RUITER BRAGA CALDAS

MODELAGEM, VERIFICAÇÃO FORMAL E CODIFICAÇÃO DE SISTEMAS REATIVOS AUTÔNOMOS.

Tese apresentada ao Programa de Pós--Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais - Departamento de Ciência da Computação como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Sérgio Vale Aguiar Campos

Belo Horizonte

Dezembro de 2009

© 2009, Ruiter Braga Caldas.
Todos os direitos reservados.

Caldas, Ruiter Braga
C145m Modelagem, Verificação Formal e Codificação de Sistemas Reativos Autônomos. / Ruiter Braga Caldas. — Belo Horizonte, 2009
xxiii, 133 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Orientador: Prof. Sérgio Vale Aguiar Campos

1. Sistemas reativos - Teses. 2. Modelo formal - Teses. 3. Verificação de modelos - Teses.
I. Orientador. II. Título.

CDU 519.6*23(043)

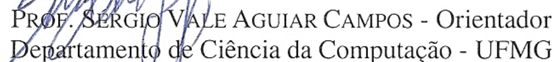
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

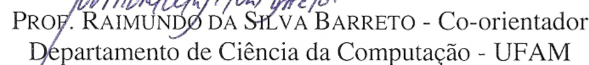
Modelagem, verificação formal e codificação de sistemas reativos autônomos

RUITER BRAGA CALDAS

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

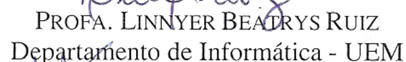


PROF. SÉRGIO VALE AGUIAR CAMPOS - Orientador
Departamento de Ciência da Computação - UFMG

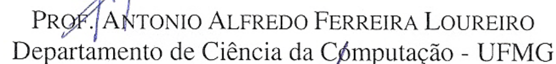


PROF. RAIMUNDO DA SILVA BARRETO - Co-orientador
Departamento de Ciência da Computação - UFAM

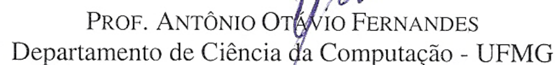
PROF. DJAMEL FAWZI HADJ SADOK
Centro de Informática - UFPE



PROFA. LINNYER BEATRYS RUIZ
Departamento de Informática - UEM



PROF. ANTONIO ALFREDO FERREIRA LOUREIRO
Departamento de Ciência da Computação - UFMG



PROF. ANTÔNIO OTÁVIO FERNANDES
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 17 de dezembro de 2009.

Dedico este trabalho a:
meu pai Rodolfo, cujo exemplo motiva a minha conduta,
minha mãe Francisca, cuja bondade é exemplo de amor,
minha esposa Ruth, pela confiança na minha vitória,
minha filha Agatha, pela sua paixão pela vida,
a todos que, de alguma forma, me ajudaram neste trabalho.

Agradecimentos

Agradeço a Deus, razão de tudo.
Agradeço ao meu orientador, Prof. Sérgio Campos, a quem tive o prazer de encontrar num momento tão especial, quando andava perdido pelo universo da pesquisa acadêmica, ele apontou a porta de saída.
Agradeço ao Prof. Claudionor Coelho que me recebeu na porta de entrada.
Agradeço a toda a minha família, minha esposa Ruth, minha filha Agatha, meus pais Rodolfo e Francisca e meus irmãos Rosangela, Roberto, Rosana, Rosilane e Junior, que acreditaram em mim até o fim.
Agradecer a instituição UFMG, um lugar cheio de vida acadêmica, onde se respira, conversa e se produz pesquisa. Agradeço a todas as pessoas que encontrei neste universo, as secretárias sempre com um sorriso no rosto. Aos professores, que no início chamava de mestres e agora chamo de amigos.
Agradeço ao DCC/UFAM, que permitiu que eu completasse a minha missão.
Nesse caminho encontrei muitas pessoas que dividiram comigo os momentos de tensão e as alegrias das nossas pousadas nos sítios, nos passeios em família e na turma do futebol, que ficarão pra sempre na minha lembrança e na nossa história. A todos vocês meus amigos obrigado por tudo. Espero encontrá-los em breve. Gostaria de citar o nome de todos vocês, mas corro o risco de esquecer de listar o nome de alguém importante, por isso peço perdão, sintam-se citados.

"A person who never made a mistake
never tried anything new."
(Albert Einstein)

Resumo

Os sistemas computacionais são utilizados nas mais variadas áreas da vida cotidiana, desde o controle das contas bancárias até os pacientes nos hospitais. Nas aplicações onde vidas humanas ou altos investimentos estão em risco, a qualidade dos sistemas computacionais tem uma importância fundamental para eliminar ou reduzir as falhas. A utilização de modelos formais no processo de desenvolvimento de sistemas apresentam uma resposta ao problema citado, pois oferecem uma descrição das partes mais relevantes do sistema com um nível adequado de abstração. Este trabalho apresenta o **Modelo de Desenvolvimento *Bare***, para o desenvolvimento de aplicações em Sistemas Reativos Autônomos e mostra a possibilidade de modelar aplicações para diversas área. O modelo inicia com a descrição da aplicação por meio de uma máquina de estados finito, chamada *X-machine*. A aplicação a ser desenvolvida é a peça principal, ou núcleo, de um sistema reativo onde os eventos detectados no ambiente são capturados, avaliados com base no estado corrente da máquina, produzindo como resposta ao evento uma transição de estado e um elemento de atuação, que pode ser um novo evento ou uma comunicação. No Modelo *Bare* a especificação da aplicação é feita usando uma ferramenta gráfica chamada *GeradorXM*, após a modelagem da aplicação a X-machine resultante é transformada para um modelo tabular, onde cada linha é independente e contém informação suficiente para executar uma computação. A aplicação no modelo tabular é carregada na plataforma alvo, onde é interpretada por um programa pequeno, chamado *ExecutorXM*, que é o responsável pela execução da aplicação. Antes de executar a aplicação um modelo do sistema é gerado pelo *GeradorXM* para ser utilizado como entrada para a ferramenta de verificação de modelos NuSMV. Com isso as propriedades desejáveis para a aplicação podem ser verificadas para confirmar a sua correção. A execução do modelo *Bare* fecha um ciclo de desenvolvimento de aplicação para sistemas reativos autônomos por meio de um modelo formal, com geração automática de código para um interpretador e verificação de propriedades para o modelo do sistema.

Palavras-chave: Sistemas Reativos, Especificação Formal, Verificação de Modelos.

Abstract

The computer systems are used in many areas, since bank accounts until patient's monitoring. In applications where human lifes or high investments are critical, the systemt́s quality is fundamental to reduces or eliminate failures. The formal methods are used to describes parts of the system using appropriate level of abstraction. This thesis presents the Bare Model to development of applications in Autonomous Reactive Systems and shows his capabilities to development a lot of applications. The model starts with a application description by a finite state machine, called *X-machine*. The application acts as a centerpiece, or core, of a reactive system. The events detected in the environment are captured and evaluated, based on the current state of the machine. The response to the events detected are the transitions of states and the actions, which can be an new event or just a communication. On the Bare Model, the specification of the application is performed using a graphical tool called *GeradorXM*. After the design, the X-machine is transformed to a tabular model, where each line is independent and contains enough information to perform a computation. The tabular model is uploaded into the target hardware, where it is interpreted by a slight code, called *ExecutorXM*, which is responsible for his execution. Before to run the application, a *model of the system* can be generated to be used as an input to the *NuSMV*, which is a model checking's tool. Thus the desirable properties for the application can be checked to confirm its correctness. Thus closing a cycle of application development for autonomous reactive systems by means a formal model, with automatic code generation which is executed by an translator and using formal verification of properties for the system model.

Keywords: (.Reactive Systems, Formal Specification, Model Checking)

Lista de Figuras

1.1 Fases do Método Proposto 4

2.1 Esquema Geral da X-machine 10
2.2 Estrutura Kripke de M_e 16
2.3 Árvore de Computação de M_e 17
2.4 Representação Semântica da Lógica CTL 18
2.5 Máquina de Estados do Sistema 22
2.6 Reconfiguração Dinâmica Baseada em Cenários 24
2.7 Contextos de Reconfiguração Dinâmica 25

3.1 Esquema Geral de um Sistema Reativo Autônomo 29
3.2 Fases do Modelo de Desenvolvimento 31
3.3 Aplicação Blink 32
3.4 Interligação de X-machines 38
3.5 Esquema para a Aplicação Blink 40
3.6 Modelo do Sistema para a Aplicação Blink 45
3.7 Reconfiguração Dinâmica no Modelo *Bare* 48
3.8 Esquema de Reconfiguração 50

4.1 Grafo de Execução do Blink 52
4.2 Algoritmo do *ExecutorXM* 55
4.3 Interfaces do Robô NXT 56
4.4 Sensores e Atuadores do Robô NXT 57
4.5 Função *ExecutorXM* 59
4.6 Função *LeCorr* 59
4.7 Gerador de Aplicações *Bare* 62

5.1 Aplicação de Detecção de Intruso 70
5.2 Geração da Aplicação de Detecção de Intruso 73

5.3 Tabelas Auxiliares para Detecção de Intruso 75
5.4 Execução da Aplicação de Detecção de Intruso no PC 76
5.5 Configuração Robô NXT para os Experimentos 78
5.6 Interface para Detecção de Eventos 79
5.7 Rotina do ExecutorXM em pbLua 80
5.8 Rotina para Capturar o Estado Corrente 81
5.9 Tabela com Trecho da Aplicação de Detecção de Intruso 82
5.10 Execução da Aplicação de Detecção de Intruso no Robô NXT 83
5.11 Modelo do Sistema para Aplicação de Detecção de Intruso 86
5.12 Verificação da Aplicação de Detecção de Intruso 87
5.13 Verificação da Aplicação de Detecção de Intruso (cont) 88
5.14 Aplicação de Caminhamento Autônomo 89
5.15 Gerador da Aplicação de Caminhamento Autônomo 93
5.16 Tabelas Auxiliares para Caminhamento Autônomo 95
5.17 Interface para Detecção de Obstáculos 96
5.18 Interface para Atuação de Eventos 97
5.19 Tabela com Trecho da Aplicação de Caminhamento Autônomo 98
5.20 Arquivo de Inicialização para o robô NXT 98
5.21 Execução da Aplicação Caminhamento Autônomo 99
5.22 Modelo do Sistema para Aplicação de Caminhamento Autônomo 101
5.23 Verificação da Aplicação de Caminhamento Autônomo 102
5.24 Verificação da Aplicação de Caminhamento Autônomo (cont.) 103
5.25 Algoritmo para Posicionamento de Nós Sensores Usando Robô 106
5.26 Posicionamento de Nós Sensores Usando Robô 107
5.27 Aplicação de Posicionamento de Nós Sensores usando Robô 108
5.28 Gerador da Aplicação de Posicionamento de Nós com Robô 110
5.29 Tabelas Auxiliares para o Posicionamento de Nós Sensores 112
5.30 Interface para Detecção do Host Alvo 113
5.31 Interface para Atuação com Movimento 114
5.32 Tabela com Trecho da Aplicação do Posicionamento de Nós Sensores . . . 115
5.33 Arquivo de Inicialização 115
5.34 Execução da Aplicação Posicionamento de Nós Sensores usando Robô . . . 116
5.35 Modelo do Sistema para Aplicação de Posicionamento de Nós Sensores . . 118
5.36 Verificação da Aplicação de Posicionamento de Nós Sensores 119
5.37 Verificação da Aplicação de Posicionamento de Nós Sensores 120

Lista de Tabelas

2.1 Modelo do Sistema na Linguagem NuSMV 21

3.1 Definição de Tipos 34
3.2 Tabela para a Aplicação Blink 43

4.1 Tabela de Execução para Aplicação Blink 54
4.2 Tabela de Eventos Monitorados 54
4.3 Tabela de Condições 54
4.4 Trecho Inicial da Aplicação Blink 64
4.5 Continuação da Tabela para a Aplicação Blink 65

5.1 Tabela para a Aplicação Detecção de Intruso 74
5.2 Tabela de Execução para Aplicação de Caminhamento Autônomo 94
5.3 Tabela de Execução para Aplicação Posicionamento de Nós com Robô . . . 112

Sumário

Agradecimentos **vii**

Resumo **xi**

Abstract **xiii**

Lista de Figuras **xv**

Lista de Tabelas **xvii**

1 Introdução **1**

1.1 Motivação 2
1.2 Definição do Problema 3
1.3 Objetivo Geral 3
1.4 Método Proposto 4
1.5 Contribuições da Tese 4
1.6 Estrutura da Tese 5

2 Referencial Teórico **7**

2.1 Sistemas Reativos 7
2.2 X-machines 10
2.3 Verificação de Modelos 11
2.3.1 Lógica Temporal Ramificada 13
2.3.2 Lógica Temporal Linear 17
2.4 NuSMV 19
2.5 Reconfiguração de Aplicação 22

3 Modelo de Desenvolvimento *Bare* **27**

3.1 Componentes do Modelo 29
3.2 Modelagem da Aplicação 32

3.2.1 Modelo Dinâmico para *X-Machine* 35
3.3 Mapeamento para o Modelo Tabular 40
3.4 Verificação Formal do Modelo *Bare* 43
3.5 Reconfiguração no Modelo *Bare* 48

4 Implementação do Modelo *Bare* **51**
4.1 Modelo de Execução *Bare* 51
4.2 Executor do Modelo *Bare* 53
4.3 Implementação do Executor do Modelo *Bare* 56
4.3.1 O Robô NXT 57
4.3.2 A Linguagem LUA 57
4.3.3 Implementação do Executor em LUA 58
4.3.4 Gerador de Aplicações *Bare* 61

5 Aplicações do Modelo *Bare* **67**
5.1 Detecção de Intruso 69
5.1.1 Detecção de Intruso no Modelo *Bare* 69
5.1.2 Gerador da Aplicação de Detecção de Intruso 71
5.1.3 Mapeamento para o Modelo Tabular 72
5.1.4 Execução da Aplicação no PC 75
5.1.5 Execução da Aplicação no Robô 77
5.1.6 Verificação para o Modelo da Detecção de Intruso 85
5.2 Caminhamento Autônomo de Robôs 89
5.2.1 Caminhamento Autônomo no Modelo *Bare* 90
5.2.2 Gerador da Aplicação de Caminhamento Autônomo 92
5.2.3 Mapeamento para o Modelo Tabular 92
5.2.4 Execução da Aplicação de Caminhamento no Robô 94
5.2.5 Verificação para o Modelo do Caminhamento Autônomo 100
5.3 Posicionamento de Nós Sensores usando Robôs 104
5.3.1 Algoritmo de Posicionamento de Nós Sensores 105
5.3.2 Posicionamento de Nós Sensores no Modelo *Bare* 106
5.3.3 Gerador da Aplicação de Posicionamento de Nós com Robô . . . 111
5.3.4 Mapeamento para o Modelo Tabular 111
5.3.5 Execução da Aplicação de Posicionamento no Robô 111
5.3.6 Verificação para o Modelo do Posicionamento de Nós Sensores . 117

6 Conclusão e Trabalhos Futuros **121**
6.1 Trabalhos Futuros 125

Referências Bibliográficas **129**

Capítulo 1

Introdução

Sistemas computacionais estão presentes em quase todas as áreas de atividades no mundo moderno. As pessoas aprenderam a confiar nesses sistemas e hoje permitem que eles controlem partes importantes das suas vidas. Sistemas computacionais são utilizados para controlar as contas bancárias, os carros, os aviões, as comunicações, as plantas industriais e até mesmo os pacientes nos hospitais. Considerando as aplicações, onde vidas humanas ou altos investimentos estão em risco, a qualidade dos sistemas computacionais passa a ter uma importância fundamental para que os riscos envolvidos sejam evitados ou minimizados. Para que isso ocorra é necessário a utilização de métodos e técnicas rigorosos para seu desenvolvimento. Como uma resposta a essa necessidade, os métodos formais fornecem modelos para descrever as partes relevantes do sistema em um nível adequado de abstração, por meio de gráficos ou descrições da especificação do sistema, o que constitui uma possibilidade viável para alcançar o nível de confiabilidade necessário no processo do desenvolvimento dos sistemas computacionais. Existe hoje uma grande demanda por aplicações em que o computador está conectado a processos externos e interage de forma dinâmica com o meio ambiente onde ele está inserido, sobretudo onde a resposta do sistema computacional deve ser de forma imediata aos sinais provenientes destes processos externos. Os sistemas que apresentam estas características, de interagir com um ambiente externo, mantendo um relacionamento dinâmico com esse ambiente são os chamados sistemas reativos [Harel & Pnueli, 1985]. Uma especialização dessa classe de sistemas além de interagir frequentemente com seu ambiente sem finalização, ou seja, além de executar por longos períodos de tempo, deve interagir com o meio ambiente por meio de elementos atuadores, tais como elementos de comunicação ou movimentação, como os que estão presentes nos sistemas robóticos e nas redes de sensores sem fio ou até mesmo na Internet. A esse tipo de sistema reativo que realizam tarefas em ambientes não estruturados e sem intervenção

contínua de humanos denominaremos de Sistemas Reativos Autônomos. O objetivo deste trabalho de pesquisa é um estudo sobre a utilização de um modelo formal para a especificação de aplicações em Sistemas Reativos Autônomos. O interesse por este tipo específico de aplicação é que ela difere do modelo convencional das aplicações, devido a sua necessidade de interagir frequentemente com seu ambiente sem uma finalização, com isso, elas não podem ser modeladas adequadamente como um sistema clássico de transformação, ou seja, os quais são definidos a partir de uma entrada, seguido da manipulação dessa entrada e a correspondente produção de uma saída [Harel & Pnueli, 1985].

