Relatório de TFC
do curso de
Licenciatura em Engenharia Informática e de Computadores
(LEIC)

Ano Lectivo 2002 / 2003

Departamento
de Engenharia
Informática

N.º da Proposta: 147

Título: Sistema de Projecto e Controlo de Missão de uma Equipa de Robots Cooperantes

Professor Orientador:

Pedro Lima _____

Co-Orientador:

Nome co-orientador (nome Entidade Externa) _____

Professor Acompanhante:

Nome professor acompanhante _____

Alunos:

Nº 41 327, Bruno Machado _____

Nº 45 435, Cláudio Cardoso _____

Bruno Machado, Cláudio Cardoso

Agradecimentos

Quem agradece o que tem...

...merece o que não tem.

Este trabalho não seria possível sem o apoio das nossas famílias prestado ao longo de todo o curso superior e é a eles que dedicamos este esforço final.

Queremos agradecer também o incentivo das namoradas, amigos e colegas.

Ao longo destes anos frequentando o curso de Engenharia Informática e de Computadores foram inúmeros os projectos realizados, mas este foi sem duvida o que nos deu mais prazer em concluir, porque aliados à constante tempo foi conseguido por em prática os conceitos teórico-práticos e conseguir relaciona-los com o já existente projecto *ISocRob*.

Agradecemos a disponibilidade dos professores e colegas de grupo (*ISocRob* e *Rescue*) que foi sem dúvida uma preciosa ajuda para a atingir os objectivos traçados.

Para todos com um abraço especial

Lisboa, 10 de Julho de 2003

Bruno e Cláudio.

Resumo

Com a necessidade de aliar a grandeza de um projecto robótico com a estruturação de código inerente aos robots, construí-se uma ferramenta capaz de produzir código C a partir de um ambiente gráfico de desenho. A ideia de partir para um sistema desta natureza face aos já existentes (*Charon* e *MissionLab*), foi produzir um motor que satisfizesse os requisitos necessários para uma geração de código intrinsecamente adaptada à filosofia da equipa de projecto.

Este trabalho visa simplificar a codificação de comportamentos mantendo a coerência e estrutura. Para a construção de uma solução foram usadas ferramentas de modelação, diagramas de estados UML, casos de utilização, e para a definição da estrutura final o XML.

Tendo-se o domínio da solução passou-se para a codificação da ferramenta, usando as tecnologias de programação Java e um interpretador da metalinguagem XML chamado *Document Model* (*DOM*). A parte gráfica da aplicação foi desenvolvida com o auxílio do *Swing*.

Feito um planeamento adequado, conseguiu-se concluir a aplicação no tempo desejado com resultados confirmados.

Palavras-chave

Machine

Control

Máquina de estados

Estado

Super estado

Transição

Lista de Figuras

Lista de Siglas

API – Application Program Interfaces

DOM – Document Model

ISocRob – Instituto de sistemas robóticos Soccer Robots

JDK – Java Developer's Kit

JSK – Java Source development Kit

SPCMERC – Sistema de Projecto e Controlo de Missão de uma Equipa de Robots Cooperantes

UML – Unified Modeling Language

XML – eXtensible MarkupLanguage

Lista de Programas

Microsoft Visio

Borland Java Builder Windows

Borland Java Builder Linux

Microsoft Word

Mini Man

1 Introdução

Com o aumento da complexidade dos sistemas, aliado à diminuição dos prazos e à consequente necessidade de construir grandes equipas de desenvolvimento, torna-se cada vez mais importante a definição de um processo que sistematize o desenvolvimento de *software*. Entretanto, além da definição do processo, é necessário fazer um acompanhamento da sua execução, se possível, guiando a equipa de desenvolvimento dentro de um ambiente de desenvolvimento de *software* que contenha as ferramentas necessárias para a construção de aplicações.

O título deste trabalho, **S**istema de **P**rojecto e **C**ontrolo de **M**issão de uma **E**quipa de **R**obots **C**ooperantes (SPCMERC), transparece a ideia deste sistema ser capaz de projectar e controlar uma equipa de robots de maneira a atingirem uma missão havendo para tal cooperação entre os robots. Neste Sistema de Projecto é possível descrever objectivos/missões através de autómatos finitos para uma equipa de robots cooperantes (ou não) e definir os comportamentos individuais de cada elemento da referida equipa. O Controlo de Missão ficará a cabo de uma aplicação de simulação dos comportamentos desenhados e gerados neste projecto.

Este projecto, no âmbito do Instituto de Sistemas e Robótica, insere-se nos já existentes projectos de investigação de robótica cooperativa: *ISocRob* e *Rescue* e pretende ilustrar como se desenvolveu uma aplicação de suporte à construção de comportamentos.

O projecto *ISocRob* (*Isr Soccer Robots*) está virado para o futebol robótico, onde o objectivo não é apenas os robots marcarem golos, mas também cooperarem de modo a se assemelharem aos comportamentos humanos.

O projecto *Rescue* tem como objectivo a busca e salvamento em situações de catástrofe por uma equipa de robots, constituída de momento por um robot com rodas para o terreno e outro robot aéreo para ajuda à navegação.

Ambos os projectos utilizam presentemente máquinas de estados para modelarem os comportamentos. Sendo que, cada robot é constituído por uma série de sensores (sonares, câmaras CCD, etc.) e actuadores (motores, *kickers*, etc.), é necessário software para receber e tratar os sinais de entrada dados pelos sensores, e dar ordens aos actuadores. Para tal, foram

definidos comportamentos através de máquinas de estados, que têm que ser programadas numa linguagem específica que requer conhecimentos avançados de programação.

Existem actualmente sistemas semelhantes (e.g.: *MissionLab* [1] e *Charon* [2]), mas ou são muito complexos porque exigem grandes conhecimentos sobre todo o sistema integrante, ou requerem um grande nível de abstracção o que também resultaria numa mudança muito drástica na equipa de projecto, a nível de modelação, a nível conceptual e a nível físico.

O objectivo deste texto (relatório) é explicar como se vai desenvolver uma ferramenta onde um utilizador possa construir graficamente uma série de comportamentos para uma equipa de robots, sem ter que mexer nas diferentes camadas da arquitectura podendo-se abstrair do nível de programação, reutilizando o código já existente e em desenvolvimento pelo resto da equipa.

No capítulo seguinte é apresentado ao leitor a descrição do projecto. Começa por explicar os sistemas equivalentes em estudo e uma introdução ao trabalho desenvolvido, o desenho do projecto, e como se fez a sua implementação, mostrando as várias fases pelas quais se passou até chegar à interface gráfica. No terceiro capítulo é explicado o motor de geração de código assim como o editor gráfico. O quarto consiste na apresentação dos resultados obtidos e uma comparação entre os objectivos previstos e atingidos.

2 Conceitos, Técnicas e Metodologias

A estrutura deste projecto foi baseada em dois dos já existentes sistemas de desenvolvimento de sistemas robóticos baseados em comportamentos: *Charon* e *MissionLab*. Ambos seguem a mesma metodologia, que consiste em garantir um suporte para uma equipa de *robots*. Nestes sistemas existe uma interface gráfica de visualização e construção das configurações, assim como uma linguagem que faz a transição entre a interface e o gerador de código que produz o código que será executado no *robot*. A possibilidade de simulação das operações é uma vantagem que permite detectar erros num ambiente virtual antes da passagem para o ambiente real onde os custos e as possíveis reparações das falhas são maiores.

A ferramenta *MissionLab* é um produto do Laboratório de Robots Moveis do Departamento de Computação da Geórgia Tech que tem como principal objectivo especificar, avaliar e executar missões militares onde os ambientes são altamente dinâmicos, imprevisíveis e possivelmente hostis. Este sistema com coordenação baseada em estados (*Finite State Automata* e MDL) e baseado em comportamentos reactivos, sensores e actuadores, procura evitar a construção do controlo robótico *ad-hoc* e garantir uma configuração fácil para o problema assim como a reutilização de elementos em novas missões, suporte para multi-arquitecturas e multi-geradores de código para cada arquitectura e bibliotecas de configurações.

A ferramenta *Charon*, do Departamento de Computação e Ciências Informáticas da Universidade de Pensilvânia, é uma linguagem de alto nível e um ambiente de desenho para especificação modular de sistemas híbridos. Está baseada nas noções de agente (*agent*) e modo (*mode*). Para uma descrição hierárquica da arquitectura do sistema, o *Charon* possibilita operações de instanciação (*instantiation*), desaparecimento (*hiding*) e composição paralela nos agentes, que podem ser usados para construir um agente complexo a partir de outros. Esta ferramenta suporta comportamentos discretos e contínuos no espaço de estados.

Os dois sistemas acima referidos não foram adoptados pela equipa de projecto, e cabe aos autores deste trabalho desenvolver uma ferramenta mais específica para os projectos existentes, *ISocRob* e *Rescue*. Estas mudanças, uma vez operadas, não teriam efeito a curto prazo, porque toda a filosofia do projecto já existente teria que ser alterada.

2.1 Conceitos e Metodologias

Os autores deste texto inseriram-se no grupo de desenvolvimento do projecto *ISocRob* mais do que no *Rescue* visto este ainda estar a passar pela concepção e construção dos modelos e onde fase de implementação de código só se fará numa fase posterior. Como tal, toda a arquitectura falada, e já implementada refere-se à do projecto do futebol robótico, será mantida como tudo indica a mesma para o projecto de busca e salvamento.

A actual arquitectura funcional resulta de uma evolução da proposta original descrita em [8], guiada pelo Trabalho Final de Curso de Luís Toscano [9]

A arquitectura original, Fig. 2.1, 2.2 e 2.3, considerava três tipos de comportamentos a serem realizados pela equipa de robots: organizacionais (no que respeitam a organização da equipa, por exemplo o papel de cada jogador), relacionais (relações ente colegas de equipa, mostrando coordenação e cooperação) e individuais (cada robot como individuo). Estes comportamentos são externamente visíveis e emergem da aplicação de certos operadores. Do ponto de vista dos operadores, a arquitectura tem três níveis:

- Nível Organizacional ou de Planeamento de Tarefas – no estado corrente do projecto, este nível ainda não foi implementado, mas consiste em modelar o comportamento do oponente para planear uma estratégia.
- Nível de Coordenação ou Coordenação Tarefa/Comportamento – aqui faz-se a troca entre os comportamentos, quer sejam relacionais quer individuais de maneira a que cada robot coordene a sua execução de tarefas de forma a garantir que a sua equipa chegue ao objectivo.
- Nível de Execução – onde se faz a interface entre as tarefas primitivas, os sensores e actuadores de cada robot. Implementa-se um comportamento ligando tarefas primitivas entre elas. No estado actual do projecto *ISocRob* os comportamentos são implementados como um autómato finito, em que os estados são tarefas primitivas e as transições são condições lógicas associadas a eventos detectados pelo sistema. Os comportamentos são individuais, se correrem num só robot, ou relacionais se houver coordenação entre dois ou mais robots.

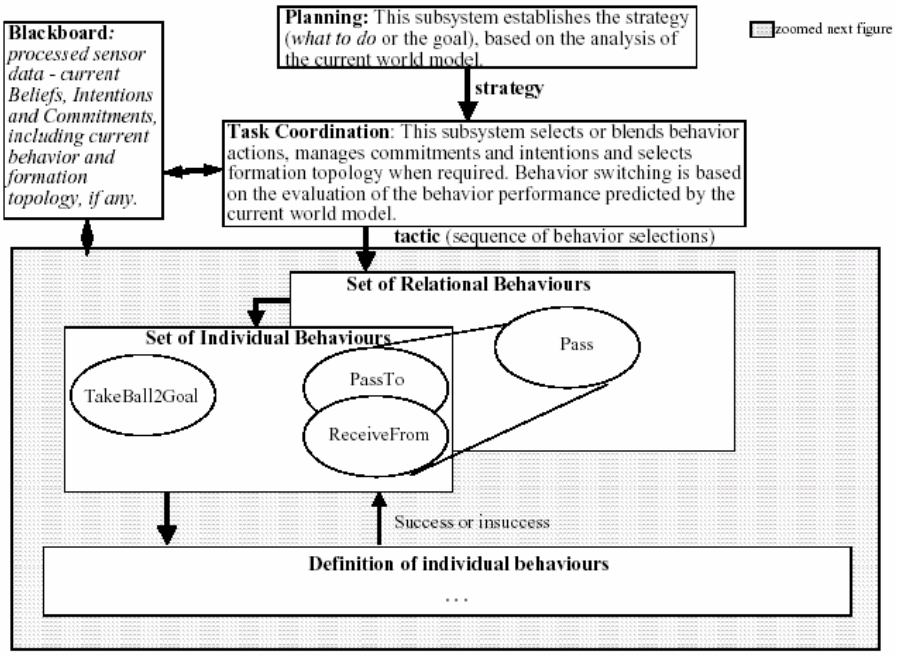


Figura 2.1 – Arquitectura funcional do ponto de vista dos operadores (extraído de [5]).

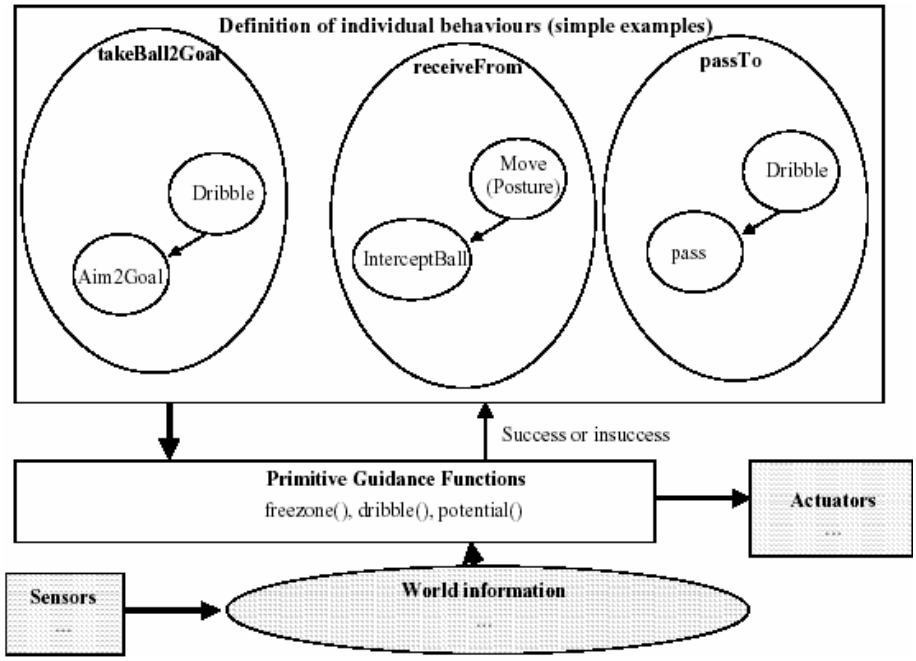


Figura 2.2 – Vista do nível de execução (extraído de [5]).

