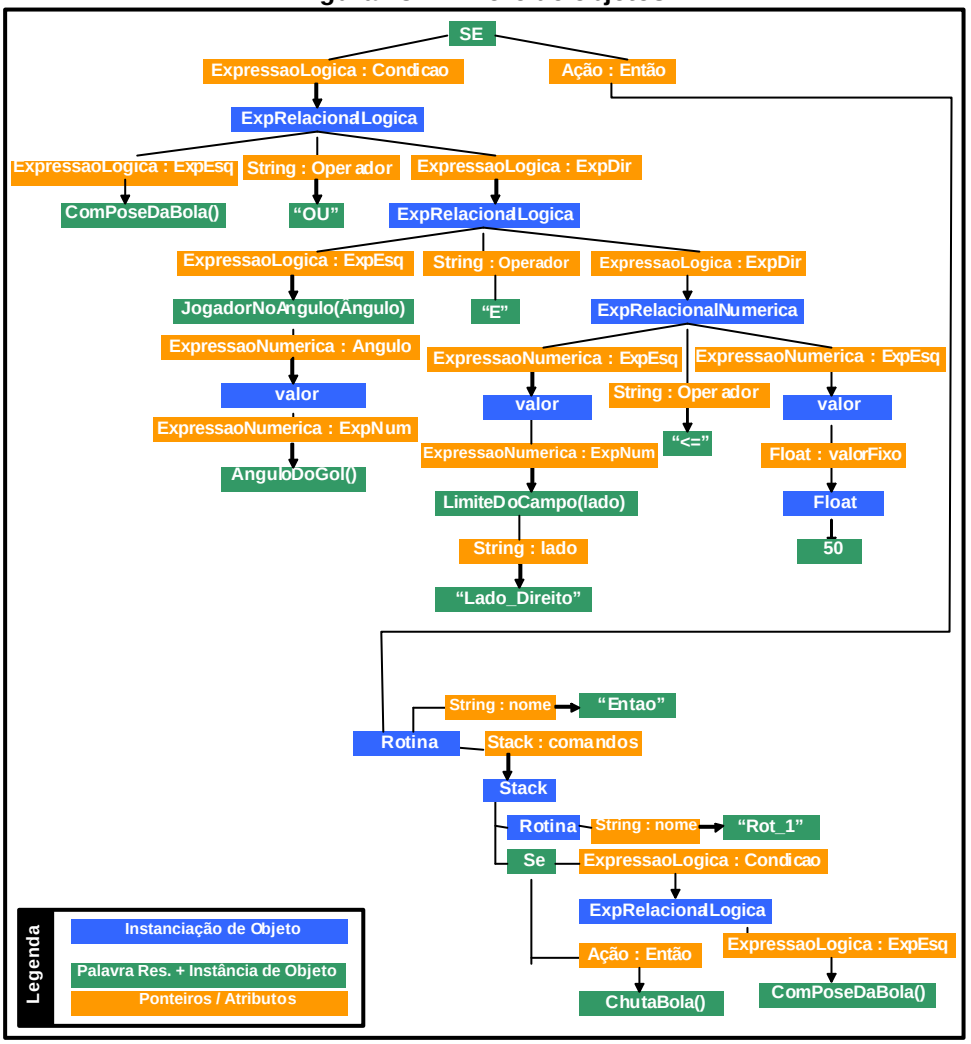
Para demonstrar o funcionamento do compilador, no quadro 10 apresenta-se um trecho de código. Este código é compilado através da linha de comando `Java Jogador joga.cmp`, onde `Jogador` é o nome da classe que implementa o compilador, e `joga.cmp` o arquivo de descrição de comportamentos (neste caso o código do quadro 10). Depois da compilação do arquivo, é gerada uma árvore de objetos conforme ilustrado na figura 18 que é executado pelo *AgenteJogador*.

Quadro 10 – Exemplo de declarações da linguagem

```
Se ( ComPoseDaBola() ou ( JogadorNoAngulo(AnguloDoGol()) E
       LimiteDoCampo (Lado_Direito) <= 50) ) entao
Inicio
       Chama ( Rot_1 );
       se ( ComPoseDaBola() ) entao
       inicio
              ChutarBola();
       Fim;
Fim;
```

Fonte: Schlei (2002).

Figura 18 – Árvore de objetos

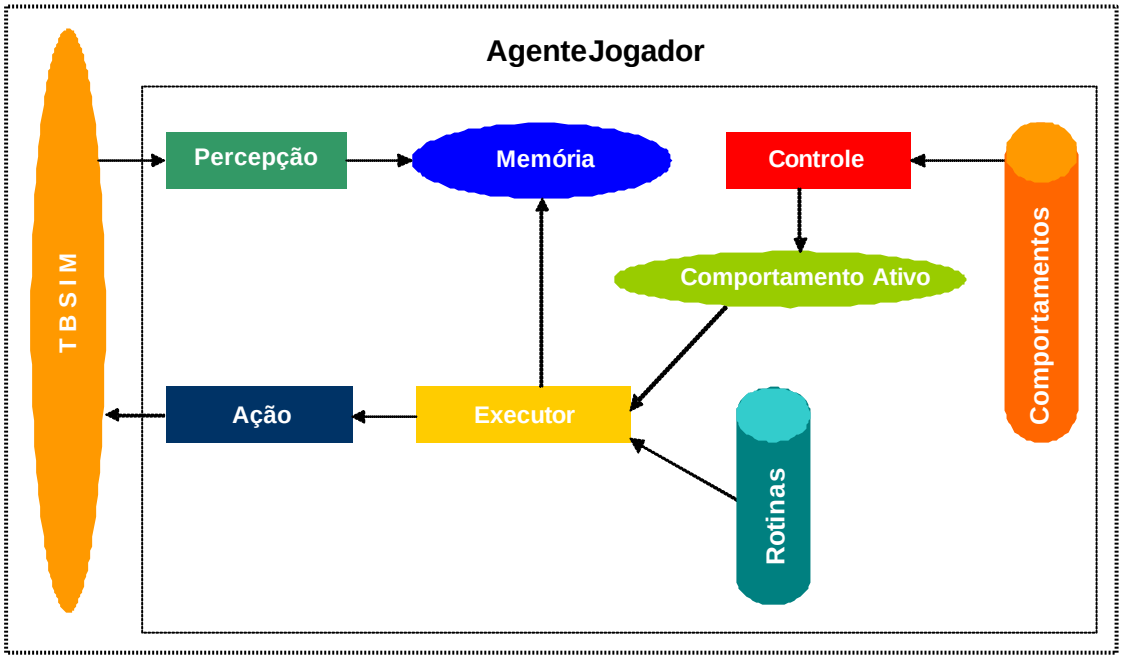


Fonte: Adaptado de Schlei (2002).

O agente jogador faz comunicação direta com o *TBSim*, sendo assim, toda a percepção do agente é fornecida pelo simulador. Com base nesta percepção o agente tem uma memória da situação do ambiente. A classe *AgenteJogador* é formada por vários módulos, como pode

ser visto na figura 19. O módulo *Controle* do agente é que faz a avaliação da memória e ativa o comportamento para a situação encontrada. Após a escolha do comportamento a ser ativado, o módulo *Executor* faz a execução das *Rotinas* (rotina é um conjunto de ações) deste comportamento. A execução das *Rotinas* é enviada para o *TBSim* que apresenta a execução das ações definidas na interface gráfica do simulador (SCHLEI, 2002).

Figura 19 – Arquitetura do *AgenteJogador*



Fonte: Adaptado de Schlei (2002).

No quadro 11 é apresentado um exemplo de execução do *AgenteJogador*. Neste exemplo os comandos são precedidos por rótulos que fazem referência a figura 19, assim possibilitando um maior entendimento na prática das funções dos módulos da classe.

Quadro 11 – Exemplo de execução do *AgenteJogador*

```
------------------ Definição do Campo ------------------
Caixa Percepção: - Dimensão do campo 1 por 1;

------------------------ Jogador 1 -------------------------
Caixa Percepção: - Definição do número do jogador 1;
Caixa Percepção: - Área de atuação do jogador, seção "jogo", na linha 1 e coluna 1;

Caixa Controle: - Se a bola está na área, ativa "comp";

Caixa Comportamentos: - Nome do comportamento "comp";
Caixa Ação: - Andar na velocidade 0.5;
Caixa Executor: - VaiPraBola;

------------------------ Jogador 2 -------------------------
Caixa Percepção: - Definição do numero do jogador 2;
Caixa Percepção: - Área de atuação do jogador, seção "jogo" na linha 1 e coluna 1;
```

```
Caixa Controle: - Se a bola está na área ativa "comp";

Caixa Comportamentos: - Nome do comportamento "comp";
Caixa Ação: - Andar na velocidade 0.8;
Caixa Executor: - VaiPraBola;

------------------------ Jogador 3 -------------------------
Caixa Percepção: - Definição do numero do jogador 3;
Caixa Percepção: - Área de atuação do jogador, seção "jogo" na linha 1 e coluna 1;

Caixa Controle: - Se a bola está na área ativa "comp";

Caixa Comportamentos: - Nome do comportamento "comp";
Caixa Ação:  Andar na velocidade 1;
Caixa Executor: - VaiPraBola;

------------------------ Jogador 4 -------------------------
Caixa Percepção: - Definição do numero do jogador 4;
Caixa Percepção: - Área de atuação do jogador, seção "jogo" na linha 1 e coluna 1;

Caixa Controle: - Se a bola está na área ativa "comp";

Caixa Comportamentos: - Nome do comportamento "comp";
Caixa Ação: - Andar na velocidade 0.7;
Caixa Executor: - VaiPraBola;

------------------------ Jogador 5 -------------------------
Caixa Percepção: - Definição do numero do jogador 3;
Caixa Percepção: - Área de atuação do jogador, seção "jogo" na linha 1 e coluna 1;

Caixa Controle: - Se a bola está na área ativa "comp";

Caixa Comportamentos: - Nome do comportamento "comp";
Caixa Ação: - Andar na velocidade 0.9;
Caixa Executor: - VaiPraBola;

--------------------- Rotinas Genéricas ---------------------
Caixa Rotinas: - Nome da rotina "VaiPraBola"
Caixa Ação: - Virar para bola;
Caixa Controle: - Se com pose da bola então
Caixa Ação:  - Chutar bola;
```

No quadro 12 apresenta-se o código fonte em LARF para efeito comparativo com o quadro 11.