1.1 Motivação

As novas aplicações dos sistemas computacionais nas mais diversas áreas da indústria levam a uma necessidade urgente do desenvolvimento de novos métodos que garantam a qualidade dos produtos finais. A qualidade deve estar presente desde a concepção, especificação, projeto, implementação e verificação dos sistemas. Uma falha que venha a ocorrer num sistema computacional além de custar milhares de reais pode levar a consequências trágicas. No ano de 2005 a fabrica Toyota anunciou o *recall* de 160.000 unidades do seu recém lançado carro híbrido modelo "Prius", devido a problemas relatados de veículos que acendiam as luzes sem nenhum motivo e motores a gasolina que paravam de funcionar inesperadamente. Entre os anos de 1985 e 1987 ocorreu uma série de acidentes com a máquina *Therac-25*, que expôs os pacientes sob tratamento de radioterapia a super-dosagem de radiação, ocasionando a morte de 6 pacientes [Baier & Katoen, 2008]. Em junho de 1996 o foguete *Ariane-5* trabalhando com o programa desenvolvido para o foguete *Ariane-4* sofre um erro na rotina de aritmética no computador de voo. O erro foi na conversão do número de ponto flutuante de 64-bit para um inteiro de 16-bit, causando um estouro de memória que desligou o computador reserva e em seguida o computador primário, o que levou a desintegração do foguete 36 segundos após o seu lançamento [Baier & Katoen, 2008]. Um erro de software no sistema de manipulação de bagagem retardou a abertura do aeroporto de Denver em 9 meses, com uma perda de 1.1 milhão de Dólares por dia [Baier & Katoen, 2008]. Estes exemplos destacam a importância da verificação e teste realizados tanto no componente de hardware quanto no componente de software dos sistemas computacionais modernos. Uma vez que a aplicação dos sistemas computacionais se espalha por setores ligados diretamente a nossa vida cotidiana, a preocupação com a execução correta dessas aplicações passa a fazer parte das restrições das aplicações. Agora não apenas a utilização

da tecnologia é importante, mas surge a necessidade de implementar mecanismos para auxiliar na construção das aplicações que, além de facilitar a sua construção possam, também, viabilizar maneiras de provar que a aplicação se comportará corretamente para as situações que ela foi planejada.

A partir das necessidades expostas defendemos a especificação das aplicações em Sistemas Reativos Autônomos por meio de um modelo formal, devido ao alto nível de confiança inerente aos métodos formais. Os métodos formais são ferramentas baseadas em modelos e técnicas matemáticas para especificação e verificação de sistemas. Depois de desenvolvida a especificação do sistema, esta pode ser usada para provar propriedades do sistema ou testar fontes de erros ou possíveis falhas. Por exemplo, podemos verificar como o sistema muda de um estado para outro ou porque não muda. Além disso uma especificação formal pode ser usada como entrada para uma ferramenta de verificação formal ou validação. Por ser um modelo formal com alto nível de abstração, este poderá ser convertido para outros modelos já consolidados e assim fazer uso de ferramentas existentes no mercado.

1.2 Definição do Problema

O problema a ser tratado por esta tese é: *Modelar uma aplicação tanto para analisar as propriedades de interesse, quanto para sintetizar automaticamente o código a ser executado em uma plataforma alvo.*

1.3 Objetivo Geral

O objetivo deste trabalho é a geração automática de código para as aplicações na área de Sistemas Reativos Autônomos.

Os objetivos específicos são:

- Utilizar um modelo formal para a especificação das aplicações em Sistemas Reativos Autônomos.

- A partir do modelo formal gerar o modelo do sistema para verificar as propriedades desejáveis para a aplicação.

- Sintetizar a aplicação para ser executada em uma plataforma alvo.

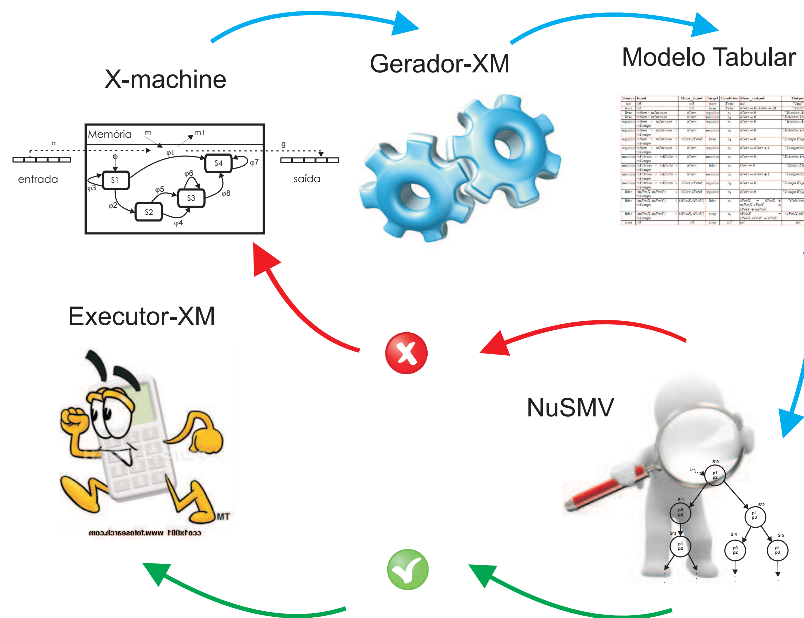


Figura 1.1: Fases do Método Proposto

1.4 Método Proposto

A Figura 1.1 apresenta as fases do método proposto para o desenvolvimento de aplicações em Sistemas Reativos Autônomos. O modelo formal a ser adotado será a *x-machine* [Holcombe, 1988]. Depois de modelada, a aplicação é transformado em um formato tabular o qual será executado na plataforma alvo. Para executar a aplicação será utilizado um ambiente de execução (*runtime environment*). Antes da aplicação ser executada um modelo do sistema pode ser verificado por uma ferramenta de verificação de modelos, com o objetivo de identificar as propriedades de interesse para a aplicação.

1.5 Contribuições da Tese

- A proposta de um modelo para o desenvolvimento de aplicações para Sistemas Reativos Autônomos com aplicação em Rede de Sensores Sem Fio e Robótica.
- A utilização de um modelo formal para a especificação das partes componentes das aplicações dos Sistemas Reativos Autônomos.
- Utilização da especificação formal das aplicações como entrada para um sistema de verificação formal de modelos.
- Utilização de um modelo para programação das aplicação baseado em tabelas de alto nível com as instruções para execução. As tabelas serão executadas por um

interpretador carregado no hardware alvo.

- Facilidade de programação e reprogramação dos Sistemas Reativos Autônomos por meio da transmissão para o hardware, das linhas individuais para compor as tabelas das aplicações, ou das linhas com as atualizações para as aplicações.
- Geração automática das tabelas com a codificação das aplicações para execução pelo interpretador.
- Geração automática do Modelo do Sistema, a partir da tabela da aplicação, para uma ferramenta de verificação formal de modelos.
- Descrição de um Modelo Dinâmico que possibilita a execução das X-Machines.

1.6 Estrutura da Tese

A tese está estruturada da seguinte forma: no capítulo 1, é feita a introdução junto com a motivação para a utilização de modelos formais no desenvolvimento de aplicações para sistemas reativos autônomos, os objetivos gerais da pesquisa e as contribuições da tese. No capítulo 2 é apresentado a fundamentação teórica das áreas de abrangência da tese: sistemas reativos, x-machines, verificação de modelos, redes de sensores sem fio e reconfiguração de software. No capítulo 3 é apresentado a descrição completa do Modelo de Desenvolvimento *Bare*, por meio do desenvolvimento de uma aplicação simples em RSSF chamada *Blink*. O modelo inicia com a especificação da aplicação em x-machine, em seguida a transformação da x-machine para o modelo tabular, por meio do *Gerador-XM* e a geração do modelo do sistema como entrada para a ferramenta de verificação formal de modelos *NuSMV*. No capítulo 4 são mostrados detalhes do modelo de execução para o Modelo de Desenvolvimento *Bare*, detalhes da implementação do interpretador do modelo tabular para *PC* e para o Robô NXT, chamado *Executor-XM*. No capítulo 5 são apresentados exemplos de aplicações completas modeladas usando o Modelo de Desenvolvimento *Bare*, a transformação para o formato tabular e respectiva execução por meio de um simulador para o *PC* e no Robô NXT. No capítulo 6 é feita uma discussão sobre os resultados obtidos, a conclusão e as propostas para trabalhos futuros.

Capítulo 2

Referencial Teórico

2.1 Sistemas Reativos

Sistemas reativos são uma designação proposta por [Harel & Pnueli, 1985], para os sistemas que interagem de forma dinâmica com o ambiente externo no qual estão inseridos, ou seja, são sistemas baseados em eventos que reagem a estímulos internos ou externos do ambiente, de forma a produzir resultados corretos dentro de intervalos de tempo determinado pelo próprio ambiente [Harel & Pnueli, 1985]. O comportamento de um sistema reativo pode ser descrito por uma execução cíclica de: espera por estímulo, calcula respostas e emite sinais de saída. Os sistemas computacionais, além das divisões tradicionais comumente observadas (Sequenciais x Concorrentes, Determinísticos x Não-Determinísticos), podem ser divididos em duas outras classes distintas: os Sistemas Transformativos e os Sistemas Reativos [Harel & Pnueli, 1985]. Dentre as técnicas de especificação de sistemas, as técnicas formais são as mais indicadas para a modelagem de sistemas reativos, dado que este tipo de sistema deve funcionar com alto grau de confiabilidade e segurança. As técnicas de especificação formal possibilitam o desenvolvimento de especificações consistentes, completas e sem ambiguidades, para isso fazem uso de linguagem com sintaxe e semântica bem definida.

Uma forma de estudar, compreender e projetar sistemas reativos é a utilização da divisão em camadas. Um programa reativo, ou seja, que implementa um sistema reativo, pode ser convenientemente composto por 3 camadas, que são : Interface, Núcleo Reativo e Manipulação de Dados [Berry & Gonthier, 1992]. A camada de Interface recebe os estímulos (sinais de entrada) oriundos do ambiente externo e devolve sinais de saída, transformando eventos físicos externos em sinais lógicos internos e vice-versa. Ela manipula as interrupções, realiza leitura dos sensores e ativa os atuadores. A camada de Núcleo Reativo contém a lógica do sistema. Realiza as computações

necessárias, de acordo com os sinais de entrada, e gera sinais de saída, se houver algum, para o ambiente externo. Ela decide qual computação e quais saídas devem ser geradas como reação às entradas. Já a camada de Manipulação de Dados executa computações que são requeridas pelo núcleo reativo. O núcleo reativo constitui a parte central e mais complexa de um sistema reativo [Berry & Gonthier, 1992][Toscani, 1993]. Essa estruturação em camadas permite que os sistemas reativos sejam estudados, projetados e implementados de forma completamente separada do ambiente externo com o qual interagem, pois não levam em consideração os detalhes de comunicação com esse ambiente. Sistemas reativos podem ser analisados a partir de um estado inicial, aguardando por uma entrada. Para cada entrada que ocorre ele executa alguma computação e passa para um novo estado, com isso os modelos baseados em autômatos são ideais para representar esse tipo de sistema.

O sistemas reativos podem ser caracterizados por seus "estados". O estado é uma descrição instantânea do sistema no qual são capturados os valores das variáveis num instante de tempo particular. Também devem ser considerados como os estados do sistema são alterados como resposta ou reação as informações capturadas pelo sistema. O par de estado antes e depois da ação ocorrer determina uma transição do sistema.

Os sistemas alvos do nosso trabalho serão os sistemas reativos autônomos, como por exemplo os softwares embarcados em aplicações espaciais, os robôs modernos e, até mesmo, as Redes de Sensores Sem Fio, que possuem tanto elementos de sensoriamento quanto elementos de atuação ou de comunicação com o mundo externo como forma de interagir e responder às suas missões de forma autônoma.

Existem diversas abordagens para a programação dos sistemas reativos. Da abordagem mais simples até as mais complexas, o usuário é forçado a escolher entre determinismo e concorrência. A programação mais simples pode ser feita usando as máquina de estado finito, os quais são usados para programar pequenos núcleos reativos, tipicamente em protocolos ou controladores. A interface é feita usando as facilidades do sistema operacional e a manipulação de dados é feita pela chamada de rotinas implementadas em linguagem convencional. O maior problema com essa abordagem é que tanto a implementação quanto manutenção estão sujeitas a erros introduzidos no processo. Além disso, pequenas alterações nas especificações podem gerar grandes alterações nos autômatos e estes, por serem puramente sequenciais, não suportam concorrência. Uma outra abordagem é utilizar as Redes de Petri, muito utilizadas na programação de controladores. Elas possuem primitivas de concorrência elementares, mas não suportam o desenvolvimento hierárquico. As linguagens de programação concorrentes, tais como ADA ou OCCAM são mais elaboradas para o desenvolvimento de sistemas reativos. Elas permitem o desenvolvimento de sistemas hierárquicos e modulares, possuem

mecanismos de tarefas e comunicação definidos na própria linguagem. Permitindo a programação das três camadas numa única linguagem.