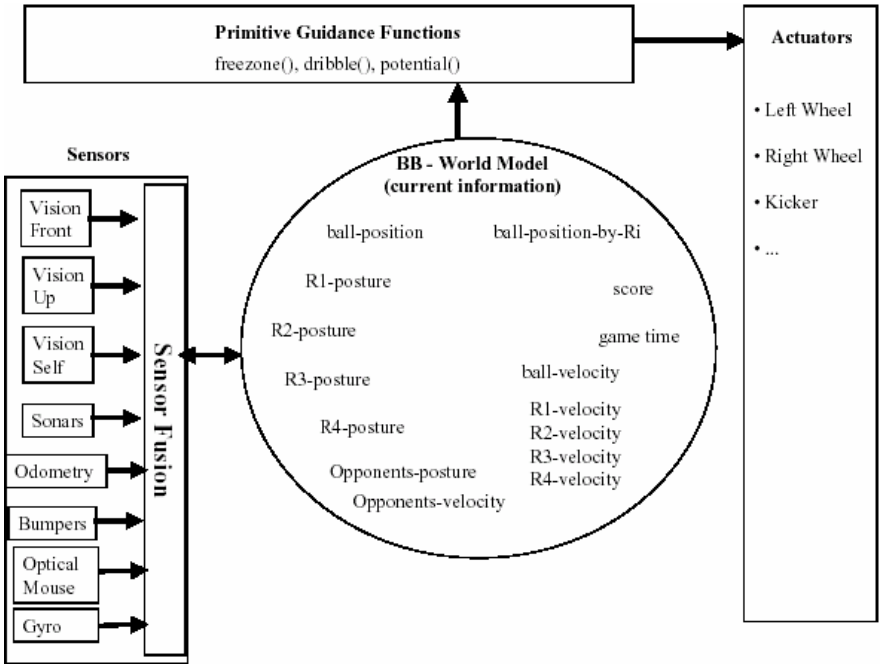


Figura 2.3 – Vista da modelação do mundo (extraído de [5]).

2.2 Processo de desenvolvimento

Para este projecto, os autores, construíram uma página de *Internet* [3], com o intuito de dar suporte ao trabalho, garantindo assim uma melhor coordenação de tarefas e possibilitando aos visitantes uma perspectiva do sistema à medida que o trabalho ia avançando. O trabalho em causa, integra-se no projecto *ISocRob* e *Rescue*, e como estes usam nos robots e nos ambientes de desenvolvimento o sistema operativo *Linux*, foi escolhida como linguagem de programação o Java devido à sua portabilidade entre sistemas operativos.

Como é sabido, o objectivo deste texto é explicar como se constrói uma ferramenta que possibilite a um utilizador desenhar uma máquina de estados num ambiente gráfico e que devolve um ou mais ficheiros em código C da mesma. Para a realização deste projecto, dividiu-se em três etapas de maneira a estruturar e modelar a solução, onde a primeira consiste no levantamento de requisitos e resulta na modelação do problema, usando uma ferramenta de modelação capaz de identificar e descrever os intervenientes de forma clara e concisa. Na segunda etapa está descrita a máquina de estados através de uma linguagem de marcação extensível (XML), usando as definições de classes resultantes da modelação da primeira fase,

sendo assim linear e transparente a passagem do modelo para a instanciação de cada um dos objectos intervenientes. Na terceira e última etapa, sendo o principal objectivo deste projecto a geração de código C a partir da descrição de uma máquina de estados, a solução foi dividida em três partes essenciais, um interpretador de XML para código C, um descodificador de um ficheiro XML que represente graficamente a informação sobre a máquina de estados, e finalmente uma interface gráfica que a partir do desenho da máquina de estados seja capaz de produzir um código C compilável e os ficheiros XML associados.

2.3 Requisitos e Modelação do problema

A arquitectura de software é baseada em micro agentes e num *BalckBoard* (quadro de informação distribuída). Estes micros agentes são processos independentes que se executam em paralelo, e implementam toda a interface com o mundo exterior, interfaces com os actuadores, sensores, colocação e leitura de variáveis do *BlackBoard*. Basicamente controlam todo o fluxo de execução da máquina (robot).

Existem vários micro agentes, Fig. 2.4: o da visão, (*Vision*) que lê as imagens das cameras, o *Kicker*, que faz a interacção com o circuito pneumático instalado, o *Proxy*, que trata da comunicação, o *Relay*, que partilha a informação contida no *BlackBoard* e os estados de cada robot com uma interface gráfica exterior para monitorização, e a destacar os dois mais importantes para este projecto, o *Machine* e o *Control*.

O *Machine* coordena os operadores/comportamentos disponíveis, no micro agente *Control*, escolhendo-os adequadamente no tempo para execução. O operador/comportamento é comunicado para o micro agente *Control*. Como foi dito anteriormente os comportamentos estão definidos através de autómatos finitos e por definição têm que possuir um estado inicial e final. Estes conceitos explicam a razão de algumas medidas tomadas para a sua implementação. Mais à frente explicaremos o conceito de máquina de estados tipo *Machine*.

O *Control* é o micro agente escolhido para execução através de uma mensagem recebida pelo micro agente *Machine*. Posto em execução o operador/comportamento solicitado, devolve uma mensagem para o *Machine* que poderá ser de sucesso ou de insucesso e é acompanhada da razão do referido (in)sucesso. Cada comportamento é aqui também definido

através de um autómato finito, este micro agente também permite, e na realidade é onde se executa, a chamada aos outros micro agentes, comunicando o pedido ou ordem, por exemplo numa chamada ao *Kicker*, ou *Vision* para ordenar um chuto ou obter imagens de uma câmara.

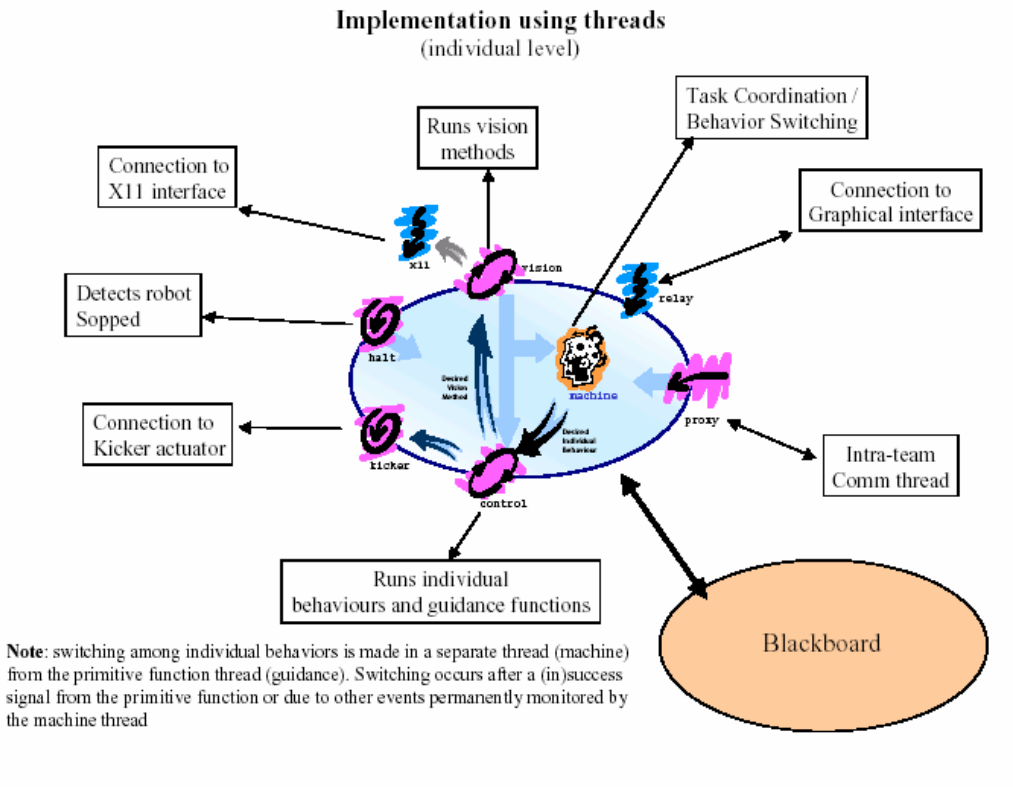


Figura 2.4 – Arquitectura de software do projecto *ISocRob* (extraído de [5]).

Foi decidido pelos autores chamar a estes autómatos finitos de máquinas de estados e chamar-lhe-emos de máquina de estados *Machine* quando é uma referência a um autómato finito para a descrição do micro agente *Machine*, e máquina de estados *Control* a um autómato finito representativo de um comportamento para o micro agente *Control*.

2.3.1 Requisitos

Um dos requisitos exigidos, é garantir que o código gerado seja Estruturado. Para tal, foi definido, pela equipa de desenvolvimento, um esqueleto que defina o código padrão dos

micro agentes *Control* e *Machine* a serem gerados, evitando-se assim erros estruturais que dificultariam a leitura à posteriori. É ilustrado na figura 2.5 o esqueleto do *Control* e na figura 2.6 o esqueleto requerido para o *Machine*.

Resumidamente o esqueleto do *Control* descreve-se da seguinte forma: para cada estado existe um modo de sensores onde é especificado quais os sensores a serem utilizados; um bloco de pré-condições onde é testada a continuidade nesse estado, ou seja, se existem condições para continuar nesse estado ou se transita para outro comportamento, pois podem ter acontecido modificações instantâneas no mundo; um corpo, onde são definidas tarefas específicas; uma função de navegação; um bloco de pós-condições, que após decorrido o comportamento pretendido podem executar uma acção e ou direccionar o fluxo de execução para um estado.

```
Init() - inicializações quando entra na máquina de estados
Destroy() - finalizações quando se sai da máquina de estados
changeTo() - o que fazer quando se muda para o comportamento
changeFrom() - o qua fazer imediatamente antes de mudar para outro comportamento
getId() - devolve o nome do estado

method()
switch(state)
      case xxxxx
            modo sensores
            vision_mode=front, up, self ou emptyspot
            pré-condições
            teste das variaveis do BB não afectadas pela função
            corpo
            activação de sensores, algoritmo do comportamento
            função
            potential(), freezone(), dribble(), etc
            pós-condições
            teste das variaveis do BB afectadas pela função
                  acção
                  típico: motorsSetVW()
      case yyyyy
            modo sensores
            pré-condições
            corpo
            função
            pós-condições
                  acção
```

Figura 2.5 – Esqueleto do código para o *Control.*

O *Machine* é descrito da seguinte forma: para cada estado existe um bloco inicial, onde se insere uma função, responsável por garantir a permanência nesse comportamento; um bloco de pré e pós-condições que se comportam como os descritos anteriormente no esqueleto do *Control*.

```
Init()
Destroy()
changeTo()

Role()
switch(state)
      case xxxxx
            bloco inicial
                  função
                  setBehaviour(),setBehaviourArg()
            pré-condições
            pós-condições
      case yyyyy
            bloco inicial
                  função
            pré-condições
            pós-condições
```

Figura 2.6 – Esqueleto do código para o *Machine.*

A Flexibilidade e a Reutilização de código foram conseguidas através de uma única caracterização para os dois modelos, no caso da aplicação, o utilizador usará a mesma interface gráfica para a criação os dois micro agentes. Nos capítulos seguintes estará explicada a modelação do problema e da solução.

Outro requisito fundamental foi possibilitar a construção de comportamentos relacionais, implementado deste modo a Cooperação entre agentes. A maneira como se poderá gerar as diferentes máquinas de estados e consequentemente uma coordenação, é através da cooperação directa chamando uma função que usa uma *hash table* para determinar o que faz num certo estado com uma determinada vista sobre o mundo. A utilização desta tabela permite presentemente coordenar a equipa de robots a ir à bola sem se atrapalharem uns aos outros. Futuramente sem alterar o código da aplicação poder-se-á implementar um passe. Na figura 2.7 está ilustrado o conteúdo dessa tabela no estado actual do projecto *ISocRob*, é

possível ver a resolução de conflitos neste caso. O predicado *ShouldIGo* devolve um valor lógico, baseado numa heurística que tem em conta a distância do robot à bola assim com a orientação do mesmo.

Para determinar se o robot em causa deve dirigir-se à bola, isto é, transitar para o estado GetClose2Ball, recorre-se ao valor lógico da coluna *Condition* na linha GetClose2Ball da tabela de *hash*. Assim a transição toma como destino o estado da coluna *State* (GetClose2Ball) se o valor (do *SouldIGo*) for verdadeiro ou pelo contrário, toma como destino o estado de falha (*Fail State*).

State	Condition
GoHome	T
GetClose2Ball	ShouldIGo
TakeBall2Goal	T
Standby	T
GoEmptySpot	T

Figura 2.7 – Tabela de *hash* (extraído de [5]).

2.3.2 Arquitectura

Esta aplicação assenta sobre uma arquitectura, Fig. 2.8, diferenciada em dois níveis conceptuais: nível superior (interface) e nível inferior (interpretador e gerador). Como base de ambos os níveis, existe o domínio (*Domain*) que não é mais que o alicerce, tanto à interface com o utilizador (*GraphicalUserInterface*) como ao motor de interpretação e geração, isto é, a sua estrutura suporta o armazenamento de todos os dados com que a ferramenta opera.

É de notar que o módulo *Parser* está incluído no motor de interpretação e que interactua directamente tanto com a interface gráfica como no acesso de leitura aos ficheiros do tipo XML.

Fazem parte integrante do motor de geração de código, os módulos de acesso a ficheiros (*FileUtils*) e geração de código C (*CGenerator*). Este último, utiliza o módulo de acesso a ficheiros para ler e escrever dados. Este motor comunica directamente com a

interface gráfica, baseando-se apenas no domínio, permitindo assim uma directa geração de código sem envolver o modulo de interpretação.

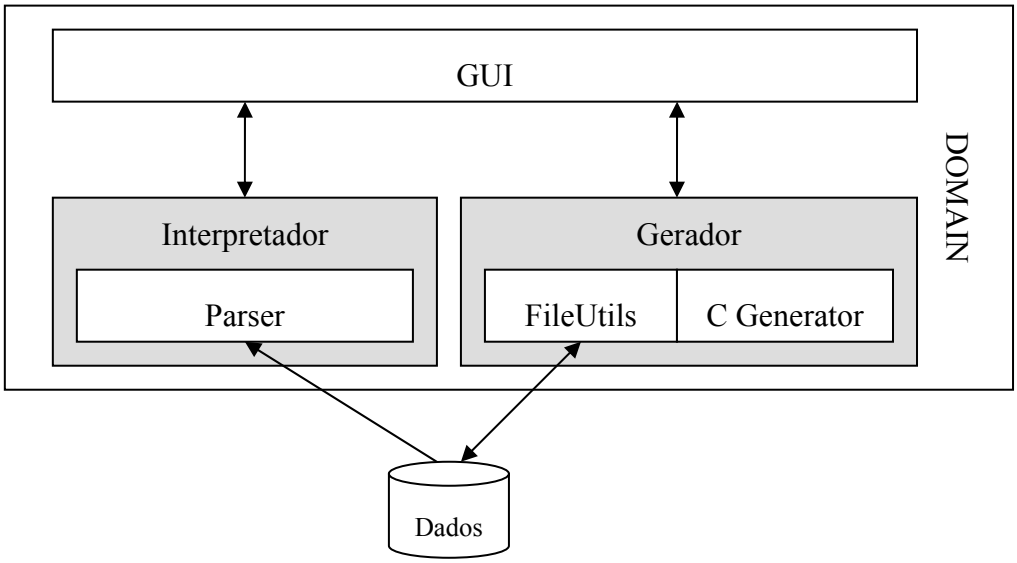


Figura 2.8 – Arquitectura SPCMERC.