Quadro 12 – Exemplo de implementação em LARF

```
DimensaoDoCampo(1,1);

DefinicaoDoJogador ( 1 )
Inicio
        AreaDeAtuacao("jogo",1,1);

        ControlePrincipal
        Inicio
                Se ( BolaNaArea("jogo") ) entao
                Ativa (Comp);
        Fim;

        Comportamento Comp
        Inicio
                Andar(0.5);
                chama(VaiPraBola);
        Fim;
Fim;
```

```
DefinicaoDoJogador ( 2 )
Inicio
	AreaDeAtuacao("jogo",1,1);

	ControlePrincipal
	Inicio
		Se ( BolaNaArea("jogo") ) entao
		Ativa (Comp);
	Fim;

	Comportamento Comp
	Inicio
		Andar(0.8);
		chama(VaiPraBola);
	Fim;
Fim;

DefinicaoDoJogador ( 3 )
Inicio
	AreaDeAtuacao("jogo",1,1);

	ControlePrincipal
	Inicio
		Se ( BolaNaArea("jogo") ) entao
		Ativa (Comp);
	Fim;

	Comportamento Comp
	Inicio
		Andar(1);
		chama(VaiPraBola);
	Fim;
Fim;

DefinicaoDoJogador ( 4 )
Inicio
	AreaDeAtuacao("jogo",1,1);

	ControlePrincipal
	Inicio
		Se ( BolaNaArea("jogo") ) entao
		Ativa (Comp);
	Fim;

	Comportamento Comp
	Inicio
		Andar(0.7);
		chama(VaiPraBola);
	Fim;
Fim;

DefinicaoDoJogador ( 5 )
Inicio
	AreaDeAtuacao("jogo",1,1);

	ControlePrincipal
	Inicio
		Se ( BolaNaArea("jogo") ) entao
		Ativa (Comp);
	Fim;

	Comportamento Comp
	Inicio
		Andar(0.9);
		chama(VaiPraBola);
	Fim;
Fim;

rotina VaiPraBola
Inicio
		VirarParaBola();
		Se ( ComPoseDaBola() ) entao
		Inicio
			ChutarBola();
		Fim;
Fim;
```

5.2.1 BNF DA LINGUAGEM LARF

A tabela 2 lista os nós terminais da linguagem, a tabela é utilizada para fazer a leitura da BNF da linguagem que é apresentada na tabela 3.

Tabela 2 – Símbolos terminais da linguagem

Não-terminal	Descrição do não-terminal
CONSTANT	Indica que neste local deve aparecer um valor numérico válido.
IDENTIFICADOR	Indica que neste ponto deverá ser informada uma seqüência de caracteres.
VIRGULA	Indica que deverá ser informado o símbolo “,”
PONTO_VIRGULA	Indica que deverá ser informado o símbolo “;”
ASPAS	Indica que deverá ser informado o símbolo “ ” ”
ABRE_PARENTESES	Indica que deverá ser informado o símbolo “(“
FECHA_PARENTESES	Indica que deverá ser informado o símbolo “)”
INICIO	Indica que deverá ser informado o símbolo “inicio”
FIM	Indica que deverá ser informado o símbolo “fim”
DEFINICAO_DO_JOGADOR	Indica que deverá ser informado o símbolo "DefinicaoDoJogador"
DIMENSAO_DO_CAMPO	Indica que deverá ser informado o símbolo "DimensaoDoCampo"
AREA_DE_ATUACAO	Indica que deverá ser informado o símbolo "AreaDeAtuacao"
CONTROLE_PRINCIPAL	Indica que deverá ser informado o símbolo "ControlePrincipal"
COMPORTAMENTO	Indica que deverá ser informado o símbolo "Comportamento"
ROTINA	Indica que deverá ser informado o símbolo "Rotina"
SE	Indica que deverá ser informado o símbolo "Se"
E	Indica que deverá ser informado o símbolo "e"
OU	Indica que deverá ser informado o símbolo "ou"
NÃO	Indica que deverá ser informado o símbolo "nao"
IGUAL	Indica que deverá ser informado o símbolo "="
MENOR	Indica que deverá ser informado o símbolo “<”
MAIOR	Indica que deverá ser informado o símbolo ">"
ENTAO	Indica que deverá ser informado o símbolo "entao"
ATIVA	Indica que deverá ser informado o símbolo "ativa"
CHAMA	Indica que deverá ser informado o símbolo "chama"
LADO_DIREITA	Indica que deverá ser informado o símbolo "Lado_Direito"
LADO_ESQUERDA	Indica que deverá ser informado o símbolo "Lado_Esquerdo"
LADO_FRENTE	Indica que deverá ser informado o símbolo "Lado_Frente"
LADO_ATRAZ	Indica que deverá ser informado o símbolo "Lado_Atraz"
BOLA_NA_AREA	Indica que deverá ser informado o símbolo "BolaNaArea"
COMPOSE_DA_BOLA	Indica que deverá ser informado o símbolo "ComposeDaBola"
PARCEIRO_COM_POSE_DA_BOLA	Indica que deverá ser informado o símbolo "ParceiroComPoseDaBola"
ADVERSARIO_COM_POSE_DA_BOLA	Indica que deverá ser informado o símbolo "AdversarioComPoseDaBola"
DISTANCIA_DA_BOLA	Indica que deverá ser informado o símbolo "DistanciaDaBola"
DISTANCIA_DO_JOGADOR	Indica que deverá ser informado o símbolo "DistanciaDoJogador"

ANGULO_DO_JOGADOR	Indica que deverá ser informado o símbolo "AnguloDoJogador"
DISTANCIA_DO_GOL	Indica que deverá ser informado o símbolo "DistanciaDoGol"
ANGULO_DO_GOL	Indica que deverá ser informado o símbolo "AnguloDoGol"
JOGADOR_NO_ÂNGULO	Indica que deverá ser informado o símbolo "JogadorNoAngulo"
LIMITE_DO_CAMPO	Indica que deverá ser informado o símbolo "LimiteDoCampo"
VIRAR_PARA_BOLA	Indica que deverá ser informado o símbolo "VirarParaBola"
IR_PARA_AREA	Indica que deverá ser informado o símbolo "IrParaArea"
GIRA_PARA_DIREITA	Indica que deverá ser informado o símbolo "GirarParaDireita"
GIRA_PARA_ESQUERDA	Indica que deverá ser informado o símbolo "GirarParaEsquerda"
ANDAR	Indica que deverá ser informado o símbolo "Andar"
PARAR	Indica que deverá ser informado o símbolo "Parar"
CHUTAR_BOLA	Indica que deverá ser informado o símbolo "ChutarBola"

Fonte: Schlei (2002).

A tabela 3 apresenta a BNF da linguagem, esta tabela foi gerada com a ferramenta *jjdoc* (veja a seção 4.2.2 que apresenta esta ferramenta), a tabela 3 segue as regras de escrita apresentada na tabela 2 que lista as regras da linguagem simbólica BNF.

Tabela 3 – BNF da linguagem LARF

verifica	::=	definicaoDoCampo (definicaoDoJogador)+ (rotinas)*
definicaoDoCampo	::=	<DIMENSAO_DO_CAMPO> <ABRE_PARENTESES> <CONSTANT> <VIRGULA> <CONSTANT> <FECHA_PARENTESES> <PONTO_VIRGULA>
definicaoDoJogador	::=	<DEFINICAO_DO_JOGADOR> <ABRE_PARENTESES> <CONSTANT> <FECHA_PARENTESES> <INICIO> (areaDeAtuacao)+ controlePrincipal (comportamento)+ <FIM> <PONTO_VIRGULA>
areaDeAtuacao	::=	<AREA_DE_ATUACAO> <ABRE_PARENTESES> <ASPAS> <IDENTIFICADOR> <ASPAS> <VIRGULA> <CONSTANT> <VIRGULA> <CONSTANT> <FECHA_PARENTESES> <PONTO_VIRGULA>
controlePrincipal	::=	<CONTROLE_PRINCIPAL> <INICIO> (seControlePrincipal)+ <FIM> <PONTO_VIRGULA>
seControlePrincipal	::=	<SE> expRelacionalLogica <ENTAO> <ATIVA> <ABRE_PARENTESES> <IDENTIFICADOR> <FECHA_PARENTESES> <PONTO_VIRGULA>
valor	::=	<CONSTANT>
	|	FuncaoNumerica
expRelacionalNumerica	::=	valor (<IGUAL> <IGUAL> | <MENOR> (<IGUAL> | <MAIOR>)? | <MAIOR> (<IGUAL>)?)? Valor
expRelacionalLogica	::=	<ABRE_PARENTESES> expRelacionalLogSimples (<OU> expRelacionalLogica)*