Com a adoção da "hipótese do sincronismo", que consiste em assumir que toda reação do sistema é instantânea, e desta forma atômica, os problemas de determinismo e concorrência ficam mais simples de serem resolvidos [Berry & Gonthier, 1992]. De maneira prática a hipótese do sincronismo permite que o sistema reaja suficientemente rápido de modo a perceber todos os eventos externos na sua ordem correta e de modo que o ambiente externo possa ser considerado inalterado durante a reação.

A partir dessas idéias surgiram as linguagens síncronas, que são linguagens que adotam a hipótese do sincronismo, para permitir uma programação mais fácil com uma semântica mais clara para os sistemas reativos. As primeiras linguagens surgiram na França na década de 80 [Halbwachs, 1998]:

A seguir são apresentadas as características de algumas linguagens utilizadas na programação de sistemas reativos:

- Statecharts - é a primeira e mais popular linguagem formal projeta no início dos anos 80 para o projeto de sistemas reativos. Ela foi proposta mais como formalismo para especificação e projeto do que como linguagem de programação [Halbwachs, 1998]. Muitas características do modelo de sincronismo estão presentes em Statecharts, mas o determinismo não é garantido, e muitos problemas semânticos apareceram. Porém a partir dela muitas outras variantes surgiram e estão presentes em ferramentas comerciais.

- Lustre - é uma linguagem síncrona textual e declarativa, do tipo *data-flow*. Um programa em Lustre é um conjunto de equações que definem as variáveis de saída em função das variáveis de entrada.

- Signal - é uma linguagem do tipo *data-flow*. Ela difere de Lustre por ser uma linguagem relacional, onde a programação é realizada através da especificação de restrições ou relações sobre os sinais.

- Sterel - é uma linguagem textual, estruturada em blocos e imperativa. É apropriada para a especificação de tarefas dominadas por controle. É concorrente e determinística, também suporta preempção e exceções. Tem a sua semântica definida matematicamente. A linguagem é destinada apenas para a programação do núcleo reativo da aplicação. Precisa de suporte para as camadas de interface e manipulação de dados por parte da linguagem hospedeira.

2.2 X-machines

Uma boa prática para lidar com problemas complexos é a modelagem do problema por meio de modelos abstratos. Existem vários modelos de computação presentes na literatura (Maquinas de Turing, Cálculo Lambda, etc) As técnicas baseadas em métodos formais tornam as especificações mais precisas e desta maneira facilitam o armazenamento e processamento das informações, sendo que o problema inerente a sua utilização prática deriva da falta de conhecimento dos desenvolvedores na utilização do formalismo para construção dos produtos finais. *X-machine* é um formalismo baseado em maquinas de estados finitos (MEFs). A diferença está na transição de um estado para outro. Enquanto MEFs utilizam o consumo de um caractere de entrada para habilitar a mudança de estado, as *X-machines* utilizam uma função de transição que é aplicada num conjunto interno, definido como o tipo da máquina. As *X-machines* são consideradas um tipo de MEFs com o conjunto interno X atuando como uma memória, tendo ainda *streams* de entrada e saída. O modelo das *X-machines* foi desenvolvido pelo matemático Samuel Eilenberg em 1974 [Eilenberg, 1974]. Em 1986, Mike Holcome iniciou a utilização das *X-machines* em especificações no domínio da biologia e depois na especificação de sistemas [Holcombe, 1988]. No últimos anos as *X-machines* tem recebido modificações importantes para ampliar a sua capacidade de especificação de sistemas. Surgiram as *Stream X-machines* [Laycock, 1993] e em seguida as *Communicating Stream X-machines* [Barnard et al., 1996][Georgescu & Vertan, 2000] que são mais indicadas para modelar os sistemas concorrentes. Essa máquina é uma extensão a partir de uma máquina de estados finitos, com duas diferenças significantes:

- existe uma memória ligada a máquina,
- as transições são funções que operam sobre as entradas e os valores da memória.

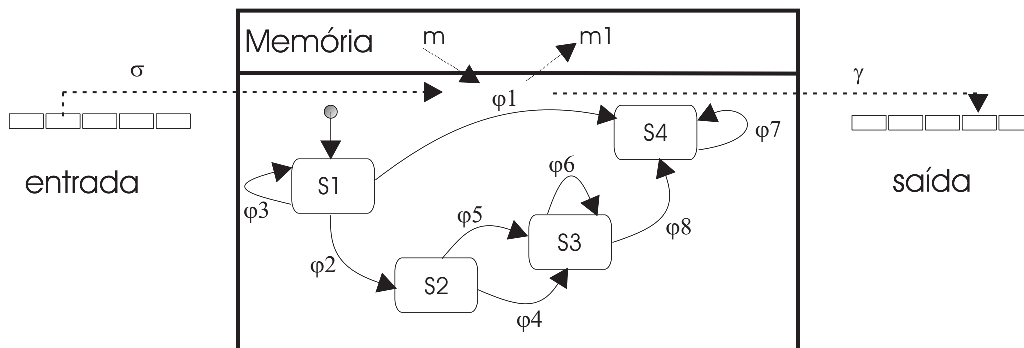


Figura 2.1: Esquema Geral da X-machine

Essas diferenças tornam tais máquinas mais expressivas e flexíveis que as máquina de estados finitos, além de empregarem uma abordagem diagramática de controle que estende o poder dessas máquinas, sendo capazes de modelar tanto os dados quanto o controle dos sistemas [Eleftherakis, 2003]. Os dados são mantidos em uma memória ligada a máquina e as transições são executadas pela aplicação de funções, escritas numa notação formal, modelando o processamento dos dados. As funções recebem as entradas e os dados da memória para produzir uma saída e uma atualização nos dados da memória, habilitando ainda uma mudança de estado da máquina. Uma outra extensão chamada *Stream X-machine* é definidas como uma *X-machine* onde a entrada e saída são feitas através de *stream* de dados [Laycock, 1993]. A Figura 2.1 apresenta um esquema geral de uma *Stream X-machine* onde podemos visualizar os elementos mais importantes. Dependendo do estado corrente e dos dados lidos da memória e do *stream* de entrada, essa entrada é consumida, um novo estado é definido e uma saída é produzida, para compor o *stream* de saída, juntamente com um novo elemento de memória. Em [Eleftherakis, 2003] é apresentada uma definição formal de uma *deterministic stream X-machine* como uma 8-upla $XM = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, onde:

- Σ, Γ são os alfabetos finitos de entrada e saída respectivamente.
- Q é o conjunto finito de estados.
- M é o conjunto (possivelmente) infinito chamado memória.
- Φ é um conjunto finito de funções parciais ϕ que mapeiam uma entrada e um valor de memória numa saída e num novo valor de memória, $\phi : \Sigma \times M \to \Gamma \times M$.
- F é a função parcial do próximo estado que, dado um estado e uma função do tipo Φ, denota o próximo estado. F normalmente é descrito como um diagrama de transição de estado, $F : Q \times \Phi \to Q$.
- q_0, m_0 são o estado e memória inicial.

2.3 Verificação de Modelos

A verificação de modelos ou *Model Checking* é uma técnica de verificação formal que consiste na representação de um sistema por meio de um modelo finito o qual será analisado para a determinação de sua conformidade com determinadas propriedades, e estas são expressas como fórmulas em lógica temporal [Edmund M. Clarke et al., 2000].

A lógica temporal é usada pela possibilidade da utilizacão dos operadores modais para expressar se uma propriedade é válida em todo o modelo ou em parte dele. O modelo pode ser representado no formato de um grafo de transição de estados, onde cada vértice representa um estado do sistema e as arestas representam as transições entre os estados. A garantia de que o processo de verificação de um modelo é finito baseia-se nos algoritmos, os quais operam sobre conjuntos de estados finitos. Uma outra característica da verificação de modelos é a geração de contra-exemplos para os casos em que o modelo não atende a suas especificações. Tradicionalmente, a aplicação desta técnica ocorre através de três etapas [Edmund M. Clarke et al., 2000]:

- Modelagem: consiste em construir um modelo formal do sistema, que deve ser aceito por uma ferramenta de verificação de modelos e, a partir deste modelo, obter todos os comportamentos possíveis do sistema. A estrutura que contém todos os comportamentos possíveis é conhecida como espaço de estados do sistema. Na maioria dos casos, esta fase é uma simples tarefa de compilação. Em outros casos, devido a limitações em tempo e memória, modelar um projeto requer mecanismos de abstração para eliminar os detalhes irrelevantes.

- Especificação: esta etapa consiste em especificar os comportamentos desejáveis do sistema. Um comportamento do sistema pode ser descrito formalmente através de lógicas temporais ou máquinas de estado.

- Verificação: esta etapa consiste em submeter o modelo e as especificações a uma ferramenta chamada verificador de modelos. Esta ferramenta produz como resultado um valor verdade que indica se a especificação é satisfeita ou não, no modelo. Em caso negativo, o verificador fornece uma sequência de estados alcançáveis, chamada de contra-exemplo, que demonstra que a especificação não é válida no modelo.

A verificação de modelos pode ser aplicada em sistemas reativos, que se caracterizam por uma interação contínua com o ambiente no qual estão inseridos. Sistemas desta natureza tipicamente recebem estímulos do ambiente e quase que imediatamente reagem às entradas recebidas. Tradicionalmente, eles são complexos, distribuídos, concorrentes e não possuem um término de execução, isto é, eles estão constantemente prontos para interagir com o usuário ou outros sistemas. Este conjunto de características exige que as propriedades destes sistemas sejam definidas não apenas em função de valores de entrada e saída, mas também em relação à ordem em que os eventos ocorrem. As lógicas temporais são utilizadas para a especificação de propriedades em verificação de modelos porque são capazes de expressar relações de ordem, sem recorrer à noção

explícita de tempo. Lógicas temporais são classificadas de acordo com a estrutura do modelo de tempo assumido, podendo ser linear ou ramificada. Tradicionalmente, duas formalizações de lógica temporal são utilizadas no contexto de verificação de modelos: LTL (Linear-time Temporal Logic) [Pnueli, 1977] e CTL (Computation Tree Logic) [Clarke & Emerson, 1982; Clarke et al., 1986]. A abordagem em LTL considera que uma propriedade pode ser quantificada para todas as execuções do sistema. A abordagem em CTL, por sua vez, considera que uma propriedade pode ser quantificada para uma ou para todas as execuções do sistema. O principal desafio à aplicação de verificação de modelos em situações reais é o problema conhecido como explosão do espaço de estados. Registrar todos os comportamentos possíveis de um sistema complexo pode esgotar os recursos de memória de uma máquina, mesmo que o número de estados alcançados pelo sistema seja finito. Por razões históricas a validade de uma fórmula em lógica temporal é analisada sobre um grafo rotulado de estado e transições, chamado estrutura Kripke [Edmund M. Clarke et al., 2000].

2.3.1 Lógica Temporal Ramificada

Clarke e Emerson [Clarke & Emerson, 1982] e paralelamente Quielle e Sifakis [Quielle & Sifakis, 1982] propuseram em meados dos anos 80 uma lógica capaz de considerar diferentes futuros possíveis, através da noção de tempo ramificado, conhecida como CTL (Computation Tree Logic). A idéia desta lógica é quantificar as possíveis execuções de um programa através da noção de caminhos que existem no espaço de estados do sistema. Assim as propriedades podem ser avaliadas em relação a alguma execução ou então em relação a todas as execuções. CTL é uma lógica proposicional temporal porque suas fórmulas são compostas de proposições atômicas, conectivos lógicos, quantificadores de caminhos e operadores temporais. Quantificadores de caminhos são usados para descrever a estrutura ramificada do tempo na árvore de computação, isto é, eles indicam quais caminhos a partir de um determinado ponto possuem propriedades relevantes. O quantificador universal, A, estabelece que todos os caminhos iniciando no estado definido possuem a propriedade considerada. Quantificadores de caminho devem ser imediatamente seguidos pelos operadores temporais que descrevem propriedades para o caminho especificado pelo quantificador. Operadores temporais básicos incluem X, F, G, e U. Seus significados são os seguintes:

- X: o operador temporal *next time* implica que no próximo estado do caminho a propriedade é satisfeita.

- F: o operador *eventual* ou *futuro* especifica que a propriedade será satisfeita eventualmente em algum estado no caminho.

- G: o operador *sempre* ou *global* especifica que a propriedade será satisfeita em todos os estados no caminho.

- U: o operador *until* que combina duas propriedades e especifica que a primeira propriedade é verdadeira até que a segunda propriedade seja verdadeira, não importando o valor da primeira após este ponto.

Quantificadores de caminho e operadores temporal são usados aos pares para compor os operadores CTL, da seguinte forma: AX, EX, AF, EF, AG e EG. Seja γ uma especificação CTL. De acordo com a definição, γ é uma proposição atômica ap ou é composta por operadores logico-temporais aplicada a sub-fórmula CTL da seguinte forma:

$$\gamma ::= ap \mid False \mid True \mid (\neg\gamma) \mid (\gamma \vee \gamma) \mid (\gamma \wedge \gamma) \mid (\gamma \rightarrow \gamma) \mid$$
$$AX\gamma \mid EX\gamma \mid AF\gamma \mid EF\gamma \mid AG\gamma \mid EG\gamma \mid A[\gamma_1\ U\ \gamma_2] \mid E[\gamma_1\ U\ \gamma_2]$$

$AX(\gamma)$ indica que γ é verdade em todo estado sucessor imediato do estado corrente do sistema. $AF(\gamma)$ especifica que γ eventualmente irá ocorrer no futuro em algum estado de todos os caminhos iniciando do estado corrente. $AG(\gamma)$ requer que γ seja verdade globalmente para todos os caminhos iniciando no estado corrente. $(A[\gamma_1\ U\ \gamma_2])$ significa que, para todo os caminho que se iniciam no estado atual, γ_2 deve valer em algum estado e γ_1 ocorrerá em todos os estado precedentes no caminho até que γ_2 ocorra. Quantificadores existenciais de caminho se comportam de maneira análoga, exceto que eles requerem que um único caminho satisfaça a propriedade. Fórmulas mais complexas podem ser expressas pelo agrupamento das fórmulas mais simples:

- $AG\neg(\gamma_1 \vee \gamma_2)$: γ_1 e γ_2 não ocorrem simultaneamente no sistema.

- $AG(\gamma_1 \rightarrow AF\gamma_2)$: se γ_1 é verdade, então γ_2 será verdade em qualquer lugar em todos os caminhos.

- $AG(\gamma_1 \rightarrow \gamma_2)$: se γ_1 é verdade, então γ_2 também é verdade.

Todos os operadores CTL podem ser expressos em termo de EX, EG a EU, os operadores temporais básicos:

- $AX\ \gamma = \neg EX(\neg\gamma)$

- $AF\ \gamma = \neg EG(\neg\gamma)$

- $EF\ \gamma = E[True\ U\ \gamma]$
- $AG\ \gamma = \neg EF(\neg\gamma)$
- $AF[\gamma_1\ U\ \gamma_2] = \neg E\ [\neg\gamma_2\ U\ (\neg\gamma_1 \vee \neg\gamma_2]\ \wedge\ \neg EG\ \neg\gamma_2$

A semântica da lógica CTL é definida com respeito a um árvore de computação, gerada a partir de uma estrutura Kripke, que é um modelo não determinístico baseado em estados. Cada estado na estrutura Kripke é definido por um conjunto de valores de variáveis que representam as proposições mantidas durante algum instante no sistema modelado. Uma estrutura Kripke M [Edmund M. Clarke et al., 2000] sobre um conjunto de propriedades atômicas AP é definido como uma tupla $M = (S, S_0, R, L)$ onde:

- S é um conjunto finito de estados.
- $S_0 \subseteq S$ é o conjunto de estados inicial.
- $R \subseteq S \times S$ é uma relação de transição que deve ser total, isto é, todo estado tem pelo menos um sucessor. Com isso não existe estado sem-saída.
- $L : S \rightarrow 2^{AP}$ é uma função que rotula cada estado com o conjunto de proposições atômicas verdadeiras no estado.