2.3.3 Modelo UML

O UML (*Unified Modeling Language*) é uma linguagem diagramática, utilizável para especificação, visualização e documentação de sistemas de *software* [4]. O UML surge em 1997 na sequência de um esforço de unificação de três principais linguagens de modelação orientadas por objectos (OMT, Booch, e OOSE). Seguidamente, adquiriu o estatuto de norma no âmbito da OMG e da ISO, tendo vindo a ser adoptado progressivamente pela indústria e academia em todo o mundo.

Dado que o sistema é baseado em comportamentos descritos por máquinas de estados, a modelação por objectos encaixa-se de forma adequada, logo, para modelar a solução, usou-se uma extensão da notação gráfica do UML que descreve graficamente um diagrama de classes, de maneira a visualizar melhor os objectos e as relações em questão.

Após a análise ao sistema em estudo, foram detectados os objectos pertencentes ao domínio da solução. Estes objectos são bem distintos, quer em comportamento, em tempo de

vida quer a nível relacional, sendo os principais, a máquina de estados (*StateMachine*), o estado (*State*), o super estado (SuperState) e a transição (*Transition*), Fig. 2.9. Estes possuem três atributos de caracterização do objecto: identificador (*ID*), nome (*Name*) e descrição (*Description*). O identificador é gerado automaticamente pelo sistema, e consiste num número que identifica univocamente o objecto perante o sistema. O nome, como o nome indica, é a identificação textual do objecto e cabe ao utilizador defini-lo. A descrição não é mais que uma nota informativa acerca do respectivo objecto e seu conteúdo.

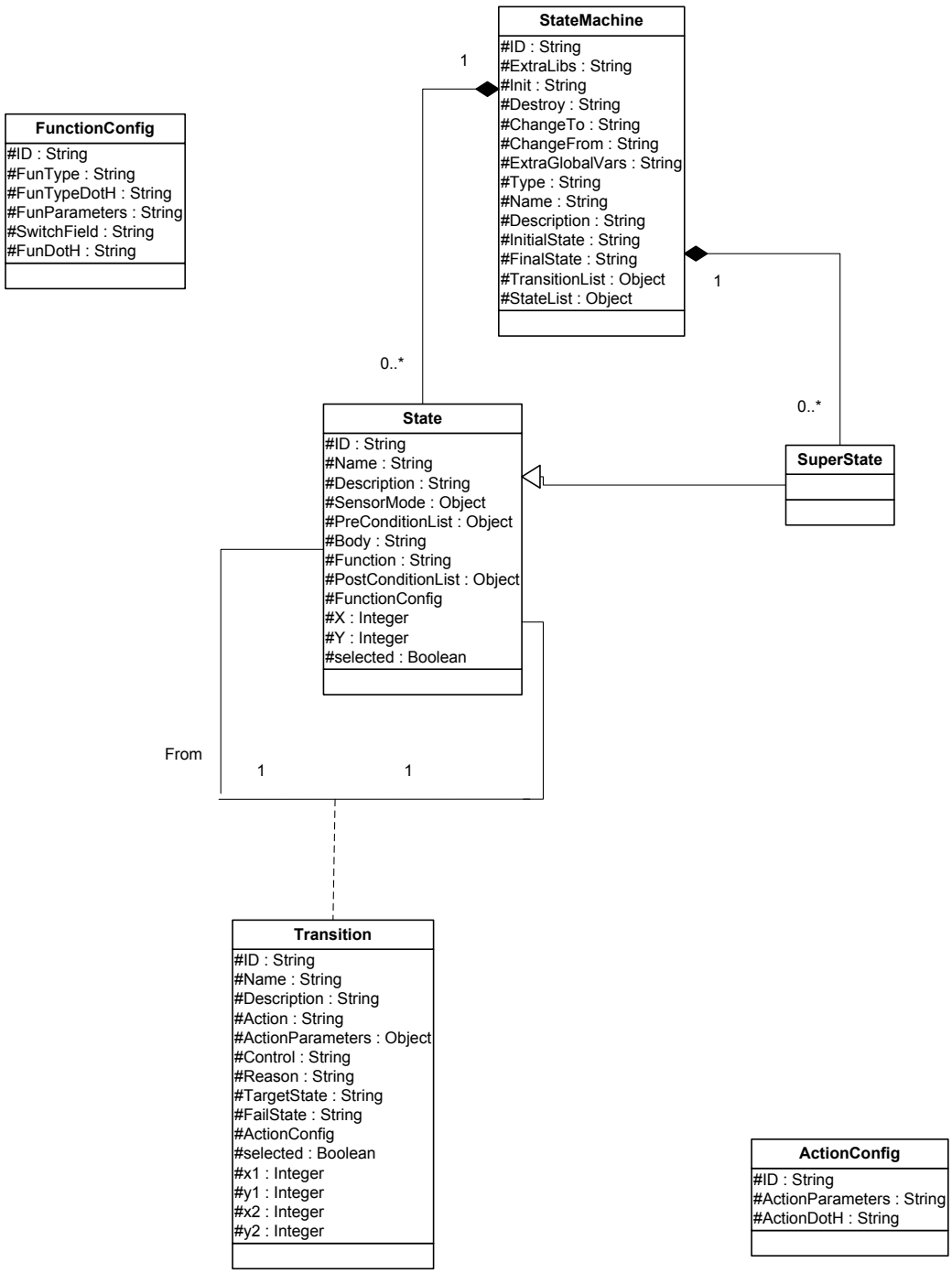


Figura 2.9 – Diagrama de Objectos do Domínio.

Uma máquina de estados tem um tipo e é composta pelos seus estados e as suas transições entre estados, tem um estado inicial e um estado final. Como esta informação é suficiente e necessária, os atributos da máquina de estados são o seu tipo (campo `Type`) que pode ser *Control* ou *Machine*, o seu nome (campo `Name`), a descrição da mesma (campo `Description`) e qual é o seu estado inicial (campo `InitialState`) e final (`FinalState`).

Compõe ainda a máquina de estados os seguintes atributos, que contêm informação opcional definida pelo utilizador: bibliotecas extra (campo `ExtraLibs`), conteúdo das funções de Init (campo `Init`), Destroy (campo `Destroy`), ChangeTo (campo `ChangeTo`), ChangeFrom (campo `ChangeFrom`), e ainda um atributo para as variáveis globais extra (campo `ExtraGlobalVars`). Nestes campos é preenchida a informação referida atribuída pelo utilizador, e compete a este assegurar um correcto formato (código C sem erros) para uma posterior geração de código (Cap. 3.1).

A esta máquina de estados, estão associados estados. Estes não são mais que uma estrutura composta por um conjunto de atributos que o caracterizam. Dado que um dos requisitos é a possibilidade de uma máquina de estados ser composta por várias máquinas de estados (no caso de o código gerado ser para o *Mahine*), então um estado pode ser um super-estado (`SuperState`) que não é mais que uma generalização de estado, ou seja, inclui a informação de um estado "normal" mas é representado internamente (nível *Machine* [5]) por uma máquina de estados. Além do nome (campo `Name`) e da descrição (campo `Description`), o estado tem um conjunto de atributos responsáveis pelo controlo do sistema em desenvolvimento, ou seja, um modo de sensores (`SensorMode`) utilizado pelos robots para os modos de visão e localização, e.g.: *front*, *up*, *self* ou *emptyspot*, um corpo (`Body`), que contém um bloco de código C relevante ao comportamento do estado e uma função (`Function`) de navegação ou de comportamento, que calcula as actuações que farão o robot andar (*Control*) ou a activação do comportamento desejado (*Machine*), respectivamente. Faz parte deste conjunto de atributos, uma ou mais pré-condições (`PreCondition`), e uma ou mais pós-condições (`PostCondition`), que são necessárias

para garantir o correcto funcionamento da máquina de estados. Estas condições são definidas através de transições.

Para suportar a informação relativa à função, caso exista, há o atributo *FunctionConfig* que é um objecto da classe *FunctionConfig*. Esta classe possui os seguintes atributos: identificador (`ID`), que identifica a estrutura perante o estado, tipo da função (`FunType`), corresponde ao tipo de retorno da função, o caminho onde está definido o tipo da função (`FunTypeDotH`) caso seja de um tipo não primitivo, os parâmetros de entrada da função (`FunParam`) resolvidos pelo sistema (Cap. 3.1.3), campo de teste (`SwitchField`), que representa a variável de retorno da função e finalmente a biblioteca onde está definida a função (`FunDotH`).

Para dar suporte à interface gráfica, visto o estado ser um objecto representado no editor, existem os atributos: x, que representa a posição no eixo horizontal do painel de desenhos, y, que representa a posição no eixo vertical do mesmo painel e ainda o atributo seleccionado (`Selected`), que indica a veracidade da selecção do estado.

As transições são funções do tipo f (estado_actual, transição_T) = estado_destino [6] e têm que ser definidas pelo nome (campo `Name`), por uma descrição do seu funcionamento (campo `Description`), uma acção (`Action`) onde se envia para os actuadores (motores) as referências calculadas pela função do estado acima referida, parâmetros de entrada para a acção (`ActionParameters`) escolhidos pelo utilizador, um estado destino (campo `TargetState`) e para o controlo do nível superior foram acrescentadas funções de retorno, são estas o controlo (campo `Control`) e a razão pela qual surgem, de maneira a poder modelar o nível comportamental (campo `Reason`). O atributo estado destino em caso de falha (`FailState`) guarda o nome do estado para o qual a transição transita caso não seja possível atingir o estado destino (`TargetState`).

Para armazenar a informação relativa à acção, caso exista, existe o atributo *ActionConfig* que é um objecto da classe *ActionConfig*. Esta classe auxiliar contempla os seguintes atributos: identificador (`ID`), que identifica a estrutura perante a transição, parâmetros de entrada e seus tipos (`ActionParam`) retirados do cabeçalho da acção ora definidos no também atributo biblioteca da acção (`ActionDotH`).

Sendo a transição objecto de desenho por parte do editor gráfico, existem ainda os atributos: x1, x2, y1 e y2 posições no painel de desenho para se desenhar uma curva (Cap. 3.2.2) e seleccionada (`selected`) que indica a veracidade da selecção da transição.

2.4 As Linguagens

2.4.1 Definição da Metalinguagem – XML

Antes de se saber o que é a linguagem marcada extensível (eXtensible Markup Language) deve-se focar o conceito de *Markup Language*. Uma *Markup Language* (ML), como descrito em [7], é uma linguagem que descreve como se estrutura um texto dentro de um dado documento. Serve para definir a estrutura, conteúdo e secções do documento. As secções do documento são definidas usando rótulos (*tags*).

A XML foi desenvolvida em 1996 por um grupo de trabalho sobre a orientação do *World Wide Web* (WWW). Esta linguagem permite definir colecções de *tags* que podem ser usadas para estruturar qualquer tipo de dados ou documentos.

A Metalinguagem está definida em XML tanto para o *Control* como para o *Machine*, Fig. 2.10 e Fig. 2.11, respectivamente, devido à facilidade de interacção com esta linguagem e às *Application Program Interfaces* (API) Java disponíveis (Cap. 2.4) para manipulá-la. Dada a actual modelação em UML (Cap. 2.3.3), Fig. 2.9, a transposição para uma metalinguagem em XML foi imediata e linear. É de observar que as classes definidas no modelo de classes e os seus atributos estão representados no XML por elementos Pai (aqueles que estão a um nível a cima) e elementos Filho (aqueles que estão dentro do nível do elemento pai) respectivamente.

Importa diferenciar as duas definições da metalinguagem na medida em que, para cada caso foram reservados apenas os campos da estrutura (ver UML) necessários, e compete a estes o armazenamento da informação para uma posterior geração de código C.

```
<StateMachine>
	<ID> </ID>
	<Init> </Init>
	<Destroy> </Destroy>
	<ChangeTo> </ChangeTo>
	<ChangeFrom> </ChangeFrom>
	<Type> </Type>
	<Name> </Name>
	<Description> </Description>
	<InitialState> </InitialState>
	<FinalState> </FinalState>
		<State>
			<ID> </ID>
			<Name> </Name>
			<Description> </Description>
			<SensorMode> </SensorMode>
			<PreCondition> </PreCondition>
			<Body> </Body>
			<Function> </Function>
			<PostCondition> </PostCondition>
			<FunctionConfig>
				<ID> </ID>
				<FunctionType> </FunctionType>
				<FunctionTypeDotH> </FunctionTypeDotH>
				<FunctionParameter> </FunctionParameter>
				<SwitchField> </SwitchField>
				<FunctionDotH> </FunctionDotH>
			</FunctionConfig>
			<X> </X>
			<Y> </Y>
		</State>
		<Transition>
			<ID> </ID>
			<Name> </Name>
			<Description> </Description>
			<Action> </Action>
			<ActionParameter> </ActionParameter>
			<Control> </Control>
			<Reason> </Reason>
			<TargetState> </TargetState>
			<ActionConfig>
				<ID> </ID>
				<ActionParameter> </ActionParameter>
				<ActionDotH></ActionDotH>
			</ActionConfig>
		</Transition>
</StateMachine>
```

Figura 2.10 – Definição da metalinguagem em XML para *Control.*

```
<StateMachine>
      <ID> </ID>
      <Init> </Init>
      <Destroy> </Destroy>
      <ChangeTo> </ChangeTo>
      <Type> </Type>
      <Name> </Name>
      <Description> </Description>
      <InitialState> </InitialState>
      <FinalState> </FinalState>
            <SuperState>
                  <ID> </ID>
                  <Name> </Name>
                  <Description> </Description>
                  <PreCondition> </PreCondition>
                  <Function> </Function>
                  <PostCondition> </PostCondition>
                  <X> </X>
                  <Y> </Y>
            </SuperState>
            <Transition>
                  <ID> </ID>
                  <Name> </Name>
                  <Control> </Control>
                  <Reason> </Reason>
                  <TargetState> </TargetState>
                  <FailState> </FailState>
            </Transition>
</StateMachine>
```

Figura 2.11 – Definição da metalinguagem em XML para *Machine.*

Em ambas as definições, a metalinguagem está estruturada como se de uma árvore se tratasse, ou seja, no topo está o nó raiz (<*StateMachine*>), composto por nós folhas (<*State*> ou <*SuperState*> e <*Transition*>). Todos os nós são caracterizados por atributos (*tags*). Estes atributos já foram explicados anteriormente (Cap. 2.3.3). As *tags* <*PreCondition*> e <*PostCondition*> são preenchidas com o identificador (<*ID*>) da transição (<*Transation*>) em questão. Com o mesmo raciocínio são preenchidos o estado inicial (<*InitialState*>), estado final (<*FinalState*>) e o estado destino (<*TargetState*>) mas desta vez com o identificador do estado em questão.