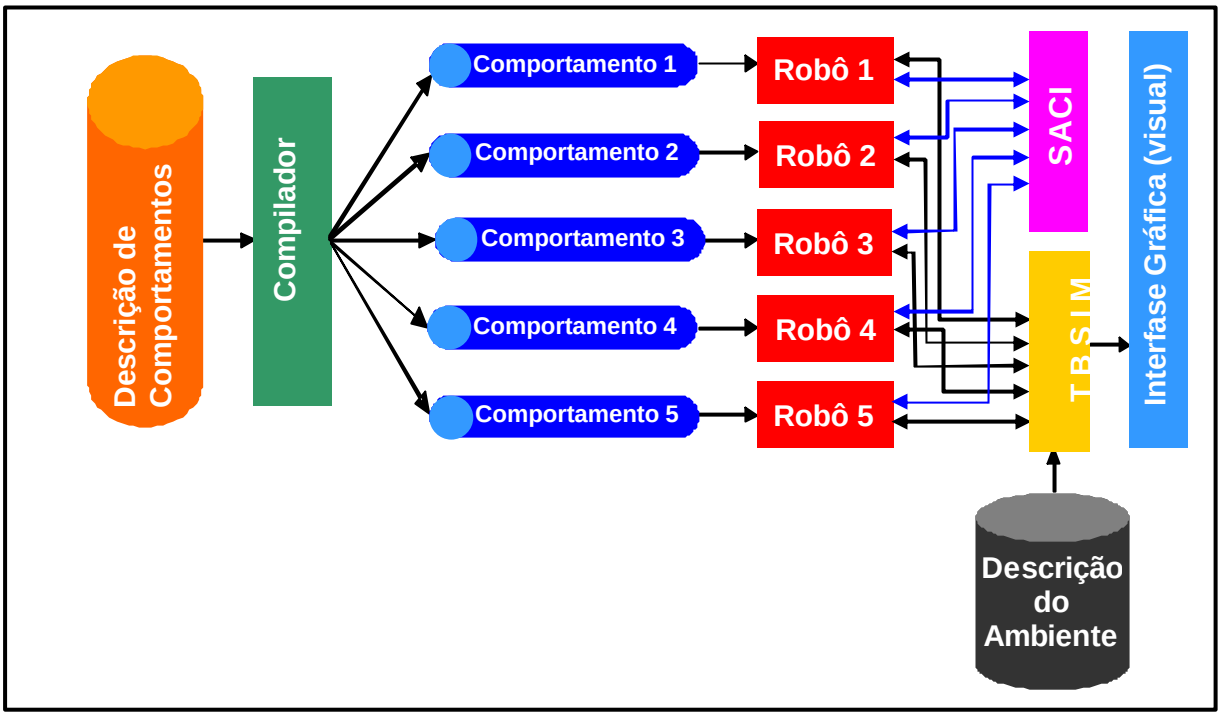
		<FECHA_PARENTESES>
expRelacionalLogSimples	::=	termoLogico (<E> termoLogico)*
termoLogico	::=	(<NÃO>)? (funcaoLogica | expRelacionalNumerica | expRelacionalLogica)
funcaoLogica	::=	<BOLA_NA_AREA> <ABRE_PARENTESES> <ASPAS> <IDENTIFICADOR> <ASPAS> <FECHA_PARENTESES>
	|	<COMPOSE_DA_BOLA> <ABRE_PARENTESES> <FECHA_PARENTESES>
	|	<PARCEIRO_COM_POSE_DA_BOLA> <ABRE_PARENTESES> <FECHA_PARENTESES>
	|	<ADVERSARIO_COM_POSE_DA_BOLA> <ABRE_PARENTESES> <FECHA_PARENTESES>
	|	<JOGADOR_NO_ANGULO> <ABRE_PARENTESES> valor <FECHA_PARENTESES>
funcaoNumerica	::=	<DISTANCIA_DA_BOLA> <ABRE_PARENTESES> <FECHA_PARENTESES>
	|	<DISTANCIA_DO_JOGADOR> <ABRE_PARENTESES> valor <FECHA_PARENTESES>
	|	<ANGULO_DO_JOGADOR> <ABRE_PARENTESES> valor <FECHA_PARENTESES>
	|	<DISTANCIA_DO_GOL>
	|	<ANGULO_DO_GOL> <ABRE_PARENTESES> <FECHA_PARENTESES>
	|	<LIMITE_DO_CAMPO> <ABRE_PARENTESES> lado <FECHA_PARENTESES>
lado	::=	<LADO_DIREITA>
	|	<LADO_ESQUERDA>
	|	<LADO_FRENTE>
	|	<LADO_ATRAZ>
comportamento	::=	<COMPORTAMENTO> <IDENTIFICADOR> <INICIO> (comandos)* <FIM> <PONTO_VIRGULA>
seNormal	::=	<SE> expRelacionalLogica <ENTAO> <INICIO> (comandos)* <FIM> <PONTO_VIRGULA>
comandos	::=	<CHAMA> <ABRE_PARENTESES> <IDENTIFICADOR> <FECHA_PARENTESES> <PONTO_VIRGULA>
	|	acao <PONTO_VIRGULA>
	|	SeNormal
acao	::=	<VIRAR_PARA_BOLA> <ABRE_PARENTESES> <FECHA_PARENTESES>
	|	<IR_PARA_AREA> <ABRE_PARENTESES> <ASPAS> <IDENTIFICADOR> <ASPAS> <FECHA_PARENTESES>
	|	<GIRA_PARA_DIREITA> <ABRE_PARENTESES> valor <FECHA_PARENTESES>
	|	<GIRA_PARA_ESQUERDA> <ABRE_PARENTESES> valor <FECHA_PARENTESES>
	|	<ANDAR> <ABRE_PARENTESES> valor <FECHA_PARENTESES>
	|	<PARAR> <ABRE_PARENTESES> <FECHA_PARENTESES>
	|	<CHUTAR_BOLA> <ABRE_PARENTESES> <FECHA_PARENTESES>
rotinas	::=	<ROTINA> <IDENTIFICADOR> <INICIO> (comandos)+ <FIM> <PONTO_VIRGULA>

Fonte: Schlei (2002).

6 VISÃO GERAL DO PROTÓTIPO

Para estender a linguagem LARF implementando comandos para realizar a comunicação entre robôs jogadores de futebol foi utilizada a ferramenta de comunicação entre agentes SACI. A figura 20 ilustra uma visão geral da linguagem estendida.

Figura 20 – Visão geral do protótipo



A ferramenta de comunicação entre agentes SACI, foi incorporada no protótipo tendo como função gerenciar a comunicação entre os robôs. Esta é executada paralelamente com o *TBSim*, recebendo, tratando e enviando as mensagens dos agentes.

6.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Como requisitos de *software*, a extensão da linguagem deve disponibilizar comandos para a comunicação entre os agentes robôs jogadores de futebol. Contudo, para isto foi preciso criar outros comandos para possibilitar a implementação da comunicação na linguagem. Comandos para realizar operações aritméticas, atribuição de valores para variáveis, criação de formatos de mensagens, obtenção de campos de uma mensagem e definição de um campo numa mensagem.

6.2 ESPECIFICAÇÃO E IMPLEMENTAÇÃO

Nesta seção é apresentada a especificação das classes dos comandos através de diagramas de classe, as regras de produção em BNF e as ações semânticas descritas em JavaCC, exceto o que está entre chaves que segue a sintaxe convencional do Java.

Na extensão da linguagem LARF foi definido uma interface genérica possuindo retorno do tipo *Object* em seu método *getValor()*, que substitui a interface *ExpressãoNumerica* da linguagem original definida por Schlei (2002) e dispensa a criação de interfaces para tipos *Message* do SACI e *String*. A figura 21 ilustra a interface da extensão.

Figura 21 – Interface *Expressao*

A tabela 4 lista os nós terminais da linguagem. Esta tabela é utilizada para fazer a leitura da BNF da linguagem.

Tabela 4 – Símbolos terminais adicionados na linguagem

Não-terminal	Descrição do não-terminal
STRING_LITERAL	Indica que neste ponto deverá ser informada uma seqüência de qualquer tipo de caracteres.
IMPRIMIR	Indica que deverá ser informado o símbolo "Imprimir"
SOMAR	Indica que deverá ser informado o símbolo "Somar"
SUBTRAIR	Indica que deverá ser informado o símbolo "Subtrair"
MULTIPLICAR	Indica que deverá ser informado o símbolo "Multiplicar"
DIVIDIR	Indica que deverá ser informado o símbolo "Dividir"
ATRIBUICAO	Indica que deverá ser informado o símbolo "="
GET_CAMPO_MSG	Indica que deverá ser informado o símbolo “GetCampoMsg”
CRIA_MSG	Indica que deverá ser informado o símbolo “CriaMsg”
SET_CAMPO_MSG	Indica que deverá ser informado o símbolo “SetCampoMsg”
RECEBE_MSG	Indica que deverá ser informado o símbolo “RecebeMsg
ENVIA_MSG	Indica que deverá ser informado o símbolo “EnviaMsg”

6.2.1 ATRIBUIÇÃO

Diante da necessidade de armazenar valores em variáveis, foi desenvolvido o comando de *Atribuicao*. Para isto foi preciso modificar a classe *RecebeRobo*, adicionando dois novos atributos, *memoria* e *mbox*. O atributo *memoria* é um contêiner do tipo *HashMap*, onde serão armazenados as variáveis e seus valores. Já o atributo *mbox*, é uma estrutura do tipo *Mail Box*

do SACI, que servirá para o agente enviar e receber mensagens. A classe *Atribuicao*, que executa o comando de atribuição, é composta por dois atributos, o atributo *var* do tipo *String* que é utilizado para guardar o identificador da variável, e o atributo *exp* do tipo *Expressao* que é utilizado para armazenar a expressão. O construtor *Atribuicao* recebe como parâmetro uma *String* e uma *Expressao*, e no método *executa* é calculado o valor da expressão e armazenado no contêiner *memoria*. A classe deste comando ainda possui o método *setMemoria* e o *setMbox*. Na sintaxe do comando deve vir um identificador seguido pelo sinal de atribuição e após este um *valor*. Este *valor* pode ser uma constante, uma *funcaoNumerica*, uma *funcaoMensagem*, uma *funcaoString* ou um identificador. Um exemplo de utilização do comando de atribuição é apresentado no quadro 13. A BNF do comando é apresentada na tabela 5, o diagrama de classes é apresentado na figura 22, as ações semânticas no quadro 14 (definidas conforme a sintaxe do JavaCC, e constituem-se de comandos na linguagem Java), e no quadro 15 apresenta-se o código da classe do comando de atribuição.