Um caminho iniciando no estado s_0 em M é uma sequência finita de estados $\pi = s_0 s_1 s_2 \ldots$ onde a relação $R(s_i, s_{i+1})$ é mantida para todo $i \geq 0$. Dada uma estrutura Kripke M, a semântica da lógica CTL é definida sobre a árvore de computação $M' = (S', S'_0, R', L')$, que é um grafo de estado infinito obtido de M, onde suas transições são geradas de acordo com as seguintes regras:

- S' consiste de todos os infinitos caminhos em M.
- $(\pi, \pi') \in R'$ se, e somente se, $\pi = s_0 s_1 s_2 \ldots s_n, \pi' = s_0 s_1 s_2 \ldots s_n s_{n+1}$ e $(s_n, s_{n+1}) \in R$
- S'_0 é formado por todos os caminhos em M com apenas um estado inicial.
- Para todo $\pi = s_0 s_1 s_2 \ldots s_n$ em M, $L'(\pi) = L(s_n)$.

Assim, o conjunto de estados de M' é isomórfico ao conjunto de caminhos infinitos em M. Portanto, uma estrutura de Kripke é uma máquina de estados finita que representa todas as possíveis execuções de um sistema. Todo estado do sistema

é rotulado com proposições atômicas que são verdadeiras no estado. Na definição de R, cada estado deve possuir ao menos um sucessor. Desta forma, situações reais nas quais um estado s não possui um sucessor devem ser representadas através de auto-laço $(s, s) \in R$. A estrutura deve ser gerada a partir da descrição do sistema analisado, no nosso caso ela será gerada a partir do modelo tabular, resultante do Modelo de Desenvolvimento *Bare*, utilizando uma *X-Machine* como formalismo gerador. Por exemplo seja $M_e = (S, S_0, R, L)$ uma estrutura Kripke definida sobre $AP = \{p_0, p_1, p_2\}$, um conjunto de proposições booleanas, tais que:

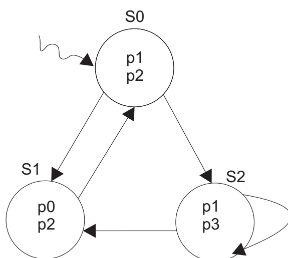


Figura 2.2: Estrutura Kripke de M_e

- $S = \{s_0, s_1, s_2\}$
- $S_0 = \{s_0\}$
- $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_2, s_1), (s_2, s_2)\}$
- $L = \{(s_0, \{p_1, p_2\}), (s_1, \{p_0, p_2\}), (s_2, \{p_1, p_3\})\}$

O grafo de transição de estados correspondente a M_e é apresentado na Figura 2.2. Após a aplicação das regras para converter a estrutura Kripke numa árvore de computação, teremos um grafo de transições infinitas M'_e, apresentado parcialmente na Figura 2.3.

A notação padrão indica que uma fórmula proposicional γ é verdadeira num estado s de uma árvore de computação M, ou seja $M, s \vDash \gamma$, leia-se M satisfaz γ. A relação $\vDash$ (satisfação) é definida indutivamente da seguinte forma:

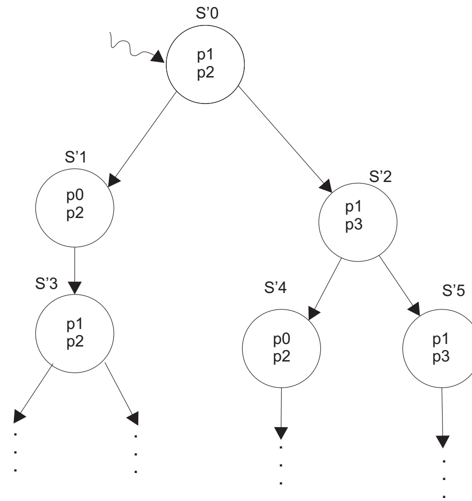


Figura 2.3: Árvore de Computação de M_e

$$
\begin{array}{lcl}
M, s_0 \models ap & \Leftrightarrow & ap \in L(s_0) \\
M, s_0 \models \neg\gamma & \Leftrightarrow & M, s_0 \not\models \gamma \\
M, s_0 \models \gamma_1 \vee \gamma_2 & \Leftrightarrow & M, s_0 \models \gamma_1 \ ou \ M, s_0 \models \gamma_2 \\
M, s_0 \models \gamma_1 \wedge \gamma_2 & \Leftrightarrow & M, s_0 \models \gamma_1 \ e \ M, s_0 \models \gamma_2 \\
M, s_0 \models \gamma_1 \rightarrow \gamma_2 & \Leftrightarrow & M, \ s_0 \not\models \gamma_1 \ ou \ M, s_0 \models \gamma_2 \\
M, s_0 \models AX \ \gamma & \Leftrightarrow & \forall \pi = s_0 s_1 ..., M, s_1 \models \gamma \\
M, s_0 \models EX \ \gamma & \Leftrightarrow & \exists \pi = s_0 s_1 ..., M, s_1 \models \gamma \\
M, s_0 \models AF \ \gamma & \Leftrightarrow & \forall \pi = s_0 ..., \exists j \geq 0, M, s_j \models \gamma \\
M, s_0 \models EF \ \gamma & \Leftrightarrow & \exists \pi = s_0 ..., \exists j \geq 0, M, s_j \models \gamma \\
M, s_0 \models AG \ \gamma & \Leftrightarrow & \forall \pi = s_0 ..., \forall i \geq 0, M, s_i \models \gamma \\
M, s_0 \models EG \ \gamma & \Leftrightarrow & \exists \pi = s_0 ..., \forall i \geq 0, M, s_i \models \gamma \\
M, s_0 \models A \ [\gamma_1 \ U \ \gamma_2] & \Leftrightarrow & \forall \pi = s_0 ..., \exists i [i \geq 0, M, s_i \models \gamma_2 \ e \ \forall j [0 \leq j < i \rightarrow M, s_j \models \gamma_1]] \\
M, s_0 \models E \ [\gamma_1 \ U \ \gamma_2] & \Leftrightarrow & \exists \pi = s_0 ..., \exists i [i \geq 0, M, s_i \models \gamma_2 \ e \ \forall j [0 \leq j < i \rightarrow M, s_j \models \gamma_1]]
\end{array}
$$

O uso dos operadores básicos CTL é mostrado na Figura 2.4, obtida considerando a árvore de computação mostrada na Figura 2.3. Os estados sombreados são aqueles que determinam a validade da fórmula em cada árvore de computação.

2.3.2 Lógica Temporal Linear

A lógica temporal linear (LTL) [Pnueli, 1977] assume o tempo como uma sequencia de execução de um sistema onde cada possível caminho de computação é considerado separadamente, raciocinando sobre uma única sequencia de execução. Ao invés de ser interpretado sobre árvores de computação, as fórmulas LTL são interpretadas com

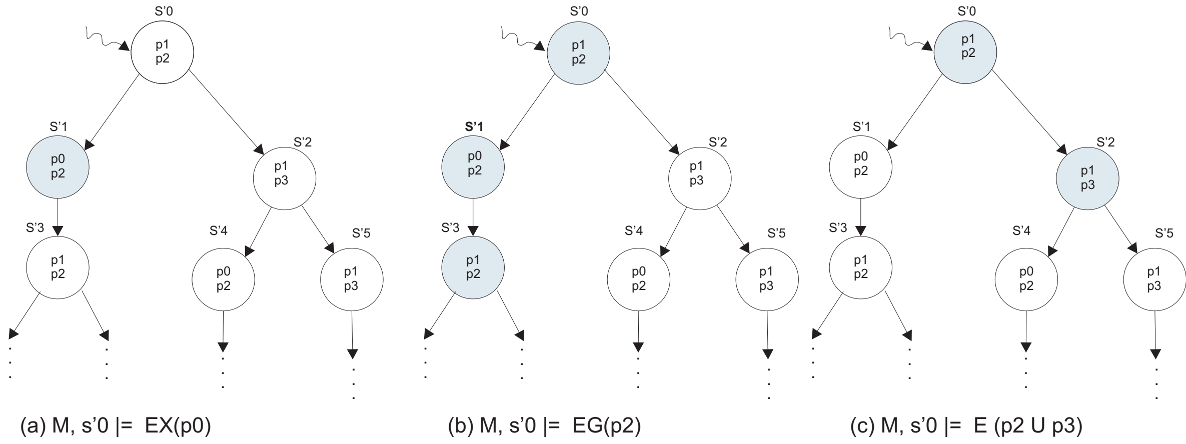


Figura 2.4: Representação Semântica da Lógica CTL

respeito a caminhos individuais de computação. Em outras palavras, a lógica temporal linear expressa propriedades sobre uma sequencia linear de execução do sistema.

As fórmulas em LTL são compostas de proposições atômicas usando os conectivos booleanos e os operadores temporais. Diferente de CTL, onde cada operador temporal deve ser prefixado com um quantificador de caminho, os conectivos proposicionais e os operadores temporais podem ser aninhados de maneira diferente em LTL. Uma fórmula LTL γ é definida recursivamente da seguinte forma:

$$\gamma ::= ap \mid False \mid True \mid (\neg\gamma) \mid (\gamma \vee \gamma) \mid (\gamma \wedge \gamma) \mid (\gamma \rightarrow \gamma) \mid (X\gamma) \mid (F\gamma) \mid (G\gamma) \mid (\gamma_1\ U\ \gamma_2)$$

onde ap é uma proposição atômica, X,F,G e U são operadores temporais previamente definidos para fórmulas CTL. É importante notar que as fórmulas LTL não possuem quantificadores de caminho explícito. Uma fórmula LTL é considerada verdadeira sobre todo o caminho computacional, isto é, as fórmulas LTL são implicitamente quantificadas universalmente no caminho. Cada fórmula LTL γ é considerada da forma $A(\gamma)$

A semântica das fórmulas LTL é definida com respeito ao caminho computacional de uma estrutura *Kripke*. Seja $M = (S, S_0, R, L)$ uma estrutura Kripke definida sobre o conjunto de fórmulas atômicas AP. Assumindo $\pi = s_0 s_1 s_2 ...$ um caminho computacional onde $R(s_i s_{i+1})$ mantem para todo $i \geq 0$ e $\pi^i = s_i s_{i+1} s_{i+2} ...$ é o sufixo de π iniciando em s_i. A relação de satisfação $M, \pi \models \gamma$ para o caminho computacional π e uma fórmula LTL γ é definida indutivamente da seguinte forma:

$$
\begin{array}{lcl}
M,\pi \models ap & \Leftrightarrow & ap \in L(s) \text{ onde } s \text{ é o primeiro estado de } \pi \\
M,\pi \models \neg\gamma & \Leftrightarrow & M,\pi \not\models \gamma \\
M,\pi \models \gamma_1 \vee \gamma_2 & \Leftrightarrow & M,\pi \models \gamma_1 \; or \; M,\pi \models \gamma_2 \\
M,\pi \models \gamma_1 \wedge \gamma_2 & \Leftrightarrow & M,\pi \models \gamma_1 \; and \; M,\pi \models \gamma_2 \\
M,\pi \models \gamma_1 \rightarrow \gamma_2 & \Leftrightarrow & if \; M,\pi \models \gamma_1 \; then \; M,\pi \models \gamma_2 \\
M,\pi \models X\,\gamma & \Leftrightarrow & \pi^1 \models \gamma \\
M,\pi \models F\,\gamma & \Leftrightarrow & \exists\, i \geq 0 \; \pi^i \models \gamma \\
M,\pi \models G\,\gamma & \Leftrightarrow & \forall\, i \geq 0 \; \pi^i \models \gamma \\
M,\pi \models \gamma_1 \, U \, \gamma_2 & \Leftrightarrow & \exists\, i[i \geq 0, \pi^i \models \gamma_2 \; and \; \forall\, j[0 \leq j < i, \pi^j \models \gamma_1]]
\end{array}
$$

Algumas especificações práticas, onde as variáveis proposicionais correspondem a estados de uma sistema reativo real:

- Não é possível chegar a um estado em que “started” se verifica e “ready” não se verifica: $G \sim (started \wedge \; \sim ready)$
- Para qualquer estado, se “request” se verifica, então no futuro também se irá verificar “ack” : $G(request \rightarrow F\ ack)$
- Se um processo é “ativado” (enabled) um número infinito de vezes, então ele executa (run) um número infinito de vezes: $GF\ enabled \rightarrow GF\ run$
- Para todos os estados, existe uma caminho para um estado que satisfaz “restart”: Não se pode exprimir em LTL.

Em LTL pode expressar alcançabilidade e segurança, porém LTL não pode expressar a existência de um caminho, ou seja não pode expressar propriedades que dizem respeito a um único caminho. Com isso LTL é implicitamente quantificada universalmente.

2.4 NuSMV

A ferramenta *New Simbolic Model Verifier - NuSMV* [Cimatti et al., 1999] é uma ferramenta de código aberto que é usada para verificação de modelos simbólicos. Ela foi originada a partir da ferramenta SMV, que é o verificador de modelos baseado em BDD desenvolvido na Carnegie Mellon University [McMillan, 1993]. NuSMV provê uma linguagem para descrição do modelo e verifica diretamente a validade das fórmulas em LTL e também em CTL. Ela recebe como entrada um texto que consiste de um programa descrevendo o modelo e algumas especificações descritas como fórmulas em lógica temporal. Ela produz como resultado “true” se a especificação é satisfeita ou

um contra-exemplo mostrando porque a especificação representada pelo programa não é satisfeita, ou seja, é "false".

NuSMV foi projetado para ser robusto e atender aos padrões requeridos pela indústria, ser fácil de manter e modificar, foi escrito em ANSI C e seu código fonte é dividido em vários módulos. A linguagem de entrada de NuSMV permite a descrição de Máquinas de Estado Finitas (MEFs), é capaz de descrever processos síncronos e assíncronos bem como condições de não determinismo. O propósito inicial da entrada de NuSMV é descrever as relações de transição das MEFs, onde as relações descrevem as evoluções válidas do estado das MEFs e, consequentemente, do sistema sendo modelado, dessa forma, podem ser identificadas as possíveis configurações futuras de um sistema a partir do seu estado atual. De um modo geral, qualquer expressão no cálculo proposicional pode ser usada para definir as relações de transição, o que dá uma grande flexibilidade e ao mesmo tempo requer um certo cuidado adicional para evitar inconsistências, como por exemplo a presença de uma contradição lógica, que pode resultar em um *deadlock*. É descrito a seguir os elementos da especificação de um sistema utilizando a linguagem de entrada para a ferramenta NuSMV:

- MODULE - Encapsula as declarações de todas as variáveis, inclusive as de estado e os eventos, que são declarados BOOLEAN. Cada módulo pode conter a regra de transição dos estados e a especificações das propriedades. É possível declarar um módulo com parâmetros para realizar a reutilização de código, e a passagem é por referência.

- VAR - As declarações das variáveis são realizadas no bloco VAR. Podem ser do tipo enumerado, intervalar, booleano e instâncias de outros módulos.