As *tags* modo sensores (<*SensorMode*>), função (<*Function*>) e acção (<*Action*>) são preenchidas com um nome já existente na biblioteca de funções que o sistema possui (Cap. 3.1.3), isto é, se já existir na biblioteca uma função de nome `Chuta` para chutar a bola,

podemos então preencher a *tag Function* com `Chuta`. A *tag* de controlo (<*Control*>) e razão (<*Reason*>) são preenchidas respectivamente pelo sucesso ou insucesso de um comportamento (e.g.: *BEHAVIOUR_SUCESS*) e pelo motivo desse sucesso ou insucesso.

Uma característica desta metalinguagem é a flexibilidade, ou seja, o sistema tem a possibilidade de não preencher as *tags* que acha desnecessárias ou momentaneamente inconvenientes.

Importa ainda referir os casos onde existe restrição à criação de *tags* dentro de certos nós. No *Control*, a impossibilidade de escolher uma acção para uma determinada pré-condição leva a que a estrutura do nó *Transition* não contenha qualquer informação referente a uma acção.

No *Machine*, na criação de uma pré-condição (entre estados) é impossível definir o *control* e *reason*, restringindo essa definição no nó *Transition*. Em qualquer transição a existência da *tag FailState* só tem significado quando existir um *Name* para a transição, ou seja, tendo a transição um nome (predicado individual) há necessidade de criar um estado de destino (*FailState*) em caso de falhar o salto para o estado pretendido (*TargetState*), caso contrário não faz sentido existir esse estado destino (*FailState*).

2.4.2 Linguagem de Programação JAVA

O Java é uma linguagem orientada pelos objectos, baseada em classes e tipificada. Contudo, do ponto de vista da orientação pelos objectos, não é uma linguagem completamente pura, só por uma razão: em Java, os dados de tipos primitivos - int, boolean, char, etc. - são valores e não objectos.

O Java também suporta concorrência, mediante a utilização de métodos synchronized e de objectos da classe predefinida Thread. O Java resulta, em grande parte, duma simplificação do C++ (concretamente da primeira versão desta linguagem). O Java deixa de fora os seguintes elementos: apontadores, "templates", gestão explícita de memória (ou seja, a primitiva *delete*), pré-processador (#defines, etc.), *typedef*, *union*, *struct*, funções globais, variáveis globais, funções membro não virtuais, *goto*, ficheiros ".h", sobreposição de

operadores, herança múltipla, conversões automáticas definidas pelo programador, enumerados.

Não obstante derivar do C++, o Java adopta a nomenclatura da linguagem Smalltalk (actualmente a nomenclatura padrão de facto do paradigma orientado pelos objectos). O Java é uma linguagem interpretada, portanto portável a nível binário. Simultaneamente começa também a ser uma linguagem razoavelmente eficiente: as implementações antigas eram ineficientes, mas a partir da versão 1.2 a eficiência do Java deu um salto em frente.

A linguagem Java, sendo relativamente pequena, herda muito do seu sabor e poder duma grande biblioteca de classes predefinidas, chamada plataforma Java (também conhecido por core Java ÁPIS e Java *runtime environment*). Esta biblioteca é robusta, intuitiva e bem desenhada. Tem vindo a crescer e a amadurecer com cada nova versão do sistema.

Actualmente, a plataforma Java é tão vasta que os programadores já não precisam de aceder directamente aos serviços do sistema operativo. Assim, a plataforma Java apresenta-se como uma plataforma de desenvolvimento universal. Sobre ela correm todos os programas Java, independentemente do sistema operativo subjacente.

Ao longo da evolução do Java, a linguagem tem mudado pouco. Na versão 1.1 e na versão 1.4, foram introduzidas algumas extensões, mas em número reduzido.

Felizmente, a linguagem Java é conservadora neste aspecto. Pelo contrário, a plataforma Java tem vindo a crescer exponencialmente.

O ambiente de desenvolvimento JDK (Java *Developer's* Kit) consiste num conjunto de ferramentas que ajudam a desenvolver, testar, documentar e executar programas em Java. O JDK é gratuitamente disponibilizado pela empresa *Sun* e existem versões para os mais variados sistema: Linux, Windows, MacOS, etc.

2.5 Definição da interface

Após descrita a linguagem XML (Cap. 2.4.1) que é de primordial importância para a realização deste trabalho, resta descrever a interface gráfica. Adianta-se que a outra linguagem utilizada na futura implementação da mesma é a linguagem de programação Java, escolhida

por ter simples interacção com a linguagem XML, e porque permite a portabilidade do sistema operativo, ou seja, funciona tanto em *Linux* como em *Windows*.

Esta interacção parece ter aparecido principalmente devido ao facto da XML permitir "dados portáveis" que podem ser combinados com o "código portável" que a linguagem de programação Java proporciona.

A popularidade da XML provocou o aparecimento de várias ferramentas de análise (*parsing*) e manipulação de documentos XML, por parte das linguagens de programação (e.g., Java). Existem duas aproximações comuns usadas para a análise [7]: SAX e Document Model (DOM). A aproximação que se irá usar será o DOM o qual explicaremos mais adiante neste texto.

Foi definido pelos autores dividir a parte de implementação em três fases:

- na primeira a ideia é pegar num ficheiro XML e ser capaz de gerar código C a partir dele. Este programa tradutor consistirá num *parsing* dos ficheiros onde se encontra a definição da máquina de estados, e devolverá uns ficheiros com código C compiláveis.
- Na segunda fase, tendo-se o tradutor implementado, prossegue-se para a criação de um descodificador que permita através do ficheiro em XML, desenhar toda a máquina de estados numa interface gráfica.
- A terceira e última fase corresponderá ao duplo sentido, ou seja, o utilizador já poderá desenhar a máquina de estados e será gerado o código C correspondente, assim como, criar a estrutura num ficheiro XML que poderá ser visualizada graficamente e ser gerado o código C.

2.5.1 Primeira fase – Interpretador XML

Esta fase corresponde ao núcleo deste projecto. A primeira coisa a fazer com o ficheiro XML é a sua análise. Como já foi dito, a aproximação que irá ser usada será o DOM que se baseia num protocolo que converte um documento XML numa colecção de objectos. Este modelo de objectos pode ser manipulado na maneira pretendida. Este mecanismo é também

conhecido por protocolo de acesso aleatório (*random access*) porque se pode aceder a qualquer parte dos dados e em qualquer momento. O pacote (*package*) DOM utilizado foi o *javax.xml.parsers* que define as classes *DocumentBuilderFactory* e *DocumentBuilder*, que retornam um objecto que implementa a interface *W3C Document*, Fig.2.12.

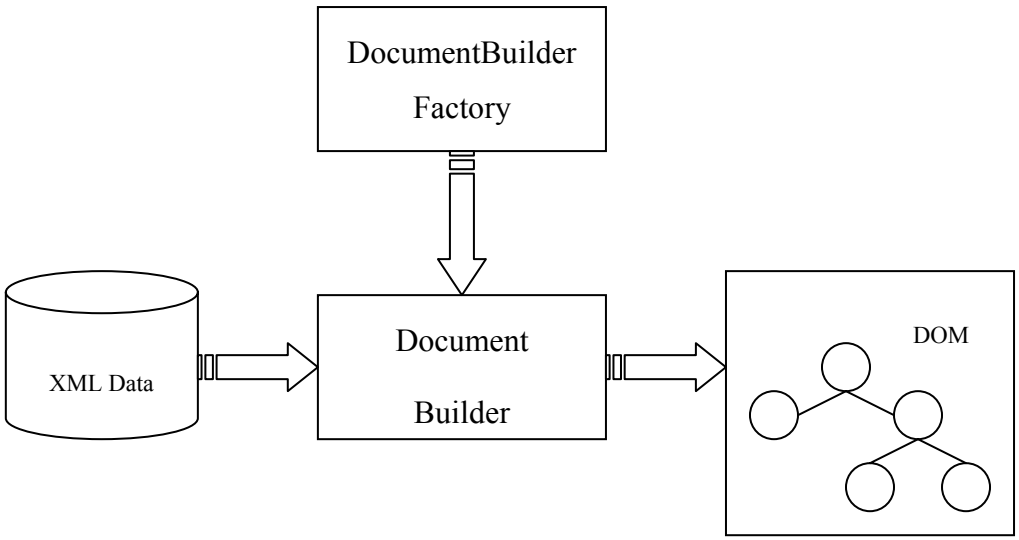


Figura 2.12 – Funcionamento da JAXP API.

Feita a análise ao documento, este será guardado pela aplicação, em estruturas dinâmicas definidas pelo diagrama de objectos descrito acima (Cap. 2.3.3). Trata-se de criar em seguida ficheiros de código, que seguem as regras padronizadas pelos autores.

2.5.2 Segunda fase – Descodificador de XML

Esta fase descreve como se desenvolveu o descodificador gráfico de um documento XML. Ao utilizador será dada hipótese de ver graficamente a sua máquina de estados antes de querer gerar o código. Para tal desenvolveu-se uma API, que através do DOM analisa o XML, tal como na primeira fase, decompondo-o em estruturas dinâmicas para representação gráfica, Fig. 2.13.

Esta fase intermédia, visa representar graficamente a estrutura XML criada anteriormente utilizando para isso a informação contida nessa estrutura (Domínio), e.g. máquina de estados, estados, transições, etc. Desta forma o utilizador consegue ter uma visão mais conceptual do que havia criado.

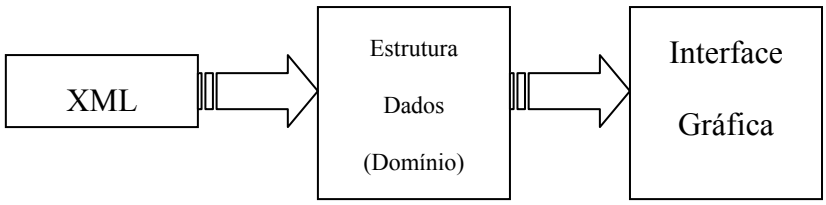


Figura 2.13 – Esquema de desenvolvimento.

2.5.3 Terceira fase – Interface gráfica

Enquanto a geração do esqueleto se aproximava de um fim, a interface gráfica ganhava forma. Após se conseguir visualizar as diferentes máquinas de estados conseguidas na segunda fase, passou-se para a concepção de uma interface de construção.

Nesta fase, Fig.2.14, programou-se toda a interface usando a linguagem Java e a sua API de desenvolvimento gráfico *Swing*. Para tal pegou-se nos objectos do domínio, definidos no Cap. 2.3.3, e criou-se um esquema de casos de uso para que se definisse todas as funcionalidades a serem implementadas. Mais à frente será mostrado esse diagrama de casos de uso.

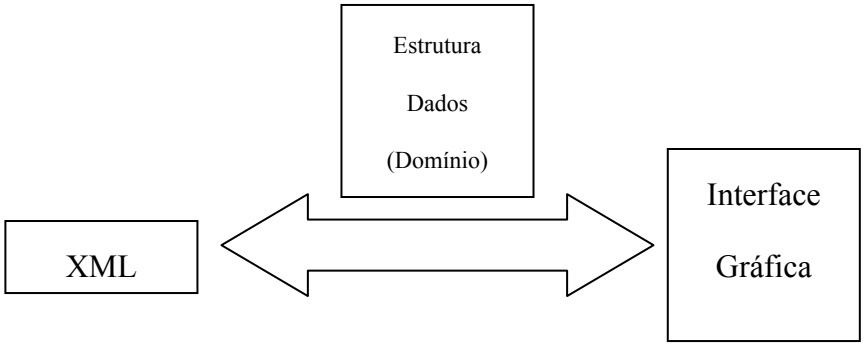


Figura 2.14 – Terceira fase.

Findo esta tarefa, e a aplicação já permitia construir, rever, salvar, abrir e modificar máquinas de estados assim como gerar código compilável para ambas as máquinas de estados. Após este passo procedeu-se a testes intensivos, que completa qualquer construção de software.

2.6 Metodologia de trabalho

Os autores deste trabalho, procuraram dividir esta ferramenta em dois níveis fundamentais: nível superior (interface gráfica) e nível inferior (geração de código e interpretação do XML). Esta divisão permitiu o desenvolvimento paralelo do sistema, ou seja, à medida que o nível inferior foi evoluindo, a interface ia ganhando forma.

O planeamento foi uma parte muito importante neste projecto, na medida em que ajudou à organização e modelação das várias tarefas ao longo deste. Feita a análise do problema, foi possível decompor o sistema em sub problemas tratáveis, para tal foi necessário definir um conjunto de etapas. O modelo de processo utilizado foi em cascata revisto, que consiste em subdivisões bem explicitas das etapas e em que a sequência lógica de execução é retroactiva. O modelo em cascata revisto, Fig. 2.15, prevê a possibilidade de a partir de qualquer tarefa do ciclo se poder regressar a uma tarefa anterior de forma a contemplar alterações funcionais e/ou técnicas que entretanto tenham surgido, em virtude de um maior conhecimento que se tenha obtido [4]. As principais etapas são: consulta, pesquisa, modelação, construção e melhoramentos. A etapa consulta consiste num estudo objectivo sobre sistemas semelhantes já

existentes, dando estes uma visão pormenorizada do problema a resolver e de uma possível solução. A principal tarefa envolvida nesta etapa foi o estudo dos sistemas *Charon* e *Mission Lab*. Na etapa pesquisa foram recolhidas informações específicas para solucionar os sub problemas e dar suporte ao desenvolvimento. Incidem sobre esta etapa as tarefas de pesquisa de bibliografia sobre os elementos relevantes ao projecto e recolha de informação sobre *Unified Modeling Language* (UML), eXtensible Markup Language (XML) e analisadores (*parsers*) e sobre programação de interfaces gráficas em Java.

Na etapa modelação, uma vez modelada a solução usando diagramas de classes (UML) e a consequente modelação da metalinguagem intermédia entre a especificação e a geração de código, modelou-se os aspectos relevantes ao interpretador da interface gráfica.

A etapa construção consiste na implementação dos três módulos acima descritos (Cap. 2.5). Finalmente a etapa melhoramentos serviu para resolver erros de programação que se encontraram e para testes.

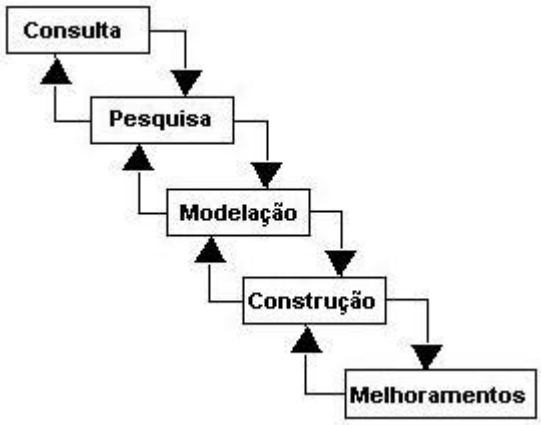


Figura 2.15 – Modelo em cascata revisto.