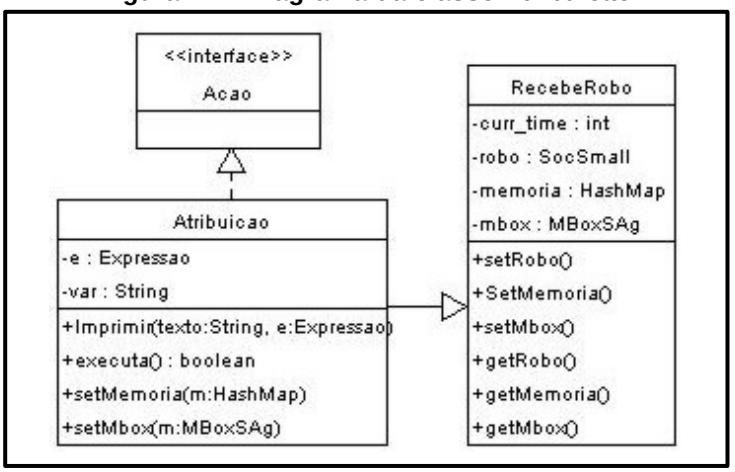
Quadro 13 – exemplo de atribuição em LARF

```
Comportamento Comp
Inicio
       m = CriaMsg("(tell :content ok)");
Fim;
```

Tabela 5 – BNF do comando de atribuição

acao	::=	<IDENTIFICADOR> <ATRIBUICAO> valor
valor	::=	<CONSTANTE>
	|	funcaoNumerica
	|	funcaoMensagem
	|	funcaoString
	|	<IDENTIFICADOR>

Figura 22 – Diagrama da classe *Atribuicao*



Quadro 14 – Ações semânticas do comando de atribuição

```
< IDENTIFICADOR >
      {
            String id = token.image;
      }
      < ATRIBUICAO > valor()
      {
            Atribuicao a = new Atribuicao(id,(Expressao)stExpNumerica.pop());
            rotinaAtual.addAcao(a);
      }
```

Quadro 15 – Implementação da classe *Atribuicao*

```
import java.io.*;
import java.util.*;
import saci.*;

public class Atribuicao extends RecebeRobo implements Acao, Serializable {

    private String var;
    private Expressao exp;

    public Atribuicao(String texto, Expressao e) {
        var = texto;
        exp = e;
    }

    public boolean executa() {
        Object resultado = exp.getValor();
        if (resultado != null) {
            memoria.put(var, resultado);
        }
        return true;
    }

    public void setMemoria(HashMap m) {
        super.setMemoria(m);
        if (exp != null) {
            ((Valor)exp).setMemoria(m);
        }
    }

    public void setMbox(MBoxSAg m) {
        super.setMbox(m);
        if (exp != null) {
            ((Valor)exp).setMbox(m);
        }
    }
}
```

Para possibilitar a realização de operações aritméticas na linguagem, foram implementadas quatro funções, respectivamente as quatro operações da aritmética: somar, subtrair, multiplicar e dividir. Estas classes possuem um construtor com dois parâmetros do tipo *Expressao*, *esq* e *dir*, e o método *getValor()* que implementa a interface *Expressao*. O atributo *esq*, guarda a parte da esquerda da expressão e o atributo *dir* a parte da direta. Estes por sua vez serão submetidos a um operador aritmético e o seu produto é retornado para o método *getVa-*

lor. Na BNF do comando é aceito o nome do comando (Somar, Subtrair, Multiplicar ou Dividir), abre parênteses, *valor*, vírgula, *valor*, fecha parênteses. No quadro 16 apresenta-se um exemplo, na tabela 6 é apresentado a BNF dos comandos, na figura 23 apresentam-se os diagramas de classe, no quadro 17 as ações semânticas, e no quadro 18 a implementação da classe do comando.

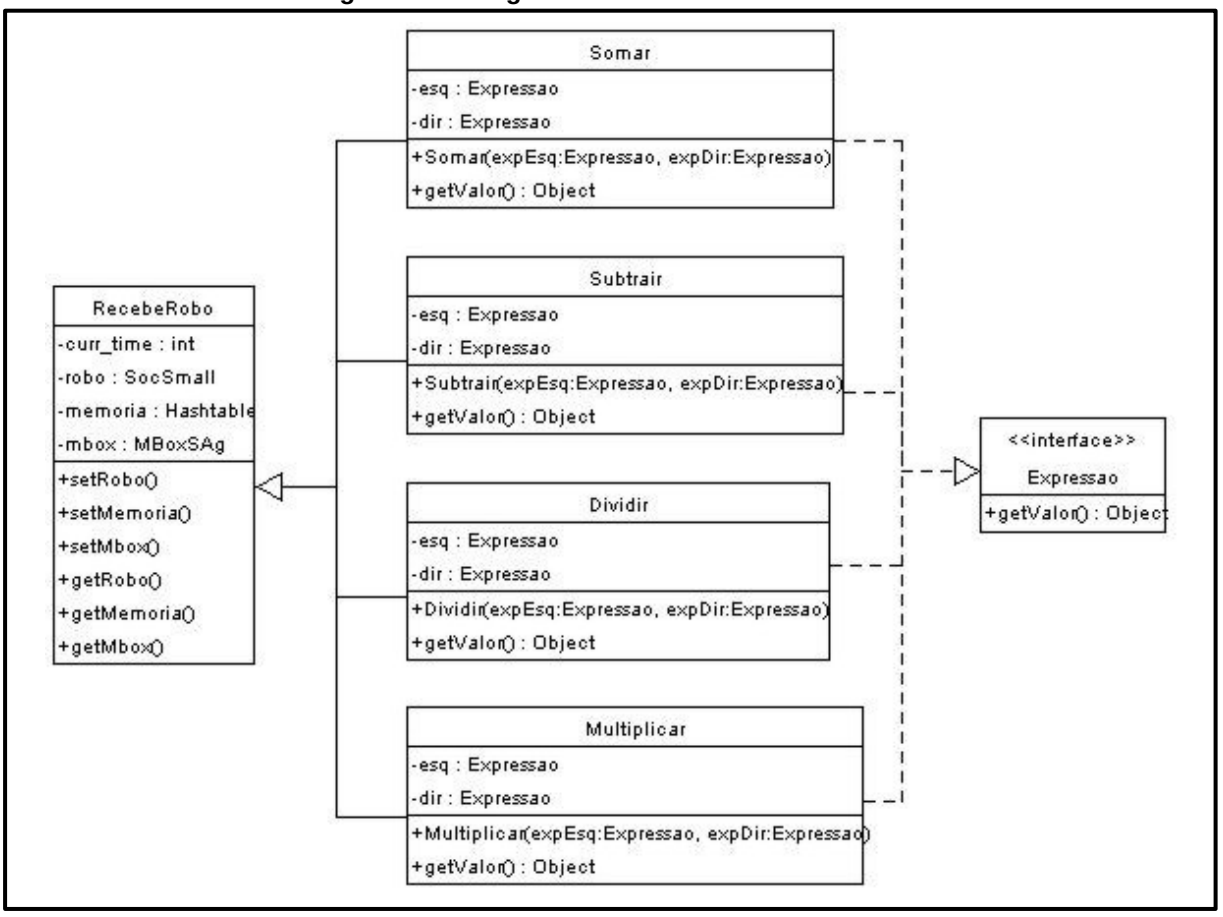
Quadro 16 – Exemplo das operações aritméticas em LARF

```
Comportamento Comp
Inicio
       x = Somar(6,4);
Fim;
```

Tabela 6 – BNF das operações aritméticas

funcaoNumerica	::=	<SOMAR> <ABRE_PARENTESES> valor <VIRGULA> valor <FECHA_PARENTESES>
	|	<SUBTRAIR> <ABRE_PARENTESES> valor <VIRGULA> valor <FECHA_PARENTESES>
	|	<MULTIPLICAR> <ABRE_PARENTESES> valor <VIRGULA> valor <FECHA_PARENTESES>
	|	<DIVIDIR> <ABRE_PARENTESES> valor <VIRGULA> valor <FECHA_PARENTESES>

Figura 23 – Diagramas das classes aritméticas



Quadro 17 – Ações semânticas das operações aritméticas

```
< SOMAR > < ABRE_PARENTESES > valor() < VIRGULA > valor() < FECHA_PARENTESES >
       {
              Somar soma = new Somar((Expressao)stExpNumerica.pop(),
                                     (Expressao)stExpNumerica.pop());
              vl.setExp(soma);
       }
|
< SUBTRAIR > < ABRE_PARENTESES > valor() < VIRGULA > valor() < FECHA_PARENTESES >
{
              Subtrair sub = new Subtrair((Expressao)stExpNumerica.pop(),
                                          (Expressao)stExpNumerica.pop());
              vl.setExp(sub);
       }
|
< MULTIPLICAR > <ABRE_PARENTESES> valor() < VIRGULA > valor() <FECHA_PARENTESES>
       {
              Multiplicar mult = new Multiplicar((Expressao)stExpNumerica.pop(),
                                                 (Expressao)stExpNumerica.pop());
              vl.setExp(mult);
       }
|
< DIVIDIR > < ABRE_PARENTESES > valor() < VIRGULA > valor() < FECHA_PARENTESES >
       {
              Dividir div = new Dividir((Expressao)stExpNumerica.pop(),
                                        (Expressao)stExpNumerica.pop());
              vl.setExp(div);
       }
```