- ASSIGN - Nesta seção são declaradas as regras que determinam a inicialização e transições para o próximo valor de uma variável. A atribuição direta estabelece o valor inicial da variável, por meio da comando *init(variável)*. O comando *next(variável)* fornece o valor da variável no próximo seguinte, a partir do estado corrente. A atribuição em *next(variable)* pode ser feita utilizando a regra *case*, onde é possível determinar o próximo valor da variável em função de várias condições.

- DEFINE - A seção DEFINE é utilizada para associar uma variável a uma expressão. Uma variável em DEFINE é sempre substituída pela sua definição quando é encontrada na especificação.

- MODULE main (módulo principal) - Toda especificação NuSMV deve possuir um módulo principal, sem parâmetros, que representar o sistema.

- CTLSPEC - As propriedades CTLs são especificadas precedidas da palavra chave CTLSPEC (SPEC anteriormente). Se as propriedades forem fórmulas LTL, deve-se utilizar LTLSPEC.

```
MODULE main
VAR
    request : boolean;
    state : {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) :=
        case
            state = ready & request: busy;
            1 : {ready, busy};
        esac;
LTLSPEC
    G(request -> F state=busy)
```

Tabela 2.1: Modelo do Sistema na Linguagem NuSMV

A Figura 2.1, ilustra o modelo do sistema descrito ou programado na linguagem NuSMV. Um programa em NuSMV pode ter um ou vários módulos e, tal como algumas linguagens de programação, um dos módulos deve ser chamado "main". Nos módulos são declaradas as variáveis e seus respectivos valores. As atribuições normalmente são feitas com um valor inicial para cada variável e em seguida uma especificação do próximo valor por meio de uma expressão formada pelos valores correntes das variáveis. Essa expressão pode ser não-determinística. O programa da Figura 2.1 consiste de duas variáveis *request* do tipo booleana e *state* do tipo enumerado {*ready,busy*}, onde 0 denota "false" e 1 representa "true". Os valores iniciais e subsequentes da variável *request* não são definidos no programa e, desta forma, os valores de *request* serão definidos pelo ambiente externo durante a execução do programa. Com isso a variável *state* fica parcialmente definida, inicialmente ela é "ready" e pode ficar "busy" se a variável *request* for "true". Porém se a variável *request* for "false" o valor de *state* fica indeterminado. A avaliação da expressão formada com o "case" é feita de cima para baixo, sendo que a primeira avaliação do lado esquerdo do sinal ":" (dois pontos) que tiver seu valor verdadeiro, atribuirá o valor do lado direito para a variável declarada. Desta forma o valor "1:" fica como a avaliação padrão, caso nenhuma das avaliações anteriores seja verdadeira.

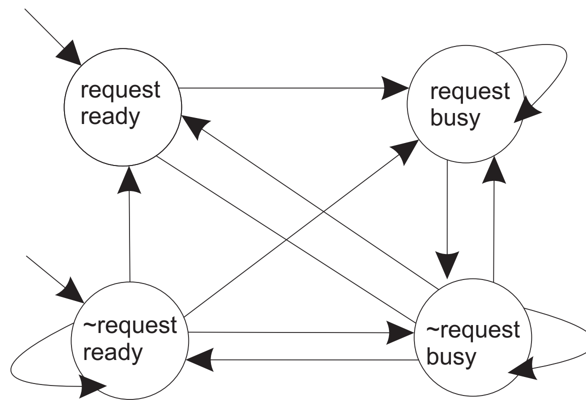


Figura 2.5: Máquina de Estados do Sistema

O programa denota o sistema de transição da Figura 2.5, onde existem 4 estados, e cada estado é definido em função dos valores possíveis para as duas variáveis. No sistema o nome "∼request" significa um valor falso para a variável "request". O programa e o sistema de transição são não-determinísticos, isto é, o próximo estado não é unicamente definido. Qualquer transição de estado baseado no comportamento da variável "state" vem em pares e o resultado é a transição para um estado sucessor onde "request" pode ser falso ou verdadeiro, respectivamente. Na Figura 2.5 pode-se notar que a partir do estado {∼request, busy} o sistema pode caminhar para quatro estados destinos, ele mesmo e mais três outros estado. As especificações em LTL são introduzidas pela palavra chave *LTLSPEC* e são simples fórmulas LTL. No programa exemplo a especificação está declarando que: Para qualquer estado, se o valor de "request" é verdadeiro, então eventualmente ocorrerá um estado "busy".

2.5 Reconfiguração de Aplicação

Quando o assunto é *Reconfiguração* a idéia que surge inicialmente é o conceito de *Reconfiguração de Hardware* usando *FPGA* (Field-Programmable Gate Array), isso ocorre em função da extensa pesquisa na área de *Reconfiguração de Hardware* [Compton & Hauck, 2002] e das possibilidades de novas aplicações que despertam a partir dessa nova tecnologia. Na área de RSSF foi proposto a utilização de um hardware reconfigurável para nó sensor, chamado RANS-300 , baseado num microcontrolador MSP430 e numa FPGA de 300k gates da família Spartan II-E [Caldas et al., 2005a]. O principal objetivo dessa arquitetura é incorporar flexibilidade, adaptabilidade e autonomia ao nó sensor, usando como paradigma de desenvolvimento o conceito de cenários de aplicação [Caldas et al., 2005b]. O conceito de cenários permite que ocorra uma evolução adapta-

tiva na execução das aplicações que executam nos nós sensores. As aplicações iniciam o seu funcionamento num cenário de instalação ou inicialização, terminada esta fase elas alteram a sua configuração para um outro cenário, que seria o cenário de trabalho normal. Assim existem novas possibilidades para a aplicação no nó sensor, ela poderá continuar no cenário de trabalho normal e concluir sua execução ou poderá alterar novamente a sua configuração (reconfiguração) para se adaptar a uma situação fora dos padrões normais, que poderiam exigir uma maior capacidade de processamento para o nó sensor, neste caso a *FPGA* seria configurada para fornecer hardware extra para o nó sensor. Todas essas possibilidade foram idealizadas com o RANS-300, mas durante a fase de implementação alguns problemas surgiram para tornar o RANS-300 um nó sensor estável para os experimentos. Havia a necessidade da construção de todos os *drivers* juntamente com o sistema operacional adaptado para o novo hardware, mais a parte da comunicação de dados. A solução adotada em face deste problema foi construir uma solução para o desenvolvimento de aplicações em RSSF que utilizasse os conceitos de reconfiguração de hardware, mas que não dependesse diretamente do hardware. O modelo de desenvolvimento *Bare* permite que o conceito de aplicações baseadas em cenário seja utilizado no desenvolvimento de aplicações e para isso empresta os conceitos de reconfiguração e os utiliza para implementar o conceito de reconfiguração das aplicações.

O termo cenário tem muitas interpretações nas mais diversas área da computação. como por exemplo:

- "Um cenário é um conjunto ordenado de interação entre parceiros, normalmente entre um sistema e um conjunto de atores externos ao sistema. Pode consistir numa sequência concreta de passos de interação ou num possível conjunto de passos de interação." [Ryser & Glinz, 1999; Glinz, 2000]

- "Um cenário é uma descrição que contém atores, a informação por trás deles, asserções sobre o seu ambiente, os seus objetivos e sequências de ações e eventos. Em alguns sistemas, os cenários podem omitir um dos elementos ou expressá-lo de forma simples ou implícita. " [Rosson & Carroll, 2001]

No contexto deste trabalho um cenário é um conjunto de estados e suas respectivas reações programadas para responder aos eventos detectados. A modelagem das aplicações baseadas em cenários tem como princípio que uma aplicação pode evoluir no tempo para se adaptar a novas situações, previstas ou não. Desta forma o cenário descreve uma aplicação com a perspectiva do usuário, onde cada cenário é formado

por um conjunto de estados que se relacionam por meio de funções com o objetivo de realizar uma tarefa específica.

Em [Skliarova, 2004] são descritos os seguintes modos de reconfiguração: a reconfiguração pode ser *Estática*, quando a configuração é executada uma única vez na instalação ou *Dinâmica*, quando a configuração pode ocorrer periodicamente para alterar o comportamento do sistema. Os cenários da aplicação pode evoluir durante o

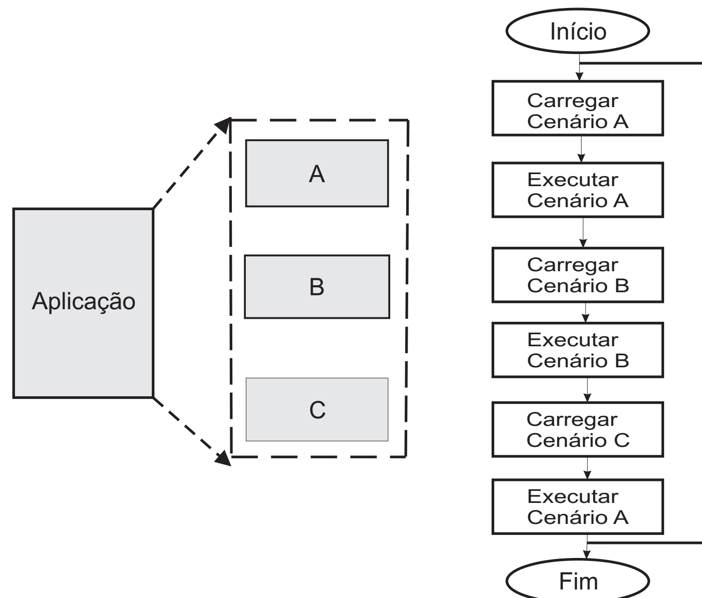


Figura 2.6: Reconfiguração Dinâmica Baseada em Cenários

tempo de execução, utilizando a reconfiguração dinâmica. Desta forma a execução da aplicação inicia com um cenário e este será substituído por outros cenários da aplicação a medida que as condições habilitam a troca. Esse mecanismo pode ser melhor visualizado por meio da Figura 2.6.

A reconfiguração *Dinâmica* [Skliarova, 2004] pode ainda ser dividida em *Contexto Único*, o qual requer uma reconfiguração completa mesmo quando for substituído um simples item da aplicação, e *Contexto Múltiplo*, quando a aplicação possui várias camadas de execução, mas somente uma das camadas está ativa por vez, esse modelo permite a reconfiguração em "background". *Contexto Parcialmente Reconfigurável*, onde pequenas partes da camada ativa podem ser alterados sem a necessidade de reprogramar todo o dispositivo. Este modelo é interessante porque pode ser feito em tempo menor que os anteriores e, caso a zona alterada não estiver em uso no momento, a reconfiguração poderá ser feita em paralelo com a execução da aplicação. Os contextos de reconfiguração são mostrados na Figura 2.7.

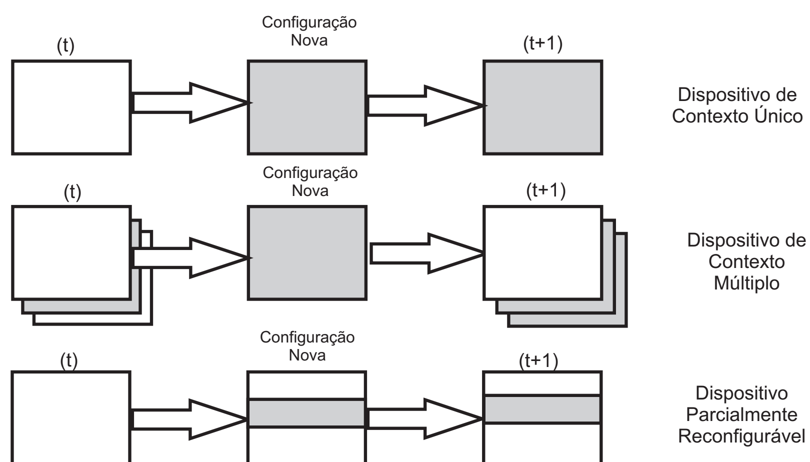


Figura 2.7: Contextos de Reconfiguração Dinâmica

Capítulo 3

Modelo de Desenvolvimento *Bare*

Neste capítulo é descrito o processo geral do Modelo *Bare* para o desenvolvimento das aplicações em Sistemas Reativos Autônomos usando as *X-machines*.

Neste trabalho um sistema reativo é considerado como uma unidade programável que está situado em algum ambiente e que deve desempenhar um papel autônomo. A programação pode ser adaptada como forma de atender às situações não previstas. Nesse contexto, o sistema deverá desempenhar um comportamento reativo e, desta forma, interagir com o meio ambiente. O sistema responderá às alterações ambientais por meio de mudanças em alguns de seus parâmetros internos o que deverá refletir em uma mudança no seu estado corrente, indicando que houve a captação da ocorrência do evento externo e responder ao estímulo com uma informação indicativa para o próprio ambiente. Essa informação poderá ser apenas uma mensagem ou a ativação de um novo evento. Para modelar o comportamento reativo em um ambiente dinâmico são destacados os seguintes elementos envolvidos no processo:

- o conjunto de estímulo ambiental ou entradas;
- o conjunto de estado internos ao sistema;
- o conjunto de regras que relacionam os estímulos com as mudanças de estados internos;
- o conjunto de respostas ou eventos gerados.

A utilização de máquinas de estados finitos (MEFs) é uma abordagem comum nas mais diversas áreas no desenvolvimento de aplicações para os sistemas computacionais. MEFs são largamente utilizadas no desenvolvimento de tecnologias chaves tais como projeto de hardware, compiladores e projeto de sistemas de tempo real. A utilização

das *X-machines* como um modelo para o desenvolvimento de aplicações em sistemas reativos autônomos é determinada, principalmente, pelo fato desse modelo ser capaz de modelar tanto os dados quanto o controle do sistema a ser desenvolvido [Eleftherakis, 2003].

Os dados são mantidos numa memória ligada a máquina e as transições entre os estados são realizadas através da aplicação de funções, as quais são descritas por meio de uma notação formal e modelam o processamentos dos dados. As funções recebem como entrada os dados e os valores armazenados na memória e produzem como saída os valores processados e, possivelmente, novos valores para a memória. Uma *X-machine* sozinha tem uma capacidade limitada para a modelagem das aplicações reais ou mais complexas, porém essa capacidade pode ser aumentada com a adição de novas características, tais como hierarquia ou comunicação ao modelo.

Sendo esse formalismo flexível e adaptável, além de simples e intuitivo, argumenta-se que ele pode ser utilizado para modelar o comportamento e a interação entre os sistemas reativos autônomos. Sendo o sistema uma entidade autônoma pode-se usar uma máquina de estados para modelar o seu comportamento na execução de uma aplicação. Com esse modelo pode-se especificar ou verificar as sequências de estados pelos quais passará o sistema em resposta aos eventos externos, bem como as respectivas respostas produzidas como saídas. De maneira análoga uma rede de sistemas autônomos será como um conjunto de sistemas formado por uma rede de máquinas que se comunicam e cooperam na execução da tarefa.

Devido a necessidade de um hardware alvo real para os experimentos optou-se pela utilização do Robô NXT da Lego pela sua capacidade de interação com o ambiente por meio de sensores e atuadores, além da sua simplicidade de programação. O foco principal desta pesquisa é a definição de um modelo geral para o desenvolvimento de aplicações a partir do desenho da aplicação num formato abstrato, usando *X-machine*, em seguida a transformação deste formato para uma especificação padrão que será executada pelo ambiente de execução, rodando no robô NXT [Lego NXT, 2009]. Viabilizando, assim, a programação ou reprogramação das aplicações de sistemas reativos autônomos por meio do envio da especificação completa ou apenas trechos com a alteração necessária. Neste processo são utilizados os conceitos de reconfiguração dinâmica de contexto parcialmente reconfigurável, conforme Seção 2.5. Usando esse formalismo acrescenta-se ao desenvolvimento de aplicações para sistemas reativos autônomos uma metodologia formal tanto na fase de desenvolvimento quanto na fase de verificação e testes, por meio da geração automática da aplicação no formato tabular, bem como da geração automática do modelos do sistema para verificação de modelos (NuSMV).