Em conformidade com a vontade do Professor em manter o grupo de projecto actualizado e atento ao desenvolvimento desta ferramenta, encontra-se em anexo (Apêndice A), alguns dos *slides* mais relevantes apresentados à equipa, com o objectivo de integrar os restantes membros no conteúdo desta ferramenta.

O relacionamento com o Professor decorreu de forma muito profissional e pedagógica. Foram mantidas reuniões semanais com o grupo de desenvolvimento do ISocRob, e reuniões

extraordinárias sempre que a vontade dos autores e orientador coincidiam. Pode-se assim dizer, que o projecto foi orientado com rumo e precisão até ao produto final.

3 Descrição do Trabalho

Como foi explicado anteriormente, este projecto enquadra-se num sistema de geração de código C a partir da criação de uma máquina de estados num ambiente gráfico. Tendo em conta alguns dos requisitos, foi desenvolvido um motor capaz de gerar código C compilável, resultante de um planeamento direccionado a uma missão de robots. Este motor utiliza automatismos como apoio à geração, capazes de proporcionar ao utilizador uma abstracção a nível da produção de código C.

Está garantida ainda, uma interface gráfica composta por janelas de navegação, que conduzem a uma criação/edição de máquinas de estado como se de um desenho se tratasse.

De seguida serão apresentados e desenvolvidos os conceitos atrás descritos.

3.1 Geração de código C

O código C gerado obedece não só a uma estrutura requerida pela linguagem, mas também a um esqueleto padrão (Cap. 2.3.1) que proporciona tanto uma leitura mais explícita do código, como a facilidade de futuras alterações no referido código C.

Respeitando o esqueleto padrão consegue-se obter um código mais consistente, reutilizável e de fácil correcção de erros permitindo um melhor enquadramento por parte do programador.

Dadas as necessidades do projecto ISocRob, foi necessário desenvolver dois geradores distintos no que respeita ao código gerado mas idênticos na concepção. Partindo da actual estrutura do domínio (*Domain*), gerou-se código C tanto para o *Control* como para o *Machine*, sendo essa a principal função das classes *CGenerator* e *SuperCGenerator* respectivamente.

3.1.1 Geração de código para *Contol*

Este motor de geração de código C teve de respeitar a estrutura exigida pela linguagem C na criação de um novo ficheiro deste formato. A estrutura do ficheiro C gerado, está dividida em quatro partes distintas: cabeçalho e inclusão de bibliotecas, funções a exportar e definição de variáveis, funções de inicialização e função principal (*method*).

No cabeçalho e inclusão de bibliotecas, está definido um comentário padrão que segue as normas de boa programação do ISocRob [10] onde é especificado o nome do ficheiro, descrição do conteúdo, versão e autor. Imediatamente a seguir vem a inclusão das bibliotecas de que o ficheiro necessita para que possa ser compilado.

Nas funções a exportar vem definido o cabeçalho das funções de *init*, *destroy*, *changeTo*, *changeFrom* e *getId* [5]. Posteriormente surge a definição das variáveis utilizadas pelo programa.

Para as funções de inicialização define-se o conteúdo das funções a exportar descritas atrás. Este conteúdo é da inteira responsabilidade do utilizador que pode ou não acrescentar novas linhas de código C caso ache relevante para o comportamento do sistema.

Na função principal está definido o núcleo do programa, onde está especificado o que fazer e quando fazer. Partindo de um esqueleto padrão utilizado pela equipa ISocRob para a criação de código C foi estruturado o motor de geração de código para cada estado, que respeita os seguintes blocos do esqueleto: modo de sensores, pré-condições, corpo, função e pós-condições.

3.1.2 Geração de código para *Machine*

Seguindo os princípios anteriores, a estrutura do ficheiro C está dividida em quatro partes: cabeçalho e inclusão bibliotecas, definição de variáveis, funções de inicialização e função principal (*Role*).

O cabeçalho e inclusão de bibliotecas seguem o mesmo conceito descrito anteriormente.

Na definição de variáveis, como o nome indica, estão definidas todas as variáveis que o programa necessita.

Nas funções de inicialização define-se o conteúdo das funções *Init*, *Destroy* e *ChangeTo*. Cabe ao utilizador preencher, ou não, o conteúdo.

Na função principal está definido o núcleo do programa, onde está especificado o que fazer e quando fazer. Não existindo um esqueleto padrão formalizado, foi adoptado o seguinte modelo: bloco inicial, pré-condições e pós-condições.

Para o *Machine* será ainda gerado um ficheiro H, cujo conteúdo não é mais que os cabeçalhos das funções de inicialização e da função principal descritas anteriormente.

3.1.3 Automatismos

Uma das principais preocupações deste trabalho foi a automatização de acções, ou seja, visou-se proporcionar ao utilizador um ambiente de fácil utilização e que requer o mínimo de informação relevante. Com isto, procura gerar código C mais estruturado, de fácil reutilização e mais consistente.

Como suporte à geração de código, tanto para o *Control* como para o *Machine*, foram criados alguns automatismos que permitem a criação automática de blocos de código em certas zonas do ficheiro, poupando esse esforço ao utilizador.

A base de todos os automatismos é o ficheiro de dados *PathConfig*, este é responsável por armazenar os caminhos que resolvem uma chave (*Key*) e possui a seguinte estrutura: identificador da chave, chave, caminho e para algumas chaves um campo de teste referente ao código C (*switch field*), Fig. 3.1. Existem os seguintes identificadores de chave: **F** para funções, **A** para acções, **M** para modo de sensores, **S** para estruturas, **B** para comportamentos (*Machine*) e **O** para outros.

A chave não é mais que um nome, que pode assumir o nome da própria função ou um nome atribuído pelo sistema. O caminho, como o nome indica, representa o endereço do ficheiro onde se encontra a chave definida. Todos os campos do ficheiro estão separados pelo carácter ‘ # ‘, Fig. 3.2. Este ficheiro de configuração entre muitas coisas permite que uma vez definida uma nova chave fique acessível a todos os utilizadores a partir desse momento.

Identificador de Chave # Chave # Caminho # Campo de Teste

Figura 3.1 – Estrutura do ficheiro *PathConfig*.

S#Actuation#../../guidance.h#.return_value

Figura 3.2 - Exemplo de um registo do ficheiro *PathConfig*.

Tirando os casos onde o utilizador pode introduzir linhas de código C, onde lhe compete a responsabilidade de definir as bibliotecas para as funções que utilizar, existe um automatismo capaz de gerar a inclusão de todas as bibliotecas necessárias para que o sistema possa compilar, isto é, todas as linhas de código são analisadas sem erros pelo compilador de C. Este automatismo recolhe da estrutura da máquina de estados as possíveis bibliotecas a serem incluídas (modo sensores, função, tipo da função caso seja não primitivo e acção), assim como as bibliotecas necessárias definidas no *PathConfig* (e.g. predicados, constantes, operadores lógicos, etc.). Se não existirem no *PathConfig* caminhos para estas ultimas bibliotecas necessárias cabe ao utilizador, no momento da geração de código, definir os endereços.

Tendo o utilizador a capacidade de poder definir endereços (caminho das bibliotecas) para um nome (e.g. função, acção, modo sensor, estrutura, etc.), foi criado um automatismo com a responsabilidade de garantir a existência do endereço e a definição do nome nesse mesmo endereço. Assim garante-se a consistência e coerência da informação.

Para a definição das variáveis globais no código, foi criado um automatismo que permite definir todas as variáveis necessárias para a execução final do ficheiro. Começa-se por definir uma estrutura de nome *private* que visa auxiliar a programação. De seguida, para cada estado e na existência de função, procura os parâmetros de entrada que se encontram dentro do *FunctionConfig* e define-os (será explicado a seguir como foram obtidos).

Como base da listagem dos predicados e das constantes está um automatismo que permite ao utilizador escolher, de uma lista de predicados e de uma lista de constantes, qual o predicado ou constante a utilizar ao longo da criação de uma máquina de estados. Este mecanismo consiste em procurar no caminho dos predicados (definido no PathConfig) todas as funções do tipo inteiro, lógico (boleano) ou real (noção de predicado). Partindo do mesmo conceito, procura no caminho das constantes todas as palavras precedidas de "`#define`"

(palavra reservada na linguagem C para a definição de constantes) e guarda-as na lista de constantes.

Um automatismo utilizado especificamente para a geração de código C a partir de uma máquina de estados do tipo *Control*, é o processo de escolha dos parâmetros de entrada para a função (*function*) de cada estado, e toda a informação inerente a esta, e ainda a escolha do campo de teste (*switch field*). Este mecanismo processa-se da seguinte forma: existindo o nome dessa função no *PathConfig* é guardado o caminho do ficheiro para o qual a função está definida e é usado para procurar tanto o tipo da função como para a obtenção dos seus parâmetros de entrada. A citada função pode assumir um de dois tipos: tipo inteiro e tipo não primitivo (Cap. 2.4.2). Se for do tipo inteiro o programa toma o seu comportamento normal, isto é, testa o resultado de retorno da função e prossegue a execução. Tratando-se de uma função de um tipo não primitivo, supõe-se o tipo estrutura (*struct*) visto consubstanciar um mecanismo que lhe permite preencher os campos da referida estrutura, cujo caminho onde esse tipo está definido será pesquisado no *PathConfig*. Se neste mesmo ficheiro de configuração existir um terceiro campo preenchido o programa utiliza-o como campo de teste (*switch field*) e prossegue a execução, caso contrário, o programa verifica na referida estrutura a existência de campos do tipo inteiro. Existindo apenas um campo desse tipo o programa automaticamente grava-o no *PathConfig* como terceiro campo e prossegue utilizando-o como campo de teste. Existindo na estrutura mais do que um campo do tipo inteiro, cabe ao utilizador escolher qual o campo de teste (Apêndice C). Com base na informação obtida o programa irá gerar código C. Para guardar a informação implícita à função, designadamente parâmetros e a biblioteca onde esta está definida, o programa utiliza uma estrutura *FunctionConfig* de suporte ao armazenamento, incluída na estrutura do estado (Cap. 2.3.3). Desta forma, cada estado possui a informação relativa à sua função, caso exista, assim como as suas próprias características.

O seguinte automatismo assim como o anterior existe apenas na geração de código para o *Control*. Este automatismo visa guardar a biblioteca onde está definida a acção e os seus parâmetros de entrada, escolhidos anteriormente pelo utilizador. Para esta escolha apenas estão disponíveis parâmetros do mesmo tipo dos parâmetros que a acção recebe, ou seja, é dado ao utilizador uma lista de parâmetros composta pelas variáveis globais definidas

anteriormente para a função do estado de origem da transição. Nesta lista apenas se encontram parâmetros cujo o tipo é suportado pelos parâmetros de entrada da acção, Fig. 3.3.

Esta informação é armazenada na estrutura *ActionConfig* (Cap. 2.3.3), e permitirá ao gerador, para cada transição, obter toda a informação necessária acerca da acção.

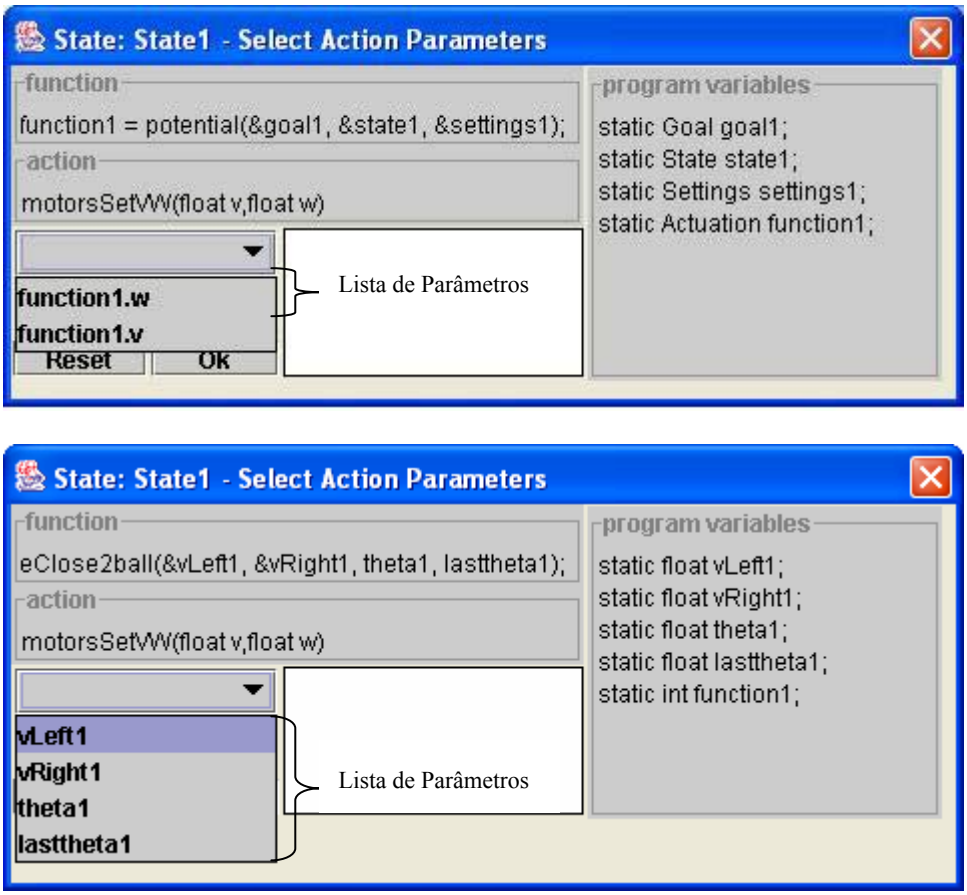


Figura 3.3 – Exemplos da escolha de parâmetros para a acção.

Para a geração de código no *Machine* foi criado um automatismo sempre que é utilizado um predicado individual. Esta ideia assenta na existência de predicados relacionais por parte do projecto *ISocRob* [5]. Este mecanismo sempre que é utilizado um predicado individual desencadeia a instrução *changeBehaviourTo* que está implicitamente relacionado com uma tabela *hash* integrada no referido projecto (Cap. 2.3.1).

Uma questão que se levantou na geração de código tanto para *Control* como para *Machine*, era garantir que os conteúdos das funções de inicialização assim como o corpo (*Body*) no caso do *Control*, mantivessem uma identação coerente em relação ao resto do código. Para este caso, foi criado um automatismo que desresponsabiliza o utilizador de tais funções e parte ele próprio para a identação de todo o código C.