Quadro 18 – Implementação da classe *Somar*

```
import java.io.*;

public class Somar extends RecebeRobo implements Expressao, Serializable {

       private Expressao esq, dir;

       public Somar(Expressao expDir, Expressao expEsq) {
              dir = expDir;
              esq = expEsq;
       }

       public Object getValor() {
              Float f1 = (Float)esq.getValor();
              Float f2 = (Float)dir.getValor();
              float result = f1.floatValue() + f2.floatValue();
              return new Float(result);
       }
}
```

6.2.3 IMPRIMIR

Foi desenvolvido o comando *Imprimir* para possibilitar exibir informações na tela. A classe desse comando possui dois construtores, um que recebe uma *Expressao*, e o outro uma *String*. O método *executa* implementa a interface *Acao* onde é impresso o conteúdo da expressão. E o método *setMemoria*, onde é salvo o lado direito da expressão no contêiner *memoria*.

A sintaxe do comando segue a seguinte estrutura; comando, abre parênteses, *String* ou *valor*, fecha parênteses. No quadro 19 é apresentado um exemplo de utilização do comando, na tabela 7 sua respectiva BNF, na figura 24 é apresentado o diagrama de classes, no quadro 20 as ações semânticas e no quadro 21 a implementação da classe.

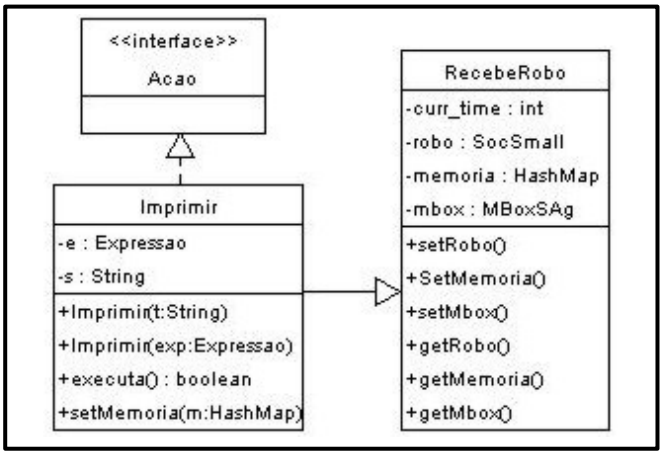
Quadro 19 – Exemplo do comando *Imprimir*

```
Comportamento Comp
Inicio
       Imprimir("Jogador 01");
Fim;
```

Tabela 7 – BNF do comando *Imprimir*

acao	::=	<IMPRIMIR> <ABRE_PARENTESES> (<STRING_LITERAL> | valor) <FECHA_PARENTESES>

Figura 24 – Diagrama da classe *Imprimir*



Quadro 20 – Ações semânticas do comando *Imprimir*

```
< IMPRIMIR > < ABRE_PARENTESES >
       (
              < STRING_LITERAL >
              {
                     texto = token.image.substring(1,token.image.length()-1);
                     Imprimir i = new Imprimir(texto);
                     rotinaAtual.addAcao(i);
              }
       |
              valor()
              {
                     Imprimir imp =  new Imprimir((Expressao)stExpNumerica.pop());
                     rotinaAtual.addAcao(imp);
              }
       )
< FECHA_PARENTESES >
```

Quadro 21 – Implementação da classe *Imprimir*

```
import java.io.*;
import java.util.*;

public class Imprimir extends RecebeRobo implements Acao, Serializable {

    private Expressao e;
    private String s = null;

    public Imprimir(String t) {
        s = t;
    }

    public Imprimir(Expressao exp) {
        e = exp;
    }

    public boolean executa() {
        if(s != null)
            System.out.println(s);
        else
            System.out.println(e.getValor());
        return true;
    }

    public void setMemoria(HashMap m) {
        super.setMemoria(m);
        if (e != null) {
            ((Valor)e).setMemoria(m);
        }
    }
}
```

6.2.4 CRIA MENSAGEM

Para criar as mensagens seguindo o formato definido pelo KQML e retornar o tipo *Message* do SACI, foi criado o comando *CriaMsg*. Esta classe estende da classe *RecebeRobo*, possui um construtor com um parâmetro do tipo *String*, e um método *getValor()* que implementa a interface *Expressao*. Sua sintaxe é composta por comando, abre parênteses, *String*, fecha parênteses. No quadro 22 apresenta-se um exemplo, na tabela 8 apresenta-se a BNF do comando, na figura 25 o diagrama da classe *CriaMsg*, no quadro 23 as ações semânticas e no quadro 24 a implementação da classe.

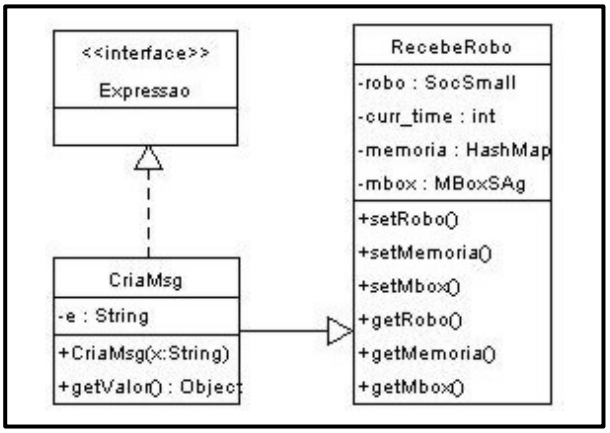
Quadro 22 – exemplo do comando *CriaMsg*

```
Comportamento Comp
Inicio
       m = CriaMsg("(tell :content ok)");
Fim;
```

Tabela 8 – BNF do comando *CriaMsg*

funcaoMensagem	::=	<CRIA_MSG> <ABRE_PARENTESES> valor <FECHA_PARENTESES>

Figura 25 – Diagrama da classe *CriaMsg*



Quadro 23 – Ações semânticas do comando *CriaMsg*

```
< CRIA_MSG > < ABRE_PARENTESES > < STRING_LITERAL >
{
       s = token.image.substring(1,token.image.length()-1);
       CriaMsg cm = new CriaMsg(s);
       vl.setExp(cm);
}
       < FECHA_PARENTESES >
```

Quadro 24 – Implementação da classe *CriaMsg*

```
import java.io.*;
import saci.*;

public class CriaMsg extends RecebeRobo implements Expressao, Serializable {

       private String e;

       public CriaMsg(String x) {
              e = x;
       }

       public Object getValor() {
              return new Message(e);
       }
}
```

6.2.5 GET CAMPO MENSAGEM

O comando *GetCampoMsg* foi desenvolvido com o intuito de possibilitar a obtenção de um campo da mensagem de acordo com a necessidade. A classe deste comando estende a classe *RecebeRobo* e implementa a interface *Expressao*. Possui um construtor com dois parâmetros do tipo *Expressao*, e o método *getValor()*. Sua sintaxe aceita a seguinte estrutura, mo-

me do comando, abre parênteses, uma expressão, virgula, outra expressão, fecha parênteses. No quadro 25 é apresentado um exemplo de utilização do comando, na tabela 9 a BNF, na figura 26 o diagrama da classe *GetCampoMsg*, no quadro 26 as ações semânticas e no quadro 27 a implementação da classe.

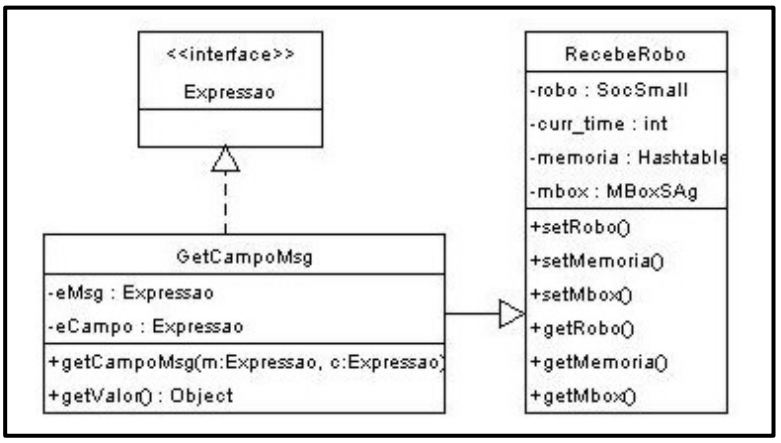
Quadro 25 – exemplo do comando GetCampoMsg

```
Comportamento Comp
Inicio
       m = GetCampoMsg(m,performativa);
Fim;
```

Tabela 9 – BNF do comando *GetCampoMsg*

funcaoString	::=	<GET_CAMPO_MSG> <ABRE_PARENTESES> valor < VIRGULA> valor <FECHA_PARENTESES>

Figura 26 – Diagrama da classe *GetCampoMsg*



Quadro 26 – Ações semânticas do comando *GetCampoMsg*

```
< GET_CAMPO_MSG > < ABRE_PARENTESES > valor() < VIRGULA > valor()
       {
       GetCampoMsg gcm = new GetCampoMsg((Expressao)stExpNumerica.pop(),
                                         (Expressao)stExpNumerica.pop());
       vl.setExp(gcm);
       }
< FECHA_PARENTESES >
```

Quadro 27 – Implementação da classe *GetCampoMsg*

```
import java.io.*;
import java.util.*;
import saci.*;

public class GetCampoMsg extends RecebeRobo implements Expressao, Serializable {

	private Expressao eMsg;
	private Expressao eCampo;

	public GetCampoMsg(Expressao m, Expressao c) {
		eMsg   = m;
		eCampo = c;
	}

	public Object getValor() {
		Message m = (Message)eMsg.getValor();
		return (String)m.get(eCampo);
	}
}
```

6.2.6 SET CAMPO MENSAGEM

Para atribuir um valor a um campo de uma mensagem foi desenvolvido o comando *SetCampoMsg*. A classe que implementa este comando estende a classe *RecebeRobo*, implementa a interface *Acao*, possui um construtor com três parâmetros do tipo *Expressao* e o método *executa()*. No quadro 28 é apresentado um exemplo, na tabela 10 encontra-se a BNF do comando, na figura 27 o diagrama de classes, no quadro 29 as ações semânticas e no quadro 30 a implementação da classe do comando.