3.1 Componentes do Modelo

O Modelo *Bare* parte do princípio que, em geral, as aplicações em sistemas reativos autônomos são formadas basicamente por 3 (três) partes fundamentais:

- O sensoriamento dos eventos ou captação dos dados,
- O processamento dos dados e
- A atuação ou transmissão dos dados que ocasiona o disparo de novos eventos.

Além dos elementos já descritos, existem os elementos que compõem a interface com *hardware* do sistema. Considera-se que todos os elementos que fazem parte do sistema terão o seu modelo funcional projetado por meio de uma *X-machine*, como mostra a Figura 3.1.

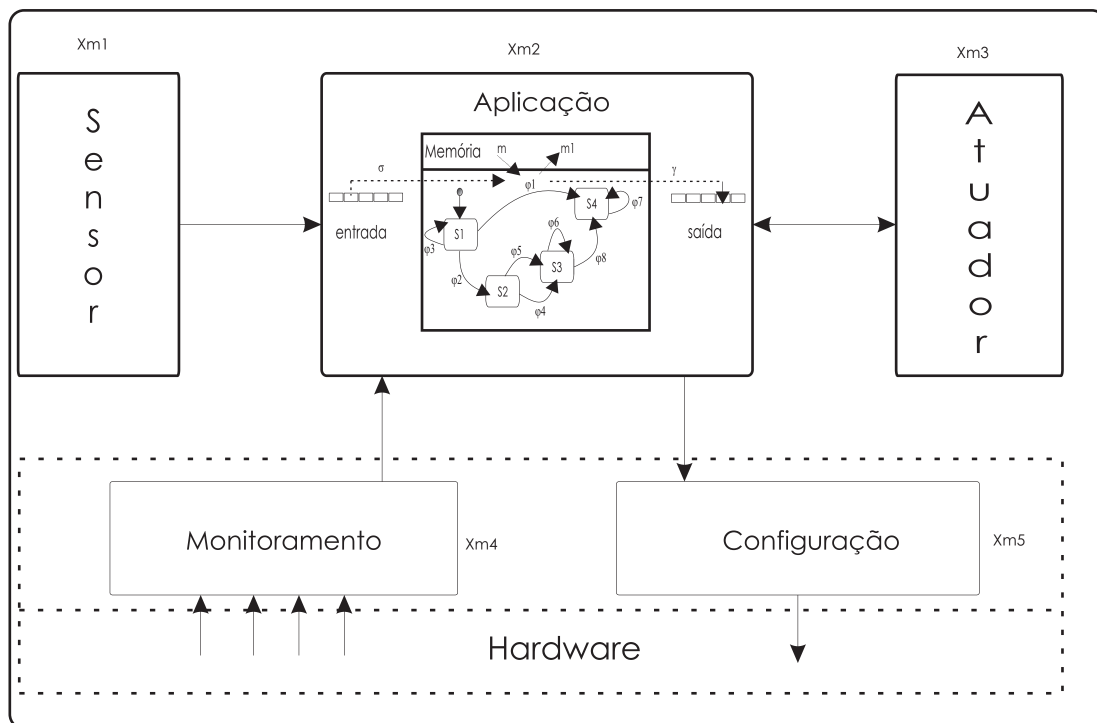


Figura 3.1: Esquema Geral de um Sistema Reativo Autônomo

Em seguida é descrito o papel de cada componente na construção do Modelo *Bare*:

- O sensor: é o responsável pela geração dos dados para a aplicação. A geração poderá ser contínua, iniciada por eventos ou programada.

- A aplicação: é a peça principal da metodologia. É por meio desse componente que o desenvolvedor poderá criar ou adaptar o processamento efetivo para atender aos requisitos da aplicação.

- O Atuador: é o responsável pela reação do sistema ou pelo envio dos dados para os demais elementos que formam a rede. A implementação do modelo de comunicação da rede ficará escondida da aplicação e este modelo poderá ser adaptado para atender os requisitos da aplicação.

O objetivo principal do Modelo *Bare* é ser capaz de capturar o esquema geral das aplicações em sistemas reativos autônomos, tornando a sua construção, distribuição e, quando for o caso, a reprogramação dos sistemas reativos autônomos mais formal.

As aplicações dos sistemas reativos em geral trabalham a partir dos dados captados pelo sensor, executam alguma manipulação correspondente e o resultado pode ser na forma de atuação direta pelo hardware ou por meio do envio de informações para outros elementos. Porém existem algumas aplicações onde é necessário uma característica de adaptação ou adequação no hardware do próprio sistema para responder de maneira satisfatória a situação detectada. Como por exemplo, a necessidade da aplicação alterar a velocidade do próprio robô para agilizar a sua resposta ou até mesmo ampliar a transmissão/recepção do mecanismo de comunicação. Para atender a esse tipo de aplicação o Modelo *Bare* permite que ocorra uma interação entre a aplicação e os elementos de hardware do sistema por meio um componentes de monitoramento e um componente de configuração, mostrados na Figura 3.1. O módulo de **monitoramento** é responsável pelas informações sobre a situação dos elementos de hardware, mantendo o sistema informado sobre os elementos críticos tais como: nível da bateria, memória disponível, etc. O módulo de **configuração** é responsável pela intervenção da aplicação diretamente no hardware do sistema, para executar tarefas como: controle da taxa de amostragem dos elementos sensores, configuração dos elementos de hardware para permitir uma reconfiguração completa do sistema, mudança na sensibilidade do sensores ou atuadores, etc. Desta forma a gestão do hardware do sistema será feita através de um nível mais alto de abstração, deixando o programador das aplicações livre para realizar as configurações necessárias de maneira segura e correta.

Depois de definido a interface entre a aplicação, os módulos de monitoramento e a configuração do hardware do sistema, o foco concentra-se no desenvolvimento das aplicações. O objetivo é construir um modelo de desenvolvimento que seja mais simples, por isso chamado de "Modelo de Desenvolvimento *Bare*[1]", o qual é composto da análise dos requisitos da aplicação, seguido do desenho lógico da aplicação e depois a sua

[1] *Bare* pode ser entendido como *Básico* ou relativo aos Índios *Barés* do Amazonas.

implementação. Após a construção do desenho lógico da aplicação existem duas possibilidades para a execução das aplicações. A Figura 3.2 mostra o encadeamento das fases, desde a concepção até a geração das aplicações e execução no hardware. A primeira possibilidade é a geração do código executável para ser depositado diretamente no hardware. Uma segunda abordagem é a geração de um código usando um padrão de alto nível que será interpretado por um interpretador que será previamente depositado no hardware, sendo está a opção adotada no trabalho.

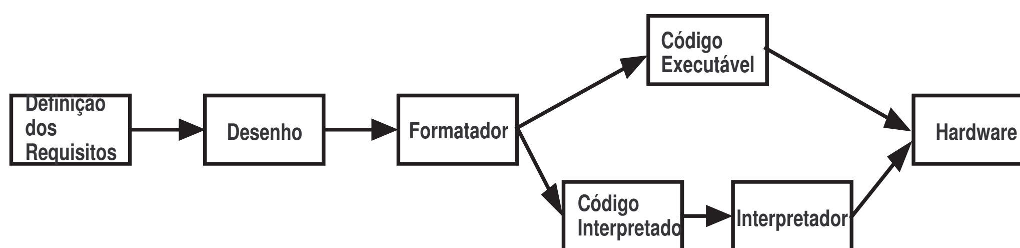


Figura 3.2: Fases do Modelo de Desenvolvimento

Existe na literatura várias propostas de aplicações para as Redes de Sensores Sem Fio - RSSF [Ruiz, 2003] e novas aplicações são idealizadas nas mais diversas áreas. A diversidade das aplicações é tanta que já existe a necessidade de criar classificações das aplicações para um melhor estudo e comparação [Romer & Mattern, 2004][Arampatzis et al., 2005]. Com o crescimento de novas aplicações em RSSF, surge a necessidade de melhores ferramentas para auxiliar na especificação e desenvolvimento dessas aplicações. Com isso, um novo tema de pesquisas fica evidente na área de RSSF que é o de simplificar a tarefa da construção das aplicações. O tema de pesquisa é de extrema relevância pois, para alguns pesquisadores da área, a construção das aplicações para RSSF é uma tarefa cheia de dificuldades [Mottola & Picco, 2008; Newton & Welsh, 2004; Abdelzaher et al., 2004; Welsh & Mainland, 2004]. Tais dificuldade são tantas que tem criado uma barreira para a disseminação apropriada da tecnologia de RSSF [Mottola & Picco, 2008; Glaser, 2004].

Como uma contribuição para facilitar o desenvolvimento de aplicações em RSSF este trabalho propõem uma abordagem diferenciada para o desenvolvimento de aplicações em RSSF. Considera-se o nó sensor como um Sistema Reativo Autônomo, que recebe estímulos do meio ambiente por meio de sensores e reage a esses estímulos por meio do envio de mensagens para os outros nós sensores que formam uma RSSF.

Programar as aplicações e depositá-las (*upload*) nos nós sensores tem demonstrado ser um campo de extremas dificuldades devido às características e restrições impostas pela tecnologia vigente. O elevado número de nós que compõem a rede torna

a tarefa de programação individual do nós sensores e, quando necessário a sua reprogramação, um desafio ainda não resolvido completamente dentro da área de RSSF. Por ser um campo extremamente novo e promissor é proposto o Modelo de Desenvolvimento *Bare* para a construção de aplicações em RSSF.

3.2 Modelagem da Aplicação

A proposta de desenvolvimento de aplicações para RSSF usando o Modelo *Bare* é apresentada por meio de um exemplo de aplicação. A aplicação escolhida é bem simples e acompanha a distribuição do TinyOs, é chamada "Blink". A Figura 3.3 mostra como a aplicação é modelada usando o modelo *Bare*.

Inicialmente são acrescentados 3 (três) novos estados Init(I), Start(S) e Halt(H) que estarão incluídos no modelo como parte de todas as aplicações geradas. Os novos estados tem as seguintes finalidades:

- Init - É o estado onde são feitas todas as inicializações necessária para a aplicação. São definidos os valores das variáveis bem como os tipos do sensoriamento executado e o padrão da comunicação (roteamento). Este estado só será executado uma única vez.
- Start - É o estado onde a aplicação está pronta para executar, aguardando algum evento ou o tempo programado para iniciar (eventos periódico). Este estado poderá ser executado várias vezes.
- Halt - É o estado que a aplicação executou a sua tarefa, e ficará adormecida até que um novo evento a desperte.

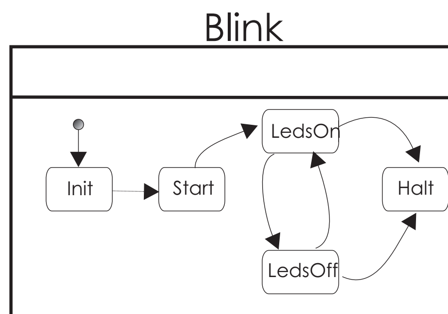


Figura 3.3: Aplicação Blink

O primeiro passo do Modelo *Bare* é a identificação dos elementos de interação entre a aplicação e o meio ambiente. Esse elementos são definidos por meios das

seguintes variáveis, que representam a interação da aplicação com o ambiente externo e o comportamento interno:

- **monitoradas**, são as variáveis cujos valores serão capturados pela aplicação e influenciam no seu comportamento. Sempre que uma variável tiver seu valor alterado um **evento** será disparado, elas formam o conjunto (Σ).

- **controladas**, são as variáveis por meio das quais a aplicação atuará no meio ambiente, repassando dados ou disparando eventos, elas formam o conjunto (Γ).

- **internas**, são variáveis internas a aplicação, usadas para representar grandezas auxiliares que são reutilizadas (Λ).

Para facilitar a identificação do papel das variáveis pelo nome, é convencionado iniciar as variáveis monitoradas com "m", as variáveis controladas com "c" e as internas com "i". No caso da aplicação *Blink* a variável monitorada é o $mTempo$ (um sinal emitido a cada intervalo de tempo) e as variáveis controladas são os comandos *cLedOn* e *cLedOff*, para ligar ou desligar um *led*, neste exemplo este eventos serão substituídos por mensagens que representam os respectivos comandos. As seguintes variáveis internas são utilizadas: $iCorr$ para armazenar a contagem de tempo e a variável $iTotal$ para armazenar o limite do tempo de execução da aplicação. No contexto do Modelo *Bare*, a definição dos nomes das variáveis ajudam a definir a qual conjunto pertencem e a sua função na aplicação. A definição do conjunto de valores que as variáveis podem manipular é importante, assim o próximo passo é definir o domínio (tipo) de cada variável. Por simplicidade construção do domínio das variáveis está restrito a dois tipo base: *inteiros* e *enumerados.*

- O tipo base inteiro é definido por meio do limite de intervalos onde a variável pode receber qualquer valor dentro deste intervalo:[$Val_{Inicial}$, Val_{Final}], por exemplo: Tempo=[1, 1000], Temperatura=[-10, 40]

- O tipo base enumerado é definido pela enumeração explícita de todos os valores que a variável pode assumir: { Val_1, Val_2 , ... , Val_n }, por exemplo: Booleano={ true, false }, Switch={ On, Off }.

Os tipos novos são definidos em função do contexto da aplicação a ser desenvolvida, sendo que alguns tipos padrão são necessários sempre, como é o caso do tipo *Booleano*. A Tabela 3.1 mostra como pode ser feita essa definição:

Tipo	**Tipo Base**	**Faixa Valores**	**Unidade**
Booleano	Enumerado	{true, false}	N/A
Tempo	Inteiro	[1, 1000]	Segundos
Switch	Enumerado	{On, Off}	N/A

Tabela 3.1: Definição de Tipos

Com a definição dos tipos das variáveis, tem-se a descrição do conjuntos de tipo T:

$$T = (Booleano = \{true, false\}, Tempo = [1, 1000], Switch = \{On, Off\})$$

Depois da identificação dos elementos de interface de entrada e saída com seus respectivos domínios, é a hora de identificar os estados da aplicação com as respectivas transições entre os estados para completar o modelo da aplicação.

A aplicação *Blink* tem a finalidade de ligar e desligar um *led*, sempre que receber um sinal de entrada indicando a passagem do tempo (mTempo) até que um limite de tempo seja alcançado (tTotal). Os estados identificados são:

- Init - estado de inicialização e configuração.
- Start - estado de pronto para executar.
- LedOn - é o estado onde o led é ligado.
- LedOff - é o estado onde o led é apagado.
- Halt - estado indicando final de execução.