3.2 Editor Gráfico

De maneira a que o utilizador possa manipular e criar máquinas de estados de forma intuitiva e rápida, foi criada uma interface gráfica de edição de máquinas de estados. Através desta o utilizador tem acesso a uma série de funcionalidades, tais como criar, modificar, salvaguardar e abrir máquinas de estados já gravadas. Nos sub capítulos seguintes poderá observar-se as funcionalidades através do caso de utilização, e uma explicação de como foram projectados os objectos para serem desenhados, as suas estruturas e como foram representados graficamente. Para a implementação deste editor gráfico, foi usado o *Swing* (*JavaTM 2 Platform, standard Edition 1.4.1*) que consiste numa compilação de componentes do Java para criação de ambientes gráficos (janelas, botões, etc.) que funciona em praticamente todos os sistemas operativos da mesma maneira. O *Swing* foi a ferramenta escolhida devido à sua portabilidade, facilidade de implementação e integração de componentes mais complexos.

O maior desafio na implementação deste editor foi arranjar uma representação gráfica para cada objecto. Ou seja, tendo um estado como é que se representa? Tendo uma transição de um estado para outro como é que será desenhado? E uma transição para o mesmo estado? Como é que se poderá seleccionar um estado ou uma transição? Quando existem várias transições para o mesmo estado como é que se faz para não serem coincidentes, e não se sobreporem?

Para responder a este tipo de questões, e para resolver o problema do desenho dos objectos, foi primeiro desenhado um diagrama de casos de utilização, onde se podem encontrar todas as funcionalidades do editor, Fig. 3.4. De seguida teremos um sub capítulo onde se responderá a estas questões mostrando os respectivos algoritmos (Cap. 3.2.2).

3.2.1 Casos de Utilização

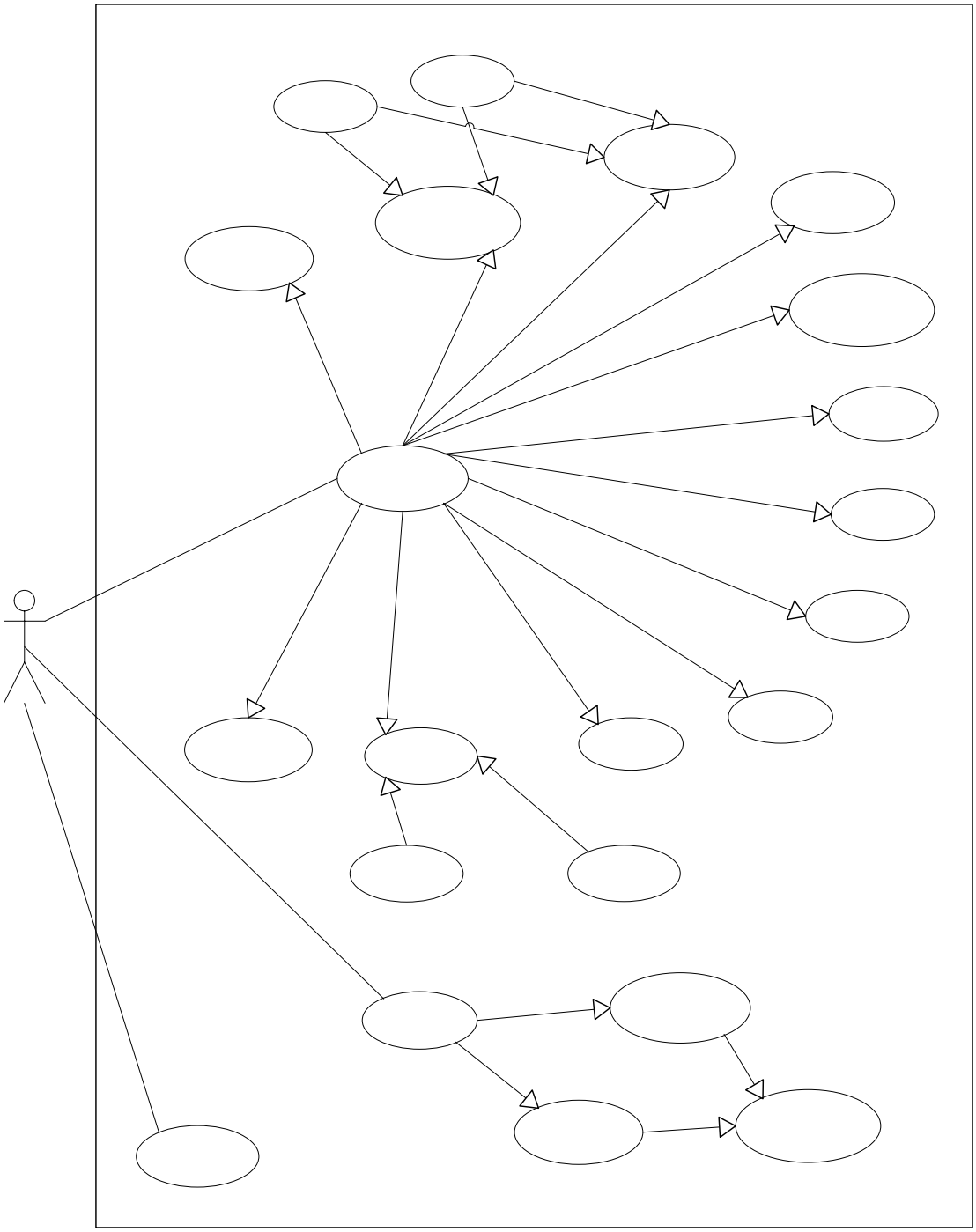


Figura 3.4 – Diagrama de Casos de Utilização.

Como se pode ver pelo diagrama de casos de utilização, são inúmeras as funcionalidades deste editor. Remete-se para o manual do utilizador a explicação de funcionamento de cada uma delas, visto que sai fora do contexto deste texto tal explicação.

Foi dada especial importância ao facto da aplicação ser esteticamente bem parecida e funcional, e para tal foram construídas janelas hierarquicamente organizadas. Ou seja, há uma janela principal, onde se encontra: o menu de ficheiros, um menu de edição de cor de fundo e cor das letras para a janela de código C, um menu onde se escolhe o modo de desenho para o tipo de transição (relacional, sem ser relacional ou ambas), no caso de se tratar de uma máquina de estados do tipo *Machine*, e um menu de geração de código C, onde existem dois sub menus, um para a geração em si e outro para gravar o código gerado. Há um painel interior designado de superfície de desenho que contém uma série de botões para edição rápida e eficiente da máquina de estados, nestes estão incluídos os botões de: acrescentar estado, remover estado, mover estado, seleccionar estado inicial, seleccionar estado final, acrescentar pós condição, acrescentar pré condição, remover transição, editar parâmetros da máquina de estados actual.

No menu de ficheiros existem as funcionalidades de abrir, gravar, nova máquina de estados, e sair da aplicação. Para criar uma máquina de estados nova, o utilizador deverá seleccionar entre os dois tipos existentes, *Machine* ou *Control*, e de seguida será definida uma variável global representativa da máquina de estados actualmente a ser desenhada pelo editor, este tipo escolhido será preenchido como explicado no Cap. 2.3.3.

De modo a que os botões acima referenciados funcionem, foram implementadas funções de escuta de eventos (*event handlers*), eventos estes que passam para as funções correspondentes informação do tipo: que botão foi carregado, quantas vezes foi carregado, onde clicou (posição x, y), etc.

Existe outro painel dentro da janela principal, Fig. 3.5, que só aparecerá quando o utilizador gerar ou já tiver gerado código C, janela esta que mostra o resultado do ficheiro gerado.

Todas as janelas foram programadas usando o *Swing* do Java e todas as interacções entre janelas dentro da aplicação principal. A janela principal da aplicação é uma classe *JFrame* (pertencente à biblioteca javax.swing.JFrame) que difere da classe normal de janelas

Frame, o facto de ter sido escolhida a *JFrame* é que sendo uma versão estendida de *Frame* permite realizar operações ao fechar a janela principal, por exemplo perguntar se o utilizador quer gravar o ficheiro. A janela onde está definida a superfície de desenho e a do ficheiro gerado é da classe *JPanel* (*javax.swing.JPanel*) esta classe foi a escolhida porque é um contentor (*Java container*) de baixo peso computacional, logo todas as operações de desenho efectuadas em cima deste *JPanel* serão mais rapidamente calculadas. As janelas de edição de transições / estados / maquina de estados, são da classe *JDialog* (*javax.swing.JDialog*) visto não necessitarem de processamento interno e apenas servindo para o utilizador inserir informação em modo texto. Desta forma pretendeu-se optimizar o código em termos de memória usada e rapidez de processamento de ambientes gráficos.

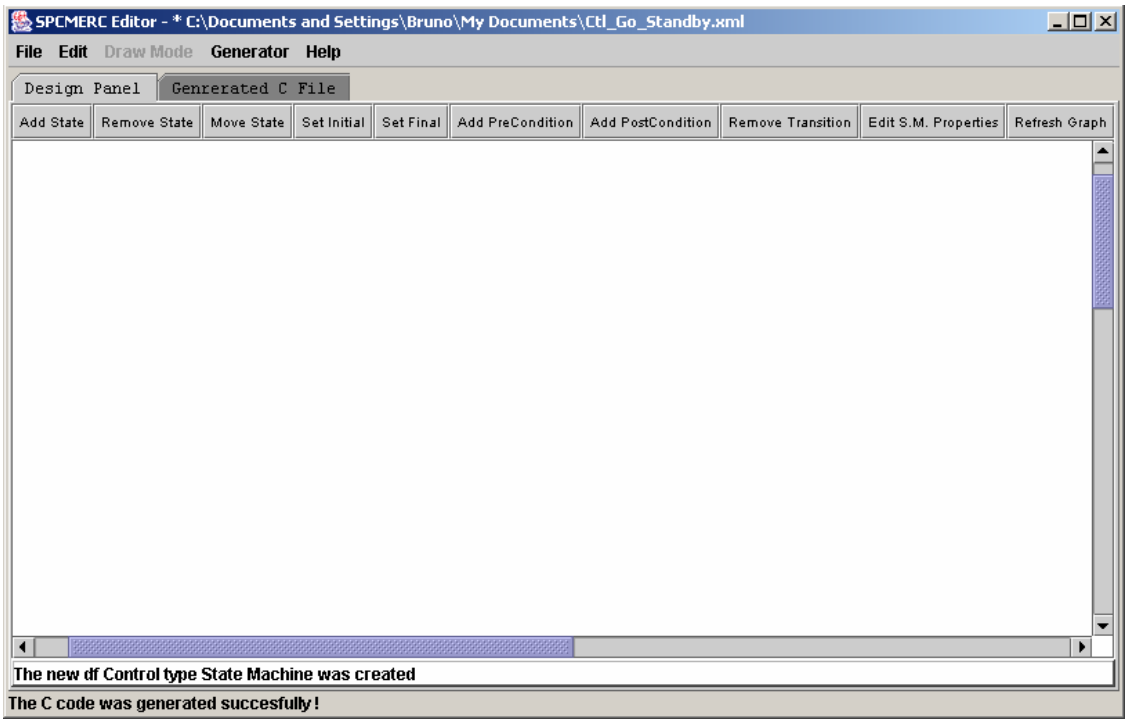


Figura 3.5 – Janela principal.

No sub capitulo seguinte será explicado em pormenor como se modelou os objectos para conseguirem ser representados graficamente.

3.2.2 Descrição da implementação dos objectos gráficos

O leitor lembrar-se-á das questões colocadas no início deste capítulo, explicar-se-á como se chegou à implementação gráfica dos objectos desenháveis. A biblioteca do Java Swing permite criar e desenhar gráficos a duas dimensões e foram usadas essas funções de maneira a representar graficamente a máquina de estados.

Para representar graficamente um estado, foi escolhida uma forma circular, para desenha-la usou-se uma biblioteca de desenho (java.awt.Graphics) que permite desenhar forma oval através da função `drawOval( X, Y, raio_horizontal, raio_vertical)`, Fig. 3.6.

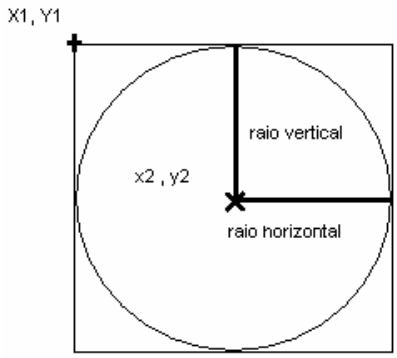


Figura 3.6 – Representação gráfica de um estado

O problema desta ferramenta, é que desenha uma oval na posição X1,Y1, e pretende-se que o centro do estado seja em x2, y2 (posição escolhida pelo utilizador). Como foi o desejo dos autores que os estados tivessem uma forma circular e com o centro na posição "clicada", ficou definido que o raio horizontal fosse igual ao raio vertical e que a posição passada para a função de desenho seria:

$$Estado.X = X\_clicado - raio\_horizontal \quad (3.1)$$

$$Estado.Y = Y\_clicado - raio\ vertical \quad (3.2)$$

Quando o utilizador acrescenta um estado à máquina de estados "clicando" na superfície de desenho, a aplicação cria um novo estado com os atributos preenchidos através da janela de edição de estados (ver sub Cap. anterior) e com a posição X e Y.

Existem algumas diferenças de como um estado, um estado inicial e estado final são representados, o normal consiste numa circunferência preta e sobreposta a este uma amarela de raio um pouco mais pequeno. No caso de ser um estado inicial, a primeira circunferência é verde, e para o final é vermelha. Deste modo os estados consoante sejam normais, final ou inicial, têm uma representação gráfica diferente.

Uma vez que se tenha definido um ou mais estados, é necessário puder seleccioná-los seja para alterar algum atributo, apaga-lo, move-lo, ou para defini-lo como estado origem/destino numa transição. Para tal é necessário haver algo que os caracterize graficamente. No caso do estado o que o caracteriza é o seu centro. Sabendo o raio dos estados, então para puder selecciona-lo basta "clicar" dentro da sua área interior. Seja X e Y a posição devolvida pelo rato, correspondente ao sistema de coordenadas do painel de desenho, então para todos os estados, se distância de X ao centro de um estado for menor que o raio e o mesmo raciocínio para Y, então o estado é dado como seleccionado e é mostrado ao utilizador colorido de cinza, Fig. 3.7.

Seguindo o pseudo código:

```
<PARA todos os estados ϵ lista de estados global>
      <SE   dist(X, estado.x) < raio_Estado
            E dist(Y, estado.y) < raio_Estado>
      <ENTÃO devolve o estado seleccionado>
```

A seguinte figura ilustra o aspecto gráfico dos diferentes tipos de estado.

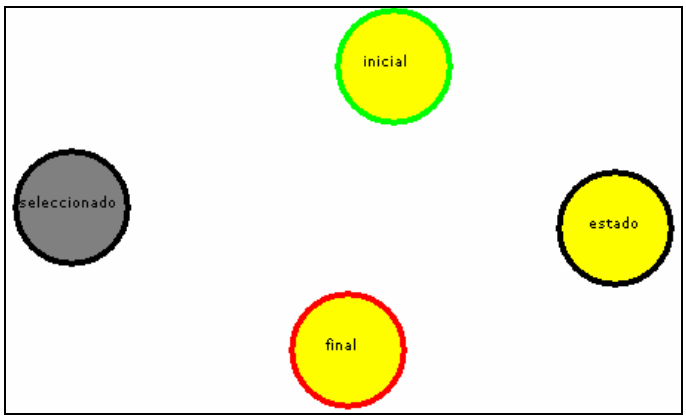


Figura 3.7 – Cores dos estados.