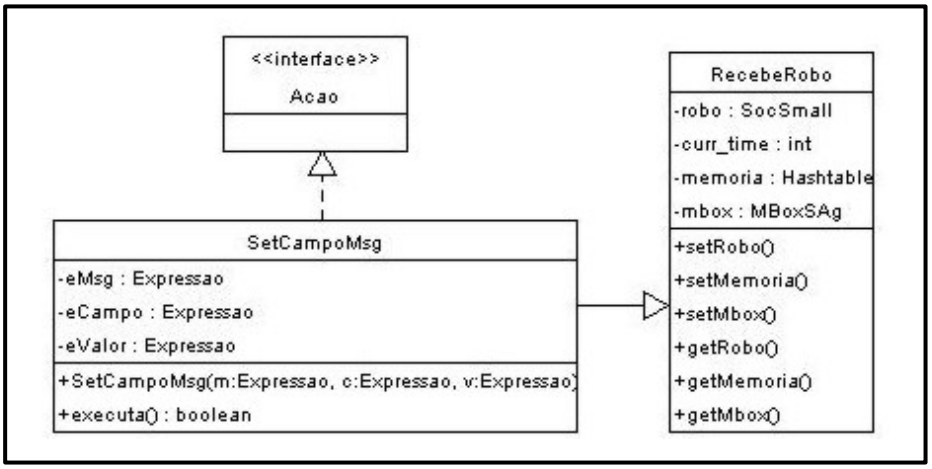
Quadro 28 – exemplo do comando *SetCampoMsg*

```
Comportamento Comp
Inicio
	SetCampoMsg(m,"performative","ask");
Fim;
```

Tabela 10 – BNF do comando *SetCampoMsg*

acao	::=	<SET_CAMPO_MSG> <ABRE_PARENTESES> valor < VIRGULA> valor < VIRGULA> valor <FECHA_PARENTESES>

Figura 27 – Diagrama da classe *SetCampoMsg*



Quadro 29 – Ações semânticas do comando *SetCampoMsg*

```
< SET_CAMPO_MSG > < ABRE_PARENTESES > valor() < VIRGULA > valor() < VIRGULA >
valor()
       {
              SetCampoMsg scm = new SetCampoMsg((Expressao)stExpNumerica.pop(),
                                                (Expressao)stExpNumerica.pop(),
                                                (Expressao)stExpNumerica.pop());
              rotinaAtual.addAcao(scm);
       }
< FECHA_PARENTESES >
```

Quadro 30 – Implementação da classe *SetCampoMsg*

```
import java.io.*;
import saci.*;

public class SetCampoMsg extends RecebeRobo implements Acao, Serializable {

       private Expressao eMsg;
       private Expressao eCampo;
       private Expressao eValor;

       public SetCampoMsg(Expressao m, Expressao c, Expressao v) {
              eMsg   = m;
              eCampo = c;
              eValor = v;
       }

       public boolean executa() {
              Message m = (Message)eMsg.getValor();
              m.put((String)eCampo.getValor(),eValor.getValor());
              return true;
       }
}
```

6.2.7 ENVIA MENSAGEM

Para realizar o envio de mensagens foi desenvolvido o comando *EnviaMsg*. Este possui em sua classe um construtor com um parâmetro do tipo *Expressao*. Esta estende a classe *RecebeRobo,* implementa a interface *Acao*. No quadro 31 apresenta-se um exemplo, na figura 28 o diagrama da classe *EnviarMsg*, na tabela 11 a BNF do comando, no quadro 32 as ações semânticas e no quadro 33 a implementação da classe.

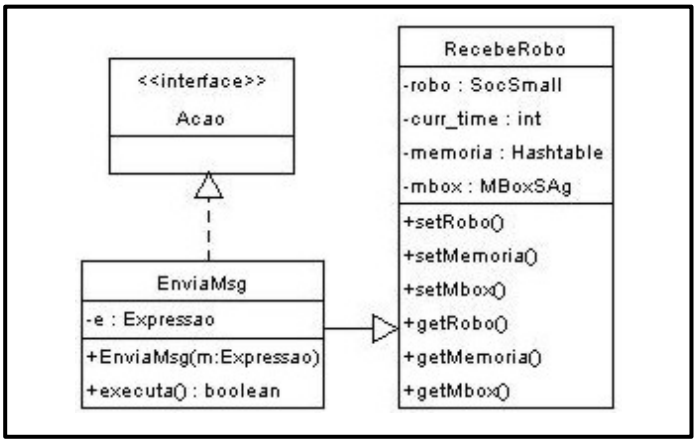
Quadro 31 – Exemplo do comando *EnviaMsg*

```
Comportamento Comp
Inicio
       EnviaMsg(CriaMsg("(tell :content ok)"));
Fim;
```

Tabela 11 – BNF do comando *EnviaMsg*

acao	::=	<ENVIA_MSG> <ABRE_PARENTESES> valor <FECHA_PARENTESES>

Figura 28 – Diagrama da classe *EnviarMsg*



Quadro 32 – Ações semânticas do comando *EnviaMsg*

```
< ENVIA_MSG > < ABRE_PARENTESES > valor()
       {
              EnviaMsg em = new EnviaMsg((Expressao)stExpNumerica.pop());
              rotinaAtual.addAcao(em);
       }
< FECHA_PARENTESES >
```

Quadro 33 – Implementação da classe *EnviaMsg*

```
import java.io.*;
import saci.*;

public class EnviaMsg extends RecebeRobo implements Acao, Serializable {

    private Expressao e;

    public EnviaMsg(Expressao m) {
        e = m;
    }

    public boolean executa() {
        try {
            this.mbox.sendMsg((Message)e.getValor());
        } catch (Exception e) {
            System.err.println("1: Error "+e);
            e.printStackTrace();
        }
        return true;
    }
}
```

6.2.8 RECEBE MENSAGEM

O recebimento das mensagens é feito através do comando *RecebeMsg*. A classe deste comando estende a classe *RecebeRobo*, implementa a interface *Expressao* e possui o método *getValor()*.No quadro 34 é apresentado um exemplo, a tabela 12 a BNF do comando, na figura 29 ilustra o diagrama da classe *RecebeMsg*, o quadro 35 as ações semânticas e no quadro 36 a implementação da classe.

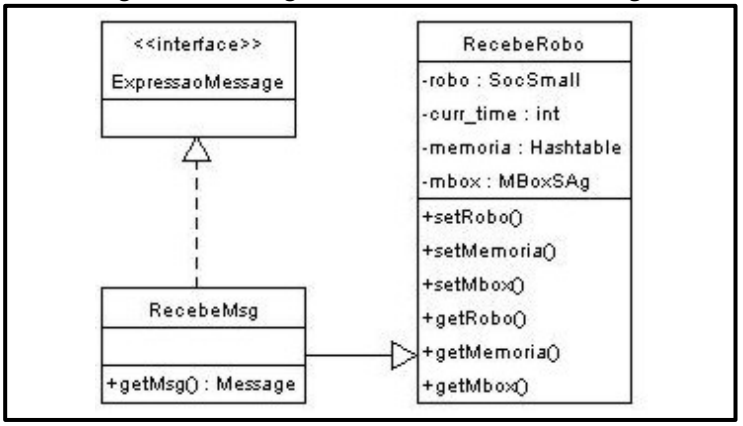
Quadro 34 – exemplo do comando RecebeMsg

```
Comportamento Comp
Inicio
       m = RecebeMsg();
Fim;
```

Tabela 12 – BNF do comando *ReceberMsg*

funcaoMensagem	::=	<RECEBE_MSG> <ABRE_PARENTESES> <FECHA_PARENTESES>

Figura 29 – Diagrama da classe *RecebeMsg*



Quadro 35 – Ações semânticas do comando *RecebeMsg*

```
< RECEBE_MSG > < ABRE_PARENTESES > < FECHA_PARENTESES >
       {
              vl.setExp(new RecebeMsg());
       }
```

Quadro 36 – Implementação da classe *RecebeMsg*

```
import java.io.*;
import saci.*;

public class RecebeMsg extends RecebeRobo implements Expressao, Serializable {

       public Object getValor() {
              Message m = mbox.receive();
              return m;
       }
}
```

6.3 TÉCNICAS E FERRAMENTAS UTILIZADAS

Os softwares utilizados neste trabalho foram desenvolvidos em Java, portanto no protótipo desenvolvido também foi utilizado como linguagem de programação a linguagem Java, em sua versão SDK 1.4.0. A versão do gerador de *parser* JavaCC utilizada para a extensão do compilador foi a 2.1 e o ambiente *TeamBots* o qual foi utilizado na simulação dos robôs foi a 2.0. O ambiente de desenvolvimento utilizado para a implementação das classes foi o JCreator versão 2.0, atendendo as necessidades de implementação do trabalho. Como ferramenta de comunicação entre agentes foi utilizada a ferramenta SACI. Para fazer a modelagem em UML das classes foi utilizada a ferramenta "Poseidon for UML Community Edition", versão 1.4. disponibilizando todos os recursos necessários para a modelagem em UML.