Até este ponto foi feito um levantamento do elementos de interação da aplicação com o ambiente externo, por meio da identificação das variáveis monitoradas, controladas e internas, com seu respectivos conjuntos de valores e a definição dos estados para a aplicação. Essa parte tem um caráter mais estático para a modelagem, porque não envolve o momento em que as transições ocorrem e nem porque elas ocorrem dentro da aplicação, ou seja, não considera os aspectos dinâmicos da aplicação. Até este ponto foram acrescidos dois novos componentes para a *X-machine*, o componente de tipo T e o componente das variáveis internas Λ, sendo que os estados Q são apenas acrescidos de 3 novos estados. Uma nova definição para a *X-machine* que representa a aplicação é apresentada a seguir, com os novos componentes destacados com * :

$$M = (T^*, \Sigma, \Gamma, \Lambda^*, Q^*, M, \Phi, F, q_0, m_0)$$

onde:

- $\Sigma = (mTempo\ : Tempo)$ o alfabeto de entrada com o tipo definido.
- $\Gamma = (cLed\ : Switch)$ o alfabeto de saída com o tipo definido.
- $\Lambda = (iCorr\ : Tempo,\ iTotal\ : Tempo)$ o alfabeto temporário com o tipo definido.
- $Q \cup \{I, S, H\} = \{LedOn,\ LedOff,\ Init,\ Start,\ Halt\}$ conjunto finito de estados acrescido dos três novos estados.

A partir deste ponto é feita uma nova alteração no modelo da *X-machine*, para incluir dois novos componente dinâmico que permitirão que a *X-machine* possa executar quando houver a ocorrência de um evento no meio ambiente, de tal forma que isso reflita internamente na máquina, e consequentemente na evolução da aplicação.

3.2.1 Modelo Dinâmico para *X-Machine*

Uma *X-machine* na sua definição original é uma máquina puramente matemática, isto é, por ser uma máquina abstrata ela não possui um mecanismo que possibilite a sua execução como uma máquina real. Neste trabalho uma contribuição importante é a inclusão dos mecanismos de (*Eventos* e *Condições*). Com o acréscimo desses dois elementos torna-se possível a execução do modelo matemático da *X-machine* como uma máquina real. Usando os *Eventos* e as *Condições* uma *X-machine* pode executar as transições dos estados, sempre que houver a detecção de um evento, que por sua vez habilita uma condição, que dispara a transição de estado correspondente. O Modelo *Bare*, neste caso, é a implementação de um modelo dinâmico de execução para as *X-machines*. Com isso a máquina abstrata passa a funcionar como uma máquina real.

Os **Eventos** são sinalizações de tempo curto disparadas sempre que ocorrer alguma alteração nos valores das variáveis (monitoradas, controladas ou internas). Por exemplo, se uma variável de entrada recebe um novo valor, ocorrerá um evento sinalizando o fato. A ocorrência deste evento poderá disparar novos eventos, e assim sucessivamente até que o sistema responda à todos os eventos disparados. Por meio da avaliação dos elementos que compõem a sequência dos eventos pode-se verificar a relação de causa e efeito existente entre as variáveis monitoradas, internas e as controladas. A expressão dos eventos E é formado por expressões dos eventos das variáveis de entrada $Inp_e = \{Inp_1 \vee Inp_2 \vee \cdots \vee Inp_m\}$ junto com as expressão dos eventos das variáveis internas $It_e = \{It_1 \vee It_2 \vee \cdots \vee It_n\}$ e a expressão dos eventos

das variáveis de saída $Oup_e = \{Oup_1 \vee Oup_2 \vee \cdots \vee Oup_k\}$, formando a expressão $E = \{Inp_e \vee It_e \vee Oup_e\} \mapsto \{true, false\}$. O valor da expressão de evento E é $false$ para indicar que não há ocorrência de eventos. Quando um evento é detectado, a variável correspondente ao evento passa para $true$ sinalizando a ocorrência do evento, levando a expressão resultante para $true$ durante o tempo suficiente para ser detectado, retornando em seguida para $false$.

Uma **Condição** é uma expressão lógica da seguinte forma $C : (\Sigma \times \Gamma \times \Lambda) \wedge E_i \mapsto \{true, false\}$, composta por variáveis da aplicação utilizando os operadores relacionais $\{=, \neq, >, \geq, <, \leq\}$ e os conectivos lógicos $\{\neg, \vee, \wedge\}$ para formar sentenças complexas. Com isso as funções ϕ passam a ter a seguinte sintaxe:

$$\phi_i :< in, m, C_i, m', out >$$

A presença de um componente do tipo evento é essencial para sincronizar o momento da avaliação da condição com a ocorrência de um evento do sistema. As condições formam o conjunto $C = \{C_0, C_1, \ldots, C_j\}$ e integram o conjunto das variáveis de ambiente da aplicação. Seus valores não são retidos após a avaliação dos seus componentes na presença do evento que a habilitou, com isso a cada execução desse mesmo evento a condição será avaliada novamente e estará sujeita a evolução dos valores das suas variáveis componentes. Comparada com o tempo de duração dos eventos, as condições são de duração maior. Uma vez que estas envolvem o conjunto de variáveis da aplicação a retenção dos valores será no tempo da permanência no estado corrente e, neste caso, a passagem de um estado para outro pode tornar a condição inválida, porém ela só será avaliada novamente na ocorrência do evento ligado a mesma. Na transição do estado “init” para o estado “start”, não há a necessidade da ocorrência de nenhum evento pois esta transição serve para organizar o início da aplicação. Neste caso existe uma condição que é sempre verdadeira, de tal forma que o conjunto de condições fica $C = \{true, C_1, \ldots, C_j\}$, esta condição será utilizada quando a passagem de um estado para outro não envolver a avaliação de uma condição lógica na presença de um evento associado.

Depois de definidos os componentes *Eventos* e *Condições*, é possível concluir a modelagem da aplicação *Blink* com os demais componentes que realizam a parte dinâmica da aplicação:

- $M = (\ mTempo,\ iCorr,\ iTotal,\ cLed\ )$ é o conjunto das variáveis que formam o ambiente da aplicação. Para cada variáveis do ambiente da aplicação existe uma variável de evento correspondente na expressão de **eventos**, tal que $M \mapsto E \mapsto$

$\{true, false\}$.

- Φ é um conjunto finito de funções parciais ϕ que mapeiam uma entrada e um valor de memória numa saída e num novo valor de memória, sujeita a uma **condição** C_j da seguinte forma $\phi : \Sigma \times M \xrightarrow{C_j} \Gamma \times M$.

- F é a função parcial do próximo estado que, dado um estado Q e uma função do tipo Φ, denota o próximo estado. F normalmente é descrito como um diagrama de transição de estado, $F : Q \times \Phi \rightarrow Q$. Esta função terá a seguinte sintaxe:

$$< Q, \phi_i, Q' >$$

  Como a habilitação da função ϕ_i é definida pela avaliação positiva da condição C_j, então a passagem do estado Q para Q' é definida pela avaliação da condição C_j. Com isso a evolução da transição dos estados fica condicionada a avaliação das condições C_j que, por sua vez, são habilitadas pela execução dos eventos E_k, os quais ocorrem sempre que os valores das variáveis de estado da aplicação são alterados, fechando assim o ciclo de execução da aplicação.

- $q_0 = \{Init\}$ é o estado inicial. Todas as aplicações começam pelo estado $Init$.

- $m_0 = (mTempo = 0,\ cLed = \text{Off},\ iCorr = 0,\ iTotal = 1000)$ o conteúdo da memória inicial.

Em seguida é apresentada a definição completa da *X-machine* utilizada pelo Modelo de Desenvolvimento *Bare*, acrescida dos elementos estáticos e dinâmicos para a execução da máquina, os elementos acrescidos estão destacados com *:

$$M = (T^*, \Sigma, \Gamma, \Lambda^*, Q^*, M, E^*, C^*, \Phi, F, q_0, m_0)$$

onde:

- T - conjunto do tipos da aplicação.

- Σ - o alfabeto de entrada.

- Γ - o alfabeto de saída.

- Λ - o alfabeto temporário.

- Q - conjunto dos estados da aplicação.

- M - memória da máquina.

- E - eventos.
- C - condições.
- Φ - função de avaliação de condição.
- F - função de transição de estado.
- q_0 - estado inicial.
- m_0 - memória inicial.

A idéia de utilizar o Modelo de Desenvolvimento *Bare* para construção de aplicações mais complexa será por meio da conexão de várias aplicações modeladas como máquinas básicas, isso leva necessidade da criação de um esquema de comunicação entre essa máquinas, conforme mostra na Figura 3.4. O trabalho atual foi todo concentrado na modelagem da aplicação por meio de uma única máquina, no exemplo a máquina *Blink*, que executa a manipulação dos dados de entrada, realizando alguma computação e envia os dados para a saída.

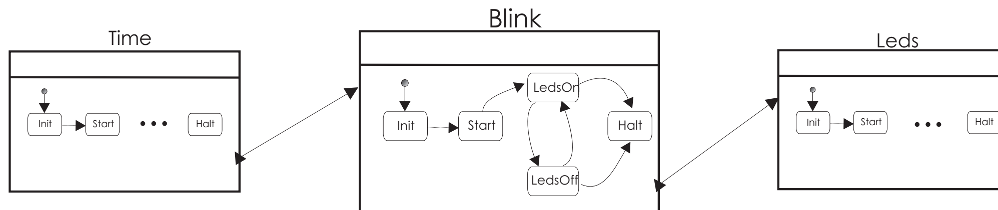


Figura 3.4: Interligação de X-machines

No exemplo da Figura 3.4 a tarefa atribuída ao sensor foi trocada pela máquina *Time*, a qual envia um *tick* de tempo (mTempo), em intervalos regulares, para a máquina Blink resultando na Figura 3.5. O comunicador foi substituído por máquina *Leds*, que por sua vez liga e desliga os leds como resposta ao recebimento dos ticks. A execução da aplicação deve considerar um tempo máximo de execução. A descrição completa para a aplicação *Blink* por meio do Modelo *Bare* tem a seguinte definição, com todos os elementos instanciados:

- $T = (Booleano = \{\ true,\ false\ \}, Tempo = [\ 0, 10000\ ],\ Switch = \{\ On,\ Off\ \})$
- $\Sigma = (mTempo = \{Tempo\})$
- $\Gamma = (cLed = \{Switch\})$
- $\Lambda = (iCorr = \{Tempo\},\ iTotal = \{Tempo\})$

- $Q \cup \{I, S, H\} = \{LedOn,\ LedOff,\ Init,\ Start,\ Halt\}$
- $M = (\ mTempo,\ iCorr,\ iTotal,\ cLed\ )$
- $E = \{e_1 = mTempo\ \vee\ e_2 = iCorr\ \vee\ e_3 = iTotal\ \vee\ e_4 = cLed\} = \{e_1 = false \vee e_2 = false \vee e_3 = false \vee e_4 = false\}$
- $C = \{$
  - $[c_0 = true]$
  - $[c_1 = e_1]$
  - $[c_2 = (iCorr \leq iTotal) \wedge e_1]$
  - $[c_3 = (iCorr > iTotal) \wedge e_1]$

  $\}$
- Φ
  - $\phi_0 = (_, _, c_0, _, _)$
  - $\phi_1 = (_, _, c_1, _, cLed = On)$
  - $\phi_2 = (_, iCorr, c_2, (iCorr + 1), cLed = Off)$
  - $\phi_3 = (_, iCorr, c_3, (iCorr + 1), cLed = Off)$
  - $\phi_4 = (_, iCorr, c_2, (iCorr + 1), cLed = On)$
- F
  - $(Init, \phi_0, Start)$
  - $(Start, \phi_1, LedOn)$
  - $(LedOn, \phi_2, LedOff)$
  - $(LedOn, \phi_3, Halt)$
  - $(LedOff, \phi_4, LedOn)$
  - $(LedOff, \phi_3, Halt)$
- $q_0 = \{Init\}$
- $m_0 = (mTempo = 0,\ cLed = \text{Off},\ iCorr = 0,\ iTotal = 1000)$

Depois de concluída a fase de especificação da aplicação, o próximo passo é a transformação da especificação no modelo tabular para em seguida ser executada na plataforma escolhida, neste caso existem duas opções de execução: 1- No simulador *ExecutorXM* no ambiente PC; 2- Execução no hardware real no Robô NXT. A verificação do Modelo *Bare* é feita por meio da geração do Modelo do Sistema extraído a partir do Modelo Tabular, sendo que a verificação é executada na ferramenta de verificação de modelos chamada NuSMV.

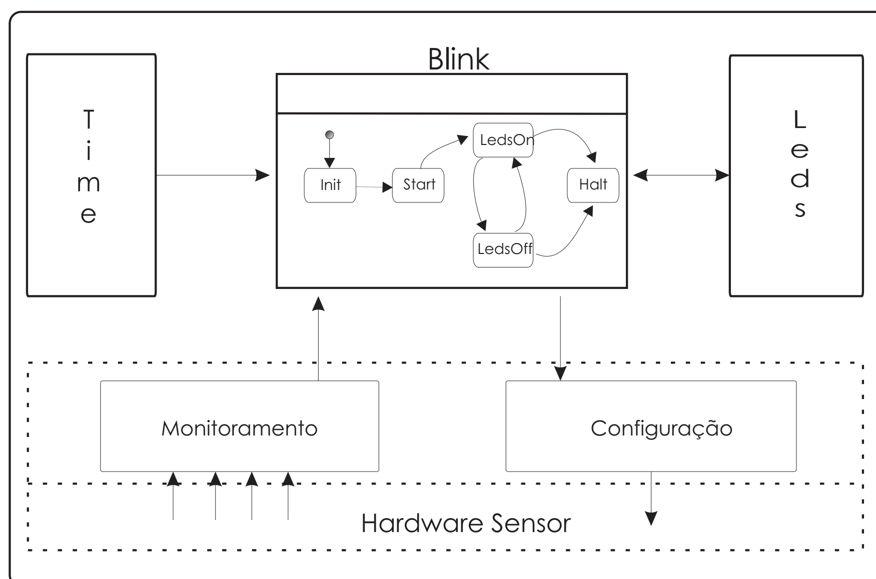


Figura 3.5: Esquema para a Aplicação Blink

3.3 Mapeamento para o Modelo Tabular

Depois de realizada a descrição da aplicação utilizando o Modelo *Bare* o próximo passo é a geração da aplicação no formato executável, ou seja, a transformação da especificação da aplicação para o formato tabular. A estrutura tabular para construir o modelo de execução da aplicação, o qual será executada pelo executor, foi escolhida para tornar a execução das aplicações mais simples. Os modelos tabulares tem sido utilizados durante anos como ferramentas para especificação de software [Heitmeyer et al., 1996; Janicki & Khedri, 2001; Breen, 2005], com a alegação de serem mais fáceis de ler e compreender que as expressões em lógica de predicados.

Foi desenvolvida uma ferramenta chamada **GeradorXM**, descrita na Seção 4.3.4, onde a aplicação é descrita com um alto nível de abstração e em seguida transformada do modelo de alto nível para o modelo de execução no formato tabular. No formato

tabular a aplicação pode ser executada pelo **ExecutorXM**, descrito na seção 4.3.3. A transformação é feita automaticamente e a tabela resultante é composta das seguintes colunas:

- $Source$ representa os estados iniciais das transições. A coluna $Source$ contêm todos os estados da aplicação incluindo os estados $Init$, $Start$, $Halt$, os quais sempre farão parte das aplicações. O objetivo destes estados é executar as operações de configuração ou inicialização da aplicação e indicar quando o processamento da aplicação chegou ao final, o que ocorre quando o estado $Halt$ for alcançado. O primeiro estado a ser executado em qualquer aplicação é sempre o estado $Init$, que passa em seguida para o estado $Start$. A partir daí as transições são controladas por eventos. Todas as aplicações devem terminar sua execução no estado $Halt$.

- $Input$ representa o evento de entrada que será tratado. Este evento poderá trazer alguma informação com dados ou apenas a indicação da ocorrência do evento.

- Mem_input são valores internos à máquina que serão considerados como entrada pela função de avaliação de condição.