Para completar o desenho de uma máquina de estados, precisamos de unir os estados com transições, e caracterizar graficamente essas transições para que o utilizador possa usando o rato seleccioná-las quer para apagar ou modificar alguns atributos.

A transição é constituída graficamente pelo seu estado origem e destino, logo para que sejam desenhadas foram definidos pontos de entrada e saída no estado, deste modo evita-se usar o centro do estado e não acontecerá que um estado se sobreponha a uma transição, sendo perceptível para o utilizador onde começa e onde acaba uma transição.

Graficamente foram imaginadas oito zonas distintas de destino para uma transição, sendo a origem um estado que se encontra sempre no centro. Todos os cálculos foram feitos estado a estado, percorrendo a lista de estados geral (a lista de estados da máquina de estados), logo o estado destino da transição pode assumir qualquer um dentro destas possibilidades, Fig. 3.8.

Tendo sempre em conta que o estado que está no centro é o de origem, é possível denotar pela imagem que existem 8 zonas bem distintas: os quatro quadrantes (diagonais ao estado de origem) e mais quatro zonas em que a diferença entre a distancia entre os X's e os Y's do estado destino para o estado de origem é pouco maior que o raio (horizontal e vertical ao estado origem).

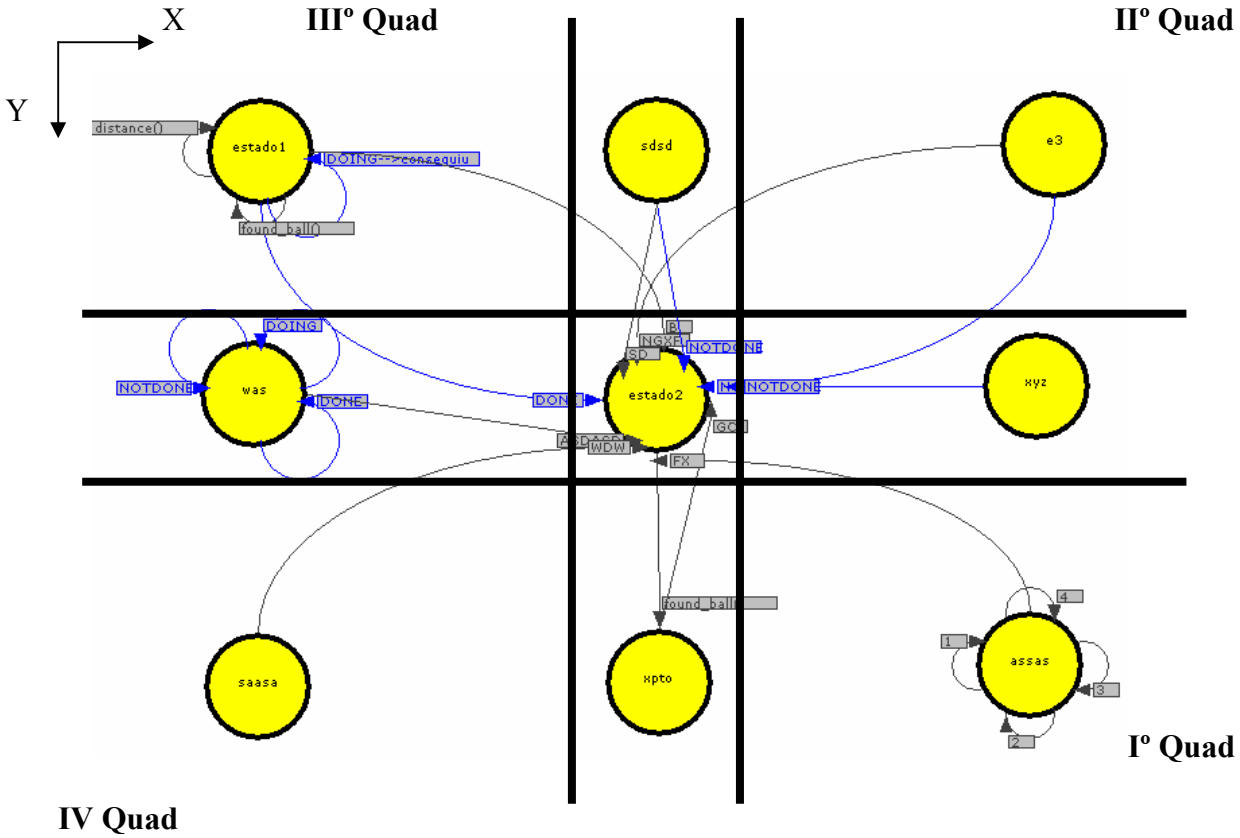


Figura 3.8 – Desenho das transições por zonas.

Enquanto que para desenhar uma transição para qualquer um dos quadrantes se usaram arcos, esta solução já não é possível para as outras quatro zonas, nessas foram usadas rectas.

O desenho das rectas foi o mais fácil de implementar, este faz-se chamando a função `DrawLine(X_inicial, Y_inicial, X_final, Y_final)`, desenha então uma recta unido os dois estados. De notar que a posição final da recta é dinamicamente calculada segundo o algoritmo abaixo explicado que tem em conta o número de transições com o mesmo estado destino.

Para o desenho do arco da transição usou-se uma função da biblioteca gráfica do *Swing* chamada `DrawArc(X, Y, deslocamento_em_X, deslocamento_em_Y, angulo_inicial, deslocamento_angular)`, esta função desenha um arco contido no rectângulo definido por X e Y, com largura de X + deslocamento_em_X e altura de Y + deslocamento_em_Y, e dois ângulos, o de partida e o deslocamento, por exemplo no caso

ilustrado na figura 3.9, o ângulo escolhido é de 180º e o deslocamento de 90º. Como os estados podem alterar a suas posições, a dificuldade inerente deste modo de desenho de arcos foi dinamicamente calcular a sua posição, e como se verá mais tarde as transições também variam dinamicamente de posição segundo o número de transições para o mesmo estado destino.

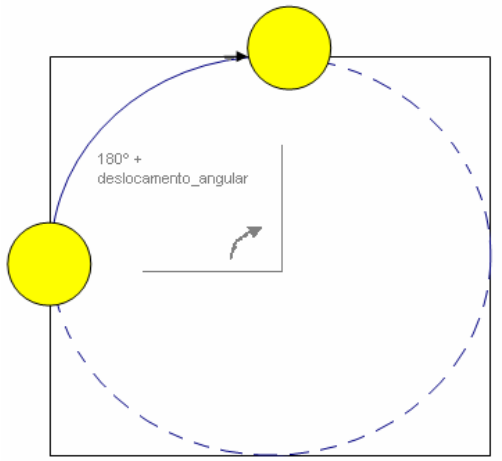


Figura 3.9 - Método de desenho do arco.

Foi usado um algoritmo com dois contadores, um que conta o número de transições originadas num estado e que têm o mesmo estado destino (counter 1), e outro que conta o número total de transições para esse mesmo estado destino independentemente do estado origem (counter 2). Deste modo é possível ir desviando de um certo valor (deslocamento) o ponto final dessa transição. Este algoritmo foi desenvolvido para que as transições nunca sejam coincidentes e possam ser seleccionadas.

```
nt = counter 1 + counter 2                          (3.3)

-1^nt * (radius - nt * Const.gapBetweenArcs)        (3.4)
```

Enquanto que as pré-condições são desenhadas como indica a figura 3.10, as pós são desenhadas com um arco com declive inverso. Supondo que se unem dois estados com uma pré e uma pós-condição, eles terão sempre declives inversos, nunca se sobrepõem.

Também é possível que o estado de origem seja o mesmo de destino. Neste caso o algoritmo criado consistiu em desenhar ovais centradas em oito posições diferentes. Quatro delas para pré-condições e quatro para as pós. Estas zonas são equidistantes e evita sobreposições nestas condições.

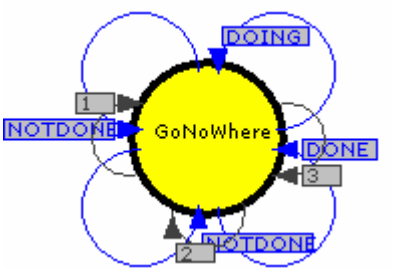


Figura 3.10 – Transições para o mesmo estado.

Explicado o processo de desenho, falta explicar como que se pode seleccionar uma transição. A caracterização gráfica de uma transição é feita através da caixa onde está o nome da mesma. Para se definir a caixa foi preciso encontrar os dois pontos associados, estes são calculados através do comprimento do nome e de um desvio causado pelo desenho da seta direccional.

A dimensão desta caixa (do nome da transição) é calculada dinamicamente usando o tamanho da fonte das letras. São quatro os valores a calcular, o X1 e Y1 referem-se ao canto superior esquerdo da caixa, enquanto que X2 e Y2 ao canto inferior direito. Mediante o quadrante onde a transição será desenhada, os valores para um par X/Y são calculados a partir da posição da seta de maneira a que a caixa fica colada à mesma, os valores do outro par são calculados através do comprimento do nome que está no interior da caixa mais um deslocamento mínimo para o caso de não ter nome, a transição ser seleccionável.

A coloração de cada transição depende do seu tipo. Se se estiver a construir/editar uma máquina de estados do tipo *Machine* haverá três hipóteses, se a transição for uma chamada pré condição, a sua coloração será preta, se for uma pós condição com uma mensagem de sucesso (*behaviour success*) associada será colorida de verde, se pelo contrário tiver uma mensagem de falha (*behaviour failure*) a sua coloração será vermelha. Para qualquer uma destas

transições, se forem relacionais, terão estados alternativos (*Fail States*) e estas serão desenhadas a magenta, Fig. 3.11.

Se a máquina de estados for do tipo *Control* então as pré condições serão desenhadas a preto e as pós a azul.

Foi tomado em conta o aspecto gráfico e as cores usadas porque facilita a legibilidade e mantém a coerência com as máquinas já desenhadas.

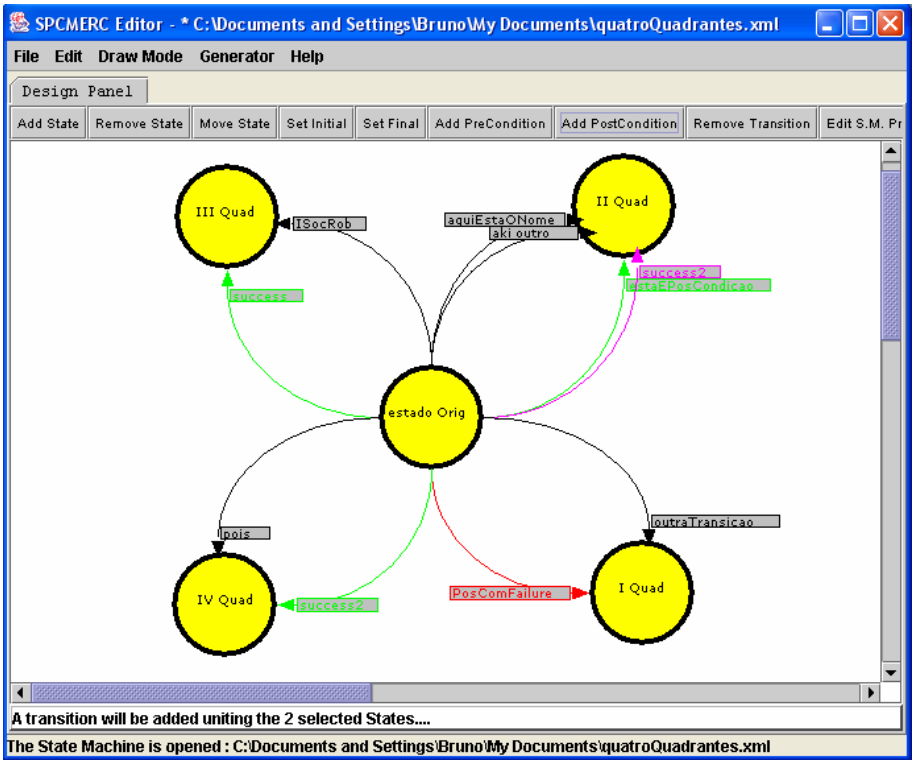


Figura 3.11 – Desenho de transições para os quatro quadrantes.

4 Resultados

Neste capítulo pretende-se mostrar os resultados atingidos e comprovar que os objectivos previstos foram conseguidos.

Foram criados ficheiros XML com o intuito de provocar possíveis erros, conseguindo detectar exactamente em que condições esse erro surgia. Posteriormente, garantiu-se que o código gerado por este sistema é suportado e executado pelos robots do ISocRob.

Constata-se que o desafio da portabilidade entre sistemas operativos foi ganho, ou seja, esta ferramenta pode ser executada em Windows, Linux ou Mac.

4.1 Comparação entre os objectivos planeados e os atingidos

O objectivo planeado, era construir uma ferramenta capaz de gerar código C a partir de uma máquina de estados criada num ambiente gráfico. Este objectivo foi atingido na totalidade com o acréscimo de algumas funcionalidades que os autores resolveram criar (Cap. 3.2.1). Foi traçado um objectivo primário (Cap. 2.4.1) que foi atingido de uma forma muita consistente, que proporcionou uma solidez na continuidade do desenvolvimento.

4.2 Testes realizados

Sendo o tempo de desenvolvimento de software proporcional ao tempo de testes do sistema, foram reservados alguns momentos para a realização de testes.

Para os testes finais, foram utilizados ficheiros XML com informação diversificada, contendo possíveis "armadilhas", tanto para testar a geração de código como para testar a interface gráfica.

4.2.1 Linux vs. Windows

Embora o projecto tenha sido completamente programado sobre um sistema operativo *Windows* já era sabido que iria ser usado em ambientes *Linux*, como tal, a parte final de

concepção da interface foi desenvolvida numa interface de desenvolvimento Java para *Linux*. Deste modo garantiu-se que a aplicação iria funcionar em ambos os sistemas operativos.

Os testes realizados confirmam a portabilidade do código produzido, porque o sistema comportou-se de igual forma em ambos sistemas operativos, ficou desta forma atingido o requisito da portabilidade do sistema.

4.2.2 Testes no campo – *ISocRob*

Com a ajuda do professor coordenador foi construída uma máquina de estados através da aplicação gráfica, Fig. 4.1. Em poucos minutos estava completa, gerou-se o código e foi posto a correr no robot no campo de futebol. Este comportou-se como seria esperado e como foi programado, e fez exactamente o que lhe foi implementado. O comportamento era muito simples e consistia em por o robot a seguir a bola sempre que esta estive no seu campo de visão, quando a perdia o robot dirigia-se para a zona central do terreno.

A seguinte figura mostra o aspecto gráfico da máquina de estados, e o respectivo código gerado através da aplicação e que foi executado pelo robot (Apêndice B).