6.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Para demonstrar a funcionalidade dos comandos criados para realizar a comunicação entre os robôs, é apresentado no quadro 37 um exemplo de código em LARF.

Quadro 37 – Exemplo de implementação da comunicação em LARF

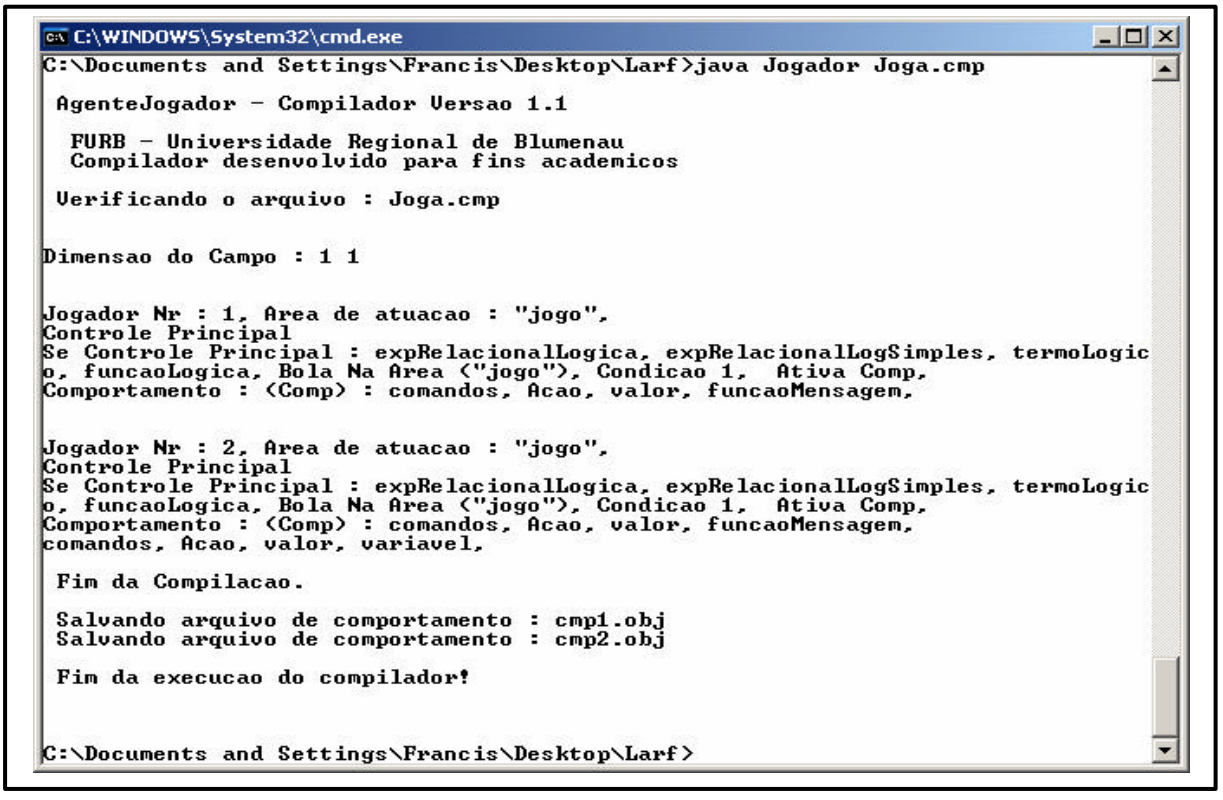
```
01 DimensaoDoCampo(1,1);
02
03 DefinicaoDoJogador ( 1 )
04 Inicio
05     AreaDeAtuacao("jogo",1,1);
06
07     ControlePrincipal
08     Inicio
09            Se ( BolaNaArea("jogo") ) entao
10            Ativa (Comp);
11     Fim;
12
```

```
13     Comportamento Comp
14     Inicio
15            EnviaMsg(CriaMsg("(tell :content oi :receiver Jogador_2)"));
16     Fim;
17 Fim;
18
19 DefinicaoDoJogador ( 2 )
20 Inicio
21     AreaDeAtuacao("jogo",1,1);
22
23     ControlePrincipal
24     Inicio
25            Se ( BolaNaArea("jogo") ) entao
26            Ativa (Comp);
27     Fim;
28
29     Comportamento Comp
30     Inicio
31            m = RecebeMsg();
32            Imprimir(m);
33     Fim;
34 Fim;
```

Na linha 15 através do comando *CriaMsg* é gerada a mensagem que será enviada pelo comando *EnviaMsg*. Na linha 31 é armazenada na variável *m* a mensagem recebida através do comando *RecebeMsg*. E na linha 32 é impresso na tela o conteúdo da variável *m*, isto é, o conteúdo da mensagem recebida.

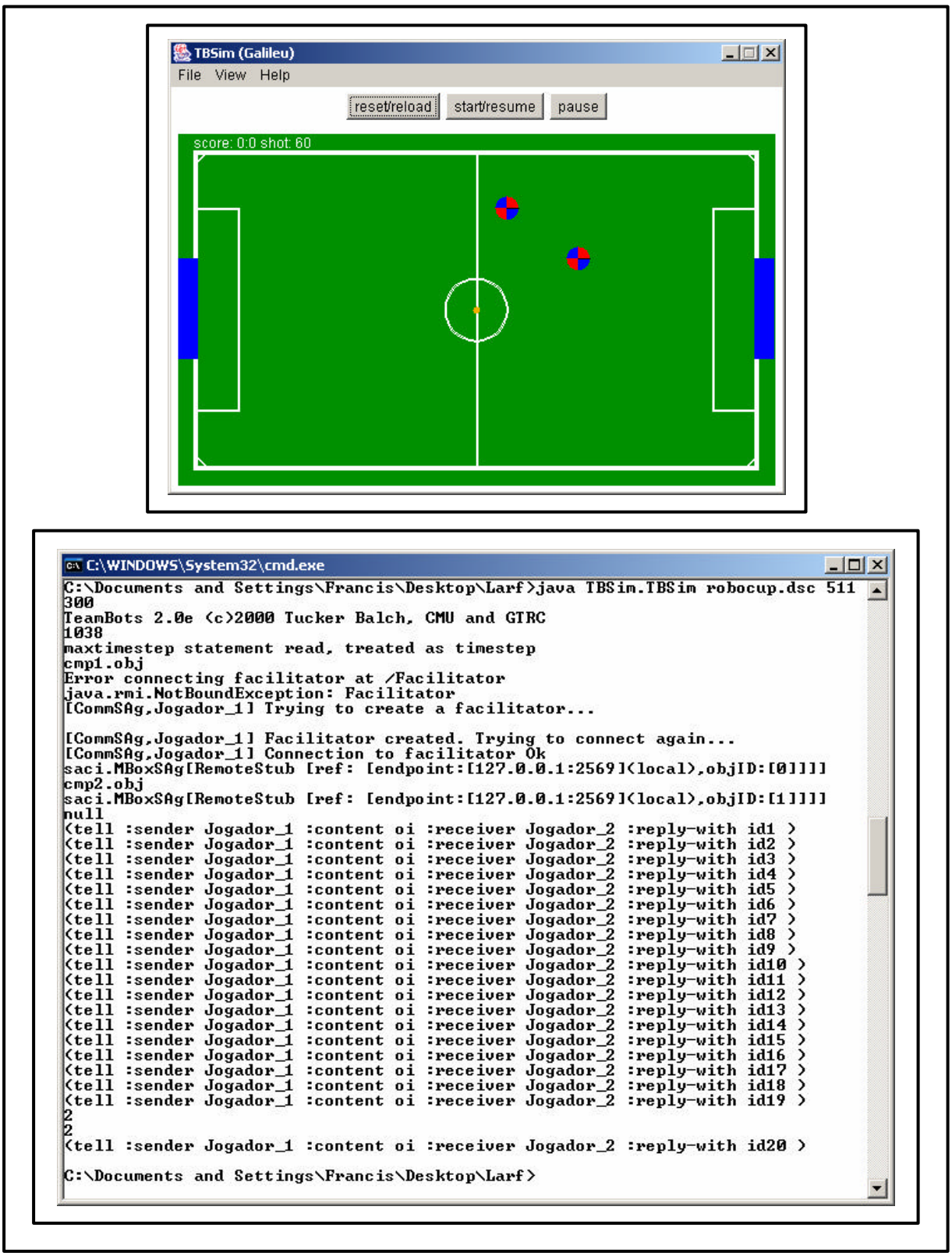
Ao compilar o código do quadro 36 são exibidas na tela as informações de compilação conforme ilustrado na figura 30.

Figura 30 – Execução do compilador



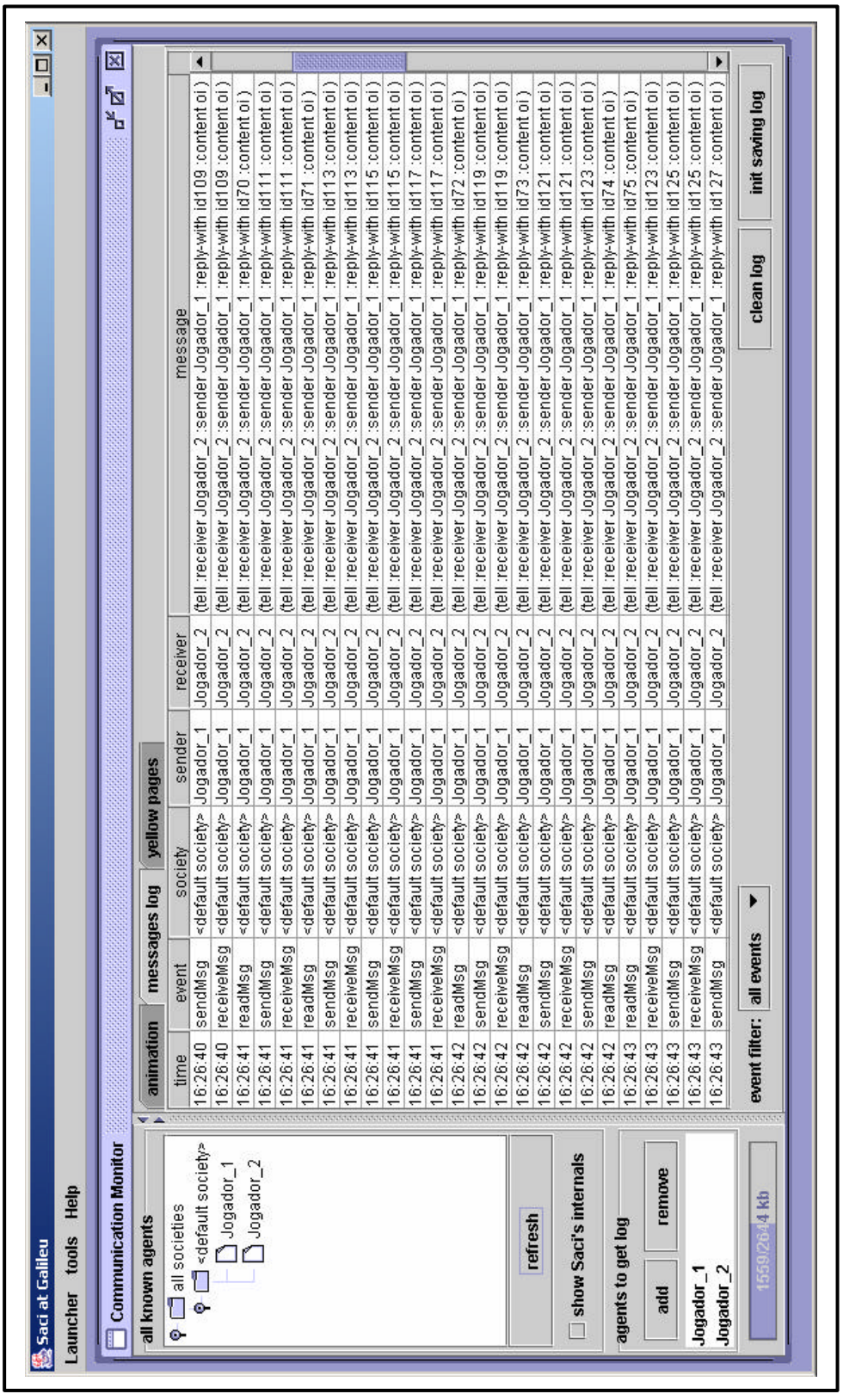
O resultado da implementação dos comandos de comunicação descritos nas linhas 15, 31 e 32 do quadro 36 é apresentado na figura 31.

Figura 31 – Execução da implementação



Para comprovar a realização da comunicação entre os dois robôs, na figura 32 é apresentado o *log* de execução da comunicação, gerado através da ferramenta de monitoramento de comunicação do SACI.

Figura 32 – Log de execução do SACI



6.5 PROBLEMAS E DIFICULDADES

Como no currículo que cursei não havia a disciplina de compiladores, obtive dificuldades para entender o código já implementado, por isso foi necessário fazer um estudo em diversos livros sobre o assunto e com base nestes obter o mínimo conhecimento necessário para o desenvolvimento dos novos comandos da linguagem.

A linguagem LARF por ter sido concebida num TCC em forma de protótipo, não possui a documentação esperada o que dificultou o entendimento da mesma.

6.6 RESULTADOS E DISCUSSÃO

Considerando que o objetivo deste trabalho é estender a linguagem LARF adicionando comandos para a comunicação entre os agentes robôs jogadores de futebol, o objetivo principal do trabalho foi alcançado. Contudo foram feitas várias melhorias no que diz respeito a linguagem LARF, como a simplificação de várias interfaces para apenas uma que retorna um objeto, a criação de funções aritméticas, impressão na tela, e criação de formatos de mensagens, facilitando a extensão da linguagem em trabalhos futuros.

7 CONCLUSÕES

Neste trabalho estendeu-se a linguagem LARF inserindo comandos para enviar e receber mensagens KQML, obter um campo de uma mensagem, definir um campo de uma mensagem, criar mensagens no formato KQML, exibir informações na tela, atribuir valores para variáveis e efetuar operações aritméticas. Possibilitou-se também a utilização de variáveis na linguagem, contudo estas não possuem checagem de tipos.

Na extensão da linguagem foi substituída a interface *ExpressaoNumerica* que retornava um tipo numérico para uma interface denominada simplesmente de *Expressao* que retorna um objeto, tornando sua utilização mais abrangente.

Como as ferramentas utilizadas neste trabalho e a própria linguagem LARF foram desenvolvidas na linguagem Java, esta se mostrou bastante própria na criação dos comandos da linguagem.

A utilização da ferramenta JavaCC forneceu suporte esperado na extensão do compilador para a linguagem formalizada. Mostrou-se um excelente gerador de código Java e um ótimo compilador o qual detecta casos de ambigüidade na linguagem formalizada, indicando o local para a resolução da ambigüidade encontrada.

7.1 EXTENSÕES

As possíveis extensões provenientes deste trabalho estão listadas abaixo:

a) definir comandos para possibilitar a coordenação dos times;

b) inserir chamadas de funções na linguagem permitindo, por exemplo, definir funções que retornem se o objetivo foi alcançado, ou retornando o jogador que está com a posse da bola, etc;

c) fazer robôs físicos e controlar estes a partir da linguagem LARF;

d) estender a linguagem para que, além da definição de comportamentos de um jogador, estratégias globais de um time possam ser definidas na linguagem. Desta forma poder-se-ia, por exemplo, definir uma estratégia do tipo; 1 jogador na defesa, 2 jogadores no meio campo e 2 jogadores no ataque, sendo que se o jogo estiver sendo ganho, possibilitar a mudança de estratégia para mais defensiva colocando 2 jogadores na defesa, 2 jogadores no meio campo e apenas 1

no ataque;

e) definir tipos para as variáveis da linguagem, para possibilitar a identificação de tipos incompatíveis, necessário para isso acrescentar a checagem de tipos;

f) refatorar o código aplicando padrões de projeto.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compilers:** principles, techniques and tools. Murray Hill, New Jersey: Addison-Wesley Publishing Company, 1988.

BALCH, Tucker. **TeamBots**, Pittsburgh, 2000. Disponível em: <http://www2.cs.cmu.edu/~trb/TeamBots/index.html> ou <http://www.teambots.org>. Acesso em: 19 fev. 2003.

BORDINI, Rafael Heitor; VIEIRA, Renata; MOREIRA, Álvaro Freitas. **Fundamentos de sistemas multiagentes,** Porto Alegre, out. 2002. Disponível em: <http://www.inf.ufrgs.br/~bordini/Publications/JAI1-2001/>. Acesso em: 19 fev. 2003.

DAVID, C. Pellejero *et* al. **FURGBOL - Futebol de robôs**, Rio Grande, RS, dez. 2001. Disponível em: <http://www.ecomp.furg.br/ecompcricte/2001/Resege36.pdf >. Acesso em: 16 jan. 2003.

HÜBNER, Jomi Fred. **SACI**: simple agent communication infrastructure, São Paulo, mar. 2001. Disponível em: <http://www.lti.pcs.usp.br/saci>. Acesso em: 15 fev. 2003.

JAVACC. **Java Compiler Compiler:** the Java parser generator, Disponível em: <http://www.webgain.com/products/java_cc/ >. Acesso em: 20 fev. 2003.

LCMI – Laboratório de Controle e Microinformática. **Departamento de automação e sistemas,** Florianópolis, out. 2000. Disponível em: <http://www.lcmi.ufsc.br/ufscteam/index.html>. Acesso em: 2 dez. 2002.

JOSÉ Neto, João. **Introdução a compilação**. Rio de Janeiro: Livros Técnicos e Científicos, 1987.

MENESES, Eudenia Xavier; Silva, Flávio Soares Corrêa da. **Jornada de atualização em inteligência artificial.** 2001.

PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. **Implementação de linguagens de programação:** compiladores. Porto Alegre: Sagra Luzzato, 2001.

SCHLEI, Edson Elmar. **Uma linguagem para definição de estratégias de controle de times de robôs jogares de futebol em um ambiente simulado.** 2002. 77 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

TAVARES, Orivaldo de Lira. **Javacc - Gerador de Parsers Java**. Vitória - ES, jan. 2001. Disponível em <http://www.inf.ufes.br/~tavares/labcomp2000/javacc.html> Acesso em: 11 mar. 2003.

WEISS, Gerhard. **Multiagent systems**: a modern approach to distributed artificial intelligence. Massachussetts: Massachussetts Institute of Technology, 1999.

WOOLDRIDGE, Michael J. **An introduction to multiagent systems.** New York: John Wiley & Sons, 2002.