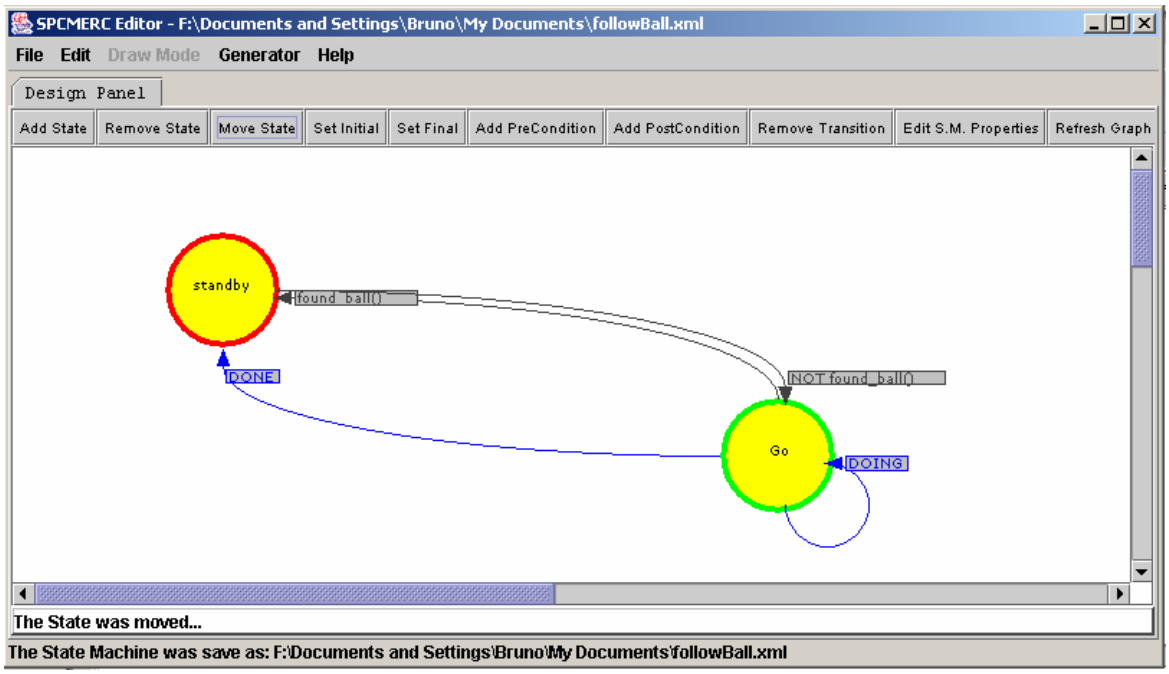


Figura 4.1 – Máquina de estados testada no robot.

4.3 Limitações do Projecto

Como em todos os sistemas de desenvolvimento de *software*, este não foge a regra, e comporta algumas limitações tanto a nível gráfico como a nível da geração de código.

Uma limitação que se prende com a estrutura das linguagens de programação é o facto de o campo nome (*Name*) do estado não poder conter mais de uma palavra, tendo esta apenas caracteres de 'a' a 'z' e de '0' a '9' (não são permitidos caracteres 'ç', '{', '&', 'á', 'ã', etc.).

As bibliotecas geradas no código podem não seguir uma ordem lógica e hierárquica, podendo provocar erros de compilação, ou seja, cabe ao utilizador garantir que a definição de bibliotecas começa da mais geral para a menos geral, e.g., se a biblioteca "`predicates.h`" necessita da biblioteca "`shared.h`", deve-se garantir que a definição de "`shared.h`" aparece primeiro.

As limitações a nível gráfico estão presentes na quantidade de transições para o mesmo estado. Como foi explicado anteriormente, existe um contador que permite contar o total de transições com o mesmo estado destino, e quando este número é muito grande, na ordem da dezena, então as transições tendem a afastar-se do estado destino. Esta limitação pode ser ultrapassada afastando os estados, deste modo não se confunde o estado destino com outro qualquer.

Embora haja um algoritmo que não deixa colocar um estado sobreposto a outro na sua criação, o mesmo já não é possível quando se move o estado. Caberá ao utilizador ter a certeza que não move o estado para uma posição coincidente com outro já existente.

5 Conclusões

O presente trabalho tem uma forte componente de investigação que não facilitou a definição dos requisitos do mesmo, sendo necessárias várias reflexões sobre o tema em análise.

As dificuldades surgiram na abordagem ao projecto ISocRob devido a conceitos novos e complexos. Mesmo assim os autores prosseguiram a sua investigação e procuraram integrar-se rapidamente na sua metodologia.

A permanente interacção entre orientador e alunos serviu de base a um começo promissor e bastante expectante.

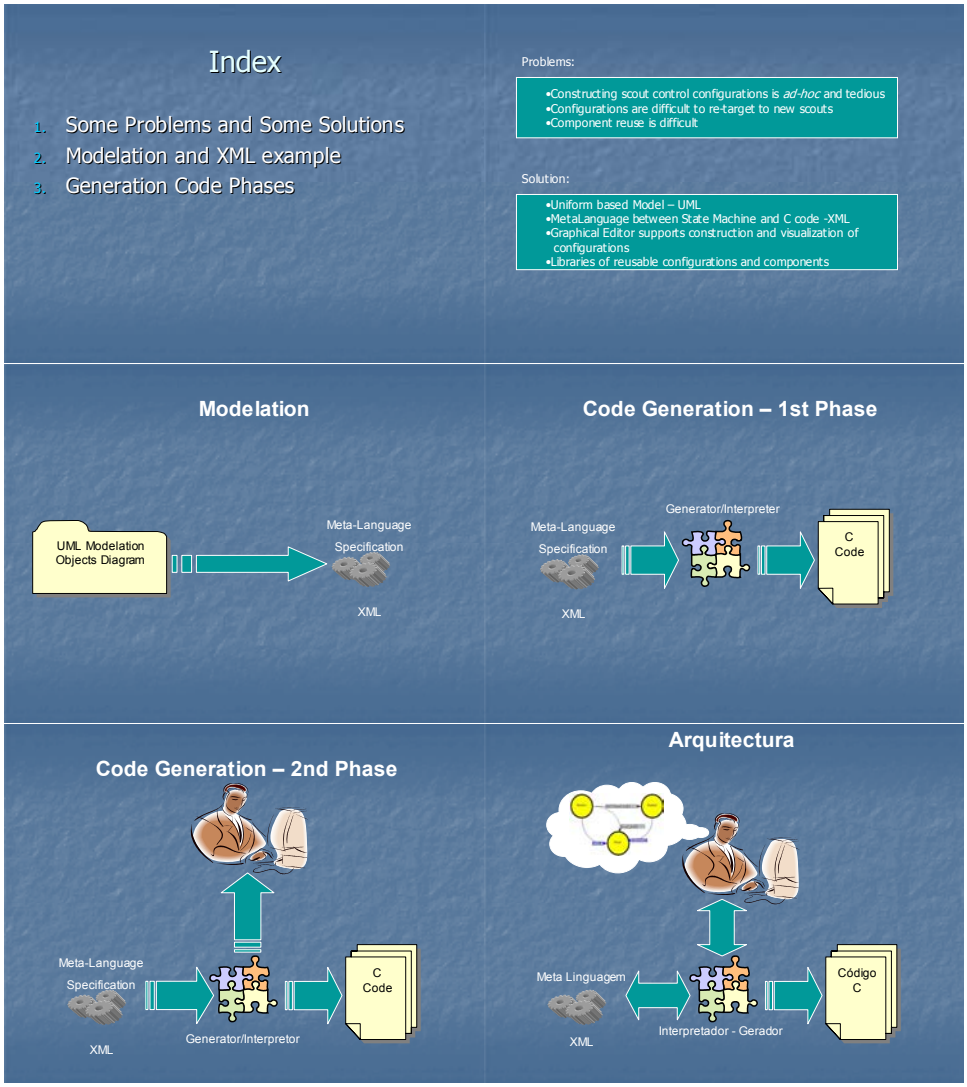
Os autores debruçaram-se sobre o código C relativo aos comportamentos já existentes, e retiraram um modelo UML que é capaz de responder a todos os requisitos para uma boa geração de código, modelo este que foi sofrendo alterações à medida que novos requisitos eram levantados. Consequentemente foi criada uma metalinguagem definida em XML com o intuído de fazer a ligação entre a interface gráfica e a estrutura de dados do domínio.

Confrontados os objectivos previstos com os atingidos, pode-se concluir que o trabalho foi realizado na totalidade dentro dos prazos propostos. Para isto, é de realçar a importância de um bom planeamento, que induziu a uma orientação mais norteada com a possibilidade de gerir os tempos entre as várias tarefas.

Conclui-se então que todo o sistema desenvolvido apresenta uma boa interacção entre utilizador e interface gráfica e que o mesmo no seu todo realiza os objectivos que haviam sido propostos, conduzindo a resultados bastante satisfatórios como haviam sido comprovados.

Como proposta após a conclusão deste trabalho fica a integração desta ferramenta com o simulador gráfico existente, assim é possível após uma geração de código observar os comportamentos definidos no referido simulador.

Apêndice A


Index
1. Some Problems and Some Solutions
2. Modelation and XML example
3. Generation Code Phases
Problems:
•Constructing scout control configurations is ad-hoc and tedious
•Configurations are difficult to re-target to new scouts
•Component reuse is difficult
Solution:
•Uniform based Model – UML
•MetaLanguage between State Machine and C code -XML
•Graphical Editor supports construction and visualization of configurations
•Libraries of reusable configurations and components
Modelation
UML Modelation Objects Diagram
Meta-Language Specification
XML
Code Generation – 1st Phase
Meta-Language Specification
XML
Generator/Interpreter
C Code
Code Generation – 2nd Phase
Meta-Language Specification
XML
Generator/Interpretor
C Code
Arquitectura
Meta Linguagem
XML
Interpretador - Gerador
Código C

Apêndice B

```
/*************************************************
@Filename: followBall.c
@Description: segue a bola
@Version: Sat Jul 12 17:19:48 BST 2003
@Maintainer: SPCMERC
*************************************************/

#include <stdio.h>

#include "../../codigo/shared.h"
#include "../../codigo/constants.h"
#include "../../codigo/guidance.h"
#include "../../codigo/shared2.h"
#include "../../codigo/predicates.h"
#include "blackboard.h"
/*************************************************
Exported Functions
*************************************************/

int init( void );
int destroy( void );
int changeTo( void );
int changeFrom( void );
char *getId( void );
int method( void );

/*************************************************
Static Functions
*************************************************/


/*************************************************
Static Variables
*************************************************/

enum
{
      GO,
      STANDBY
};

static char *strState[] =
{
      "Go",
      "standby"
};

static struct
{
      int currentState;
      int stateCounter;
} private;
```

```
static Goal goal1;
static State state1;
static Settings settings1;
static Actuation function1;

float vLeft, vRight;

/*************************************************
Id of plugin
*************************************************/

static char *id = "followBall";

/*************************************************
@Function: init ()
@Description:
@Return: int: TRUE if sucessfull, FALSE otherwise.
      Important to follow return the correct
      value because  uA might thrash plugin.
@Author: SPCMERC
*************************************************/

int
init (void) {
      private.stateCounter = 0;
      return TRUE;
}

/*************************************************
@Function: destroy ()
@Description: Destruction function of plugin where all
      previously reserved memory is now destroyed.
@Return: int: TRUE if sucessfull, FALSE otherwise.
@Author: SPCMERC
*************************************************/

int
destroy (void) {
      return TRUE;
}

/*************************************************
@Function: changeTo ()
@Description:
@Return: int
@Author: SPCMERC
*************************************************/

int
changeTo (void) {
      GO_STATE(GO, "Initialization");
      return TRUE;
}

/*************************************************
@Function: changeFrom ()
```

```
@Description:
@Return: int
@Author: SPCMERC
************************************************/

int
changeFrom (void) {
      return TRUE;
}

/************************************************
@Function: getID ()
@Description: returns pointer to id string
@Return: char *
@Author: SPCMERC
************************************************/

char
*getId (void) {
      return id;
}

/************************************************
@Function: method ()
@Description:
@Return: int:
@Author: SPCMERC
************************************************/

int
method (void) {

      switch ( GET_STATE() ) {
       case GO:
            /*************** SENSOR MODE *****************/
            visionMode ("self");


            /************** PRECONDITIONS ****************/
            if( found_ball() ) {
                  GO_STATE(STANDBY, "");
                  break;
            }

            /************** BODY ****************/
            state1.sonars=sonarGet();
            state1.posture=odometry;
            motorsGetVW(&state1.v, &state1.w);
            getHomePosture(&goal1.posture);

      settings1.avoid_ball=getBallPosition(&state1.ball.x,&state1.ball.y);

            /*********** FUNCTION & POSTCONDITIONS ***********/
            function1 = potential(&goal1, &state1, &settings1);

            switch(function1.return_value) {
                  case DONE:
```

```
                        GO_STATE(STANDBY, "");
                        break;
                  case DOING:
                        motorsSetVW(function1.v, function1.w);
                        break;
                  default:
                        break;
            }
            break;

       case STANDBY:
            /*************** SENSOR MODE *****************/
            visionMode ("up");


            /************** PRECONDITIONS ****************/
            if( NOT found_ball() ) {
                  GO_STATE(GO, "");
                  break;
            }

            /************** BODY *****************/
            freeRotate(&vLeft, &vRight,
bbGetLocalFloat("vision.ball.rangle",0), 0);
            motorsSet(vLeft, vRight);
            break;

       default:
            printf("Control.%s: Landed on unknown state\n", id);
            GO_STATE(GO, "error in state");
      }

      return TRUE;

}
```

Apêndice C

6 Referências

[1] MissionLab, http://www.cc.gatech.edu/ai/robot-lab

[2] Charon, http://www.cis.upenn.edu/mobies/charon

[3] Machado, Bruno; Cardoso, Cláudio, Pagina de suporte ao projecto, http://mega.ist.utl.pt/~bpvm

[4] Silva, Alberto; Videira, Carlos, "UML Metodologias e Ferramentas CASE", 2001

[5] Lima, Pedro: Current Status of the Project. ISR, Janeiro 2003

[6] Lafortune, S.; Cassandras, Chistos G., "Introduction to Discrete Event Systems", Kluwer Acadec Publ., 1999

[7] Flanagan, D. – Java Examples in a Nutshell, 2000

[8] Lima, Pedro; Ventura, Rodrigo; Aparício, Pedro; Custódio, Luís, "A Functional Architecture for a team of Fully Autonomous Cooperative Robots", in RobotCup 99 – Robot Soccer World Cup III, Springer-Verlag, Berlin, 1999

[9] Toscano, Luís, "Desenvolvimento de Modelos de Cooperação para uma sociedade de Agentes", TFC LEEC, DEEC/Instituto Superior Técnico, 1049-001 Lisboa, 2001

[10] Costelha, Hugo: http://socrob.isr.ist.utl.pt/private/docs/CodingRules.htm, Junho 2003