- $Target$ são os estados alvos das transições. Caso a função de avaliação de condição ϕ seja satisfeita, então uma saída para a memória e uma saída da máquina são produzidas e o estado alvo passa a ser o estado corrente.

- *Condition* é a condição que será avaliada para habilitar a transição do estado *corrente* para o estado *alvo*. Essa condição possui como dados de entrada as colunas $Input$ e Mem_input. As condições são sincronizadas pelos eventos recebidos. Elas podem estar ligadas a algum evento do sistema e a detecção deste evento habilita o momento da verificação da condição.

- Mem_output são valores internos à máquina que serão atualizados caso a transição ocorra quando a função de avaliação de condição ϕ é satisfeita.

- *Output* é o valor produzido como resultado da transição entre os estados. A saída poderá ser o disparo de algum evento para a máquina seguinte ou algum dado de saída.

As colunas da tabela da aplicação serão preenchidas com as informação diretamente do modelo. Para isso o seguintes algoritmo é executado:

1. O conjunto F das funções de transição de estado constroem a coluna $Source$. Para cada função de transição de estado é criada uma linha na tabela da aplicação, mais a linha do estado "halt" para "halt", que produz a parada do sistema. No caso da aplicação *blink* serão criadas 7 linhas na tabela da aplicação correspondentes ao modelo da aplicação.

2. Para cada linha inserida na tabela que corresponde a uma função de transição de estado F, são inserido também os elementos que compõem a função de avaliação de condição ϕ. Os componentes desta função são:

   - Um elemento de entrada da máquina - $Input$
   - Um elemento de entrada vindo da memória - Mem_input
   - Uma condição que está ligada a um evento - C_j
   - Um elemento de saída da máquina - *Output*
   - Um elemento de saída para a memória - Mem_output

   Dos elementos componentes da função avaliação de condição ϕ, são inseridos na tabela diretamente apenas as saídas (*Output* e Mem_output) e o corpo da condição C_j, uma vez que estes são independentes para cada linha da tabela e representam todos os caminho de saída para cada estado.

   Até este ponto a tabela é construída baseada diretamente nas respectivas funções de transição e nas condição de transição para cada função, gerando cada linha individual. A partir deste ponto serão considerados todas as entradas dos estados como um único canal de entrada para o estado corrente e, dependendo da condição habilitada, as saídas serão escolhidas.

3. Os elementos de entrada dos estado devem ser considerados em conjunto. Na implementação foi usada uma função que captura os eventos de entrada, onde é possível indicar quais eventos serão tratados pelo estado corrente. Ela devolve como resposta o tipo do evento e a sinalização da captura do mesmo na tabela de eventos.

4. Para o conjunto de linhas que representam um estado a entrada é substituída por uma chamada para a função de tratamento de eventos, passando como parâmetros todos os eventos tratados pelo estado. No caso do estado *LedOn* o evento tratado é o *tempo*.

5. Para a coluna de entrada de memória Mem_input o objetivo é buscar da memória todos os elementos que serão utilizados pela condição de transição ou pela saída.

Desta forma a tabela resultante para a aplicação *blink* é mostrada na Tabela 3.2. Nela estão resumidos todos os componentes necessários para a execução da aplicação.

S Current	**Source**	**Input**	**Mem_input**	**Target**	**Condition**	**Mem_output**	**Output**
true	init	_	_	start	c_0	$iCorr = 0; iTotal = 100$	_
false	start	$mTempo$	$(iCorr, iTotal)$	ledOn	c_1	$iCorr = iCorr + 1$	cLed=On
false	ledOn	$mTempo$	$(iCorr, iTotal)$	ledOff	c_1	$iCorr = iCorr + 1$	cLed=Off
false	ledOn	$mTempo$	$(iCorr, iTotal)$	halt	c_2	$iCorr = iCorr + 1$	cLed=Off
false	ledOff	$mTempo$	$(iCorr, iTotal)$	ledOn	c_1	$iCorr = iCorr + 1$	cLed=On
false	ledOff	$mTempo$	$(iCorr, iTotal)$	halt	c_2	$iCorr = iCorr + 1$	cLed=Off
false	halt	$mTempo$	$(iCorr, iTotal)$	halt	_	_	_

Tabela 3.2: Tabela para a Aplicação Blink

3.4 Verificação Formal do Modelo *Bare*

O principal objetivo da verificação de modelos (*Model Checking*) é verificar se o modelo desenvolvido M satisfaz determinadas propriedades γ especificadas pelo usuário, ou seja, se uma fórmula proposicional γ é verdadeira em todos ou em alguns estados s da árvore de computação M, ou seja se M satisfaz γ. Um estado é definido pela atribuição dos valores de todas as variáveis do modelo em um determinado instante. Este método enumera todos os estados alcançáveis, dados os estados iniciais e a regra de transição, e verifica as propriedades de acordo com as especificações fornecidas. Isso pode levar a uma explosão de estados a serem verificados, porém as ferramentas modernas possuem a capacidade de lidar com tal problema. Duas principais lógicas temporais podem ser utilizadas para a descrição das propriedades a serem testadas, conforme descritas na seção 2.3, a lógica temporal LTL e CTL. A especificação, depois de escrita na lógica temporal, pode ser avaliada por uma ferramenta. Optou-se pela utilização da ferramenta NuSMV, que determinará a veracidade ou falsidade da especificação pela avaliação da máquina de estados resultante do modelo. Caso a especificação seja avaliada como falsa, então a ferramenta constrói um contra-exemplo no formato de um *trace* na máquina de estados que torna a especificação falsa. A

linguagem de entrada para a ferramenta de verificação de modelos NuSMV, são máquinas de estados finitos(MEFs), que podem ser síncronas ou assíncronas, detalhadas ou abstratas. Essa entrada facilita o mapeamento do Modelo *Bare* para o modelo de entrada requirido pela ferramenta NuSMV. Os tipos aceitos pela ferramenta NuSMV são os tipos booleanos, escalares e vetor de tamanho fixo. O primeiro passo para realizar a verificação formal do sistema é a construção de um modelo do sistema a partir do Modelo de Desenvolvimento *Bare*. O modelo deve descrever a relação de transição dos estados, por meio das evoluções válidas para a máquina, formando um sistema de transição. A construção do modelo do sistema a partir do Modelo *Bare* será alcançada por meio do modelo tabular, porque nele estão presentes todos os elementos necessários para a extração do sistema de transição. O mapeamento do modelo tabular para o modelo do sistema é feito automaticamente pelo *GeradorXM* com base no algoritmo de alto nível apresentado a seguir:

- 1 - Construção dos elementos da seção VAR com a definição dos tipos para as variáveis:
- 1.1 - Todos os elementos da Memória são relacionados, e o seu TIPO é solicitado explicitamente para o usuário.
- 1.2 - Todos os elementos de Eventos são relacionados com o tipo BOOLEAN.
- 1.3 - Todos os elementos dos estados do sistema, exceto os estados (INIT, START), são relacionados para constituir o tipo da variável *estados*, que representa todos os possíveis estados do sistema. Os estados (INIT, START) são excluídos porque possuem a condição de transição sempre verdadeira.
- 2 - Construção dos elementos da seção ASSIGN com a definição dos valores iniciais para as variáveis:
- 2.1 - Todos os elementos da Memória são relacionados e o seu VALOR INICIAL é solicitado explicitamente para o usuário.
- 2.3 - A variável *estados* recebe o valor alvo da transição "start". Isso representa o estado inicial do sistema.
- 3 - Construção da seção de próximos estados do sistema, seção CASE:
- 3.1 - Para cada linha V_i da tabela da aplicação é construída uma expressão para o próximo estado do sistema, da seguinte forma: $estados = V_i[1]\ \&\ V_i[5]\ :\ V_i[4]$.

Cada linha V_i da tabela da aplicação possui os seguintes campos de acordo com a Seção 3.3: $V_i[1] = source, V_i[2] = input, V_i[3] = mem_input, V_i[4] = target, V_i[5] = condition, V_i[6] = mem_output, V_i[7] = output$. As linhas da tabela da aplicação representam a evolução dos estados do sistema a partir de um estado fonte $V_i[1]$ para um estado alvo $V_i[4]$, sujeito a avaliação de uma condição de transição de estados $V_i[5]$. Para cada linha desta tabela, excluídas as linhas que possuem os estados (init, start, halt) como estado fonte, será construída uma expressão para o próximo estado do sistema, que completará o modelo formal para a aplicação. Na Figura 3.6 é mostrada a saída produzida pelo *GeradorXM* para a aplicação *blink*. Os elementos da especificação de um sistema utilizando a linguagem de entrada para a ferramenta NuSMV são descrito na Seção 2.4.

```
MODULE main

VAR
iCorr : 0..101;
iTotal : 0..100;
mTempo: boolean;
estados: {halt,ledsOn,ledsOff} ;

ASSIGN
init (iCorr):= 0;
init (iTotal):= 15;
init(estados) := ledsOn ;
next(estados) := case
                  estados = ledsOn & iCorr <= iTotal & mTempo : ledsOff;
                  estados = ledsOff & iCorr <= iTotal & mTempo : ledsOn;
                  estados = ledsOn & iCorr > iTotal & mTempo : halt;
                  estados = ledsOff & iCorr > iTotal & mTempo : halt;
     1 : estados ;
     esac;
next(iCorr) := case
                  estados = ledsOn & iCorr <= iTotal & mTempo : iCorr+1;
                  estados = ledsOff & iCorr <= iTotal & mTempo : iCorr+1;
                  estados = ledsOn & iCorr > iTotal & mTempo : iCorr+1;
                  estados = ledsOff & iCorr > iTotal & mTempo : iCorr+1;
     1 :iCorr ;
     esac;
next(iTotal) := case
     1 :iTotal ;
     esac;
```

Figura 3.6: Modelo do Sistema para a Aplicação Blink

Depois da construção do modelo do sistema, o próximo passo é a definição das propriedades que serão verificadas contra o modelo. Uma das abordagens proposta para verificação de sistemas reativos é baseada na especificação das propriedades represen-

tando os requisitos que o sistema deve satisfazer [Chang et al., 1991]. Uma outra abordagens mais geral para especificação de propriedades de sistema computacional modelado por estados finitos é baseada em padrões para representação, codificação e reuso da especificações das propriedades, onde as propriedades são especificadas usando lógica temporal ou expressões regulares [Dwyer et al., 1999]. Um grande número de requisitos podem ser associado a um determinado sistema que será submetido a uma verificação. Muitos dos requisitos, apesar de específicos de cada sistema, são tão comuns que podem ser categorizados [Ferreira, 2005]. Como exemplo de requisito comum, um determinado sistema não deve executar uma determinada ação, ou ainda, não deve alcançar um determinado estado em que o sistema não possa mais continuar a sua execução.

A verificação de modelos (*model checking*) se baseia numa exploração exaustiva dos estado do sistema. Por isso é bem apropriado para a análise das propriedades expressas em termo de alcançabilidade de estados [Loer, 2003]. Uma taxonomia tradicionalmente aceita em verificação de sistemas utiliza duas propriedades principais [Lamport, 1977]:

- propriedade de segurança (safety): "uma coisa indesejada nunca ocorre".
- propriedade de vivacidade(liveness): "uma coisa desejada eventualmente ocorre".

A propriedade de segurança (safety) especifica que, sobre certa condições, uma configuração onde alguma propriedade indesejada γ ocorra, está ausente. A ausência só pode ser provada se todos os caminhos de execução forem testados. Assim, uma forma de expressar essa propriedade em CTL é:

$$AG \sim \gamma$$

Uma aplicação típica para a ausência de propriedade é a exclusão mútua de duas expressões σ e γ, que poderia significar a acesso indevido à região crítica, que em CTL é escrito:

$$AG \sim (\sigma \wedge \gamma)$$

Uma outra necessidade é de expressar a propriedade de segurança sob uma certa condição, isto é, uma propriedade σ está ausente a menos que uma outra propriedade γ seja verdadeira, em CTL é expresso:

$$A\,[\sim \gamma\; U\; \sigma]$$

O operador $U - until$ é muito forte nesta relação e implica que σ deve impreterivelmente ocorrer no sistema, o que pode não ser verdadeiro sempre. As ferramentas não

implementam o operador $Unless$, pois existe uma equivalência lógica que faz uso do operador $Until$, e em CTL fica:

$$!E\ [(\sigma\ U\ (\gamma \wedge \sigma)]$$

Até este ponto as propriedades foram criadas para especificar a ausência dos estados onde a propriedade indesejada poderia ocorrer. A mesma abordagem pode ser usada para a especificação de propriedade visando a presença dos estados onde a propriedade desejada ocorre. Uma propriedade desejada γ que deve ocorrer universalmente é especificada como uma propriedade invariante global:

$$AG\ \gamma$$

Essa propriedade pode ser refinada para especificar uma invariante local, isto é, uma propriedade σ que deve ocorrer em qualquer estado onde uma outra propriedade γ ocorra:

$$AG\ [\gamma \rightarrow \sigma]$$

A propriedade de vivacidade (liveness) expressa que, sob certas circunstâncias, um estado onde uma expressão γ ocorre será eventualmente alcançado. O exemplo mais simples de propriedade de vivacidade é ausência de impasse (deadlock freedom). De forma geral a propriedade a seguir especifica que todo estado possui um sucessor imediato, em CTL:

$$AG\ EX\ True$$

Outro uso frequente para a propriedade de vivacidade é a resposta a uma determinada configuração, isto é, sempre que em algum estado a propriedade γ ocorrer, eventualmente, um estado será no visitado, no futuro, em que a propriedade σ ocorre, em CTL:

$$AG(\gamma \rightarrow AF\ \sigma)$$

Apesar da propriedade de vivacidade declarar que alguma propriedade eventualmente ocorrerá, ela não especifica *quando* isso acontecerá. Alguns autores argumentam que na prática o *quando* é mais esclarecedor que simplesmente saber que ocorrerá a propriedade. Neste caso uma especficação mais útil determina que em qualquer estado visitado onde uma propriedade γ ocorre, um estado em resposta, onde a propriedade σ ocorre, será visitado *num intervalo de m até n unidades de tempo*. Em NuSMV a especificação dessa propriedade de vivacidade delimitada (*bounded liveness properties*)

é expressa:

$$AG(\gamma \rightarrow \; ABF \; m..n \; \sigma)$$

As propriedades de vivacidade delimitada são mais específicas quando uma resposta é requerida.

Outras formas de expressar uma propriedades de vivacidade é fazendo uso das expressões *Until*, $A[\gamma \; U \; \sigma]$ e $E[\gamma \; U \; \sigma]$. Essa interpretação é possível porque que todas elas expressam que a propriedade σ eventualmente ocorrerá.

3.5 Reconfiguração no Modelo *Bare*

Durante o desenvolvimento do Modelo de Desenvolvimento *Bare*, várias analogias com o modelo de computação reconfigurável foram traçadas e, assim o Modelo *Bare* possui aspectos de comportamento que podem ser comparados ao modelo de reconfiguração de hardware. A reconfiguração de hardware pode ser *Estática* ou *Dinâmica* [Skliarova, 2004]. No Modelo *Bare* a execução de uma aplicação pode ser composta por vários cenários, por exemplo, um para a inicialização, depois um cenário de trabalho normal que executará na maior parte do tempo e um para o tratamento de situações não previstas ou quando houver necessidade de adaptação para melhoria na atuação do sistema. Estes cenários evoluem durante o tempo de execução se comportando como uma forma de reconfiguração dinâmica. Desta forma a execução da aplicação inicia com uma tabela, que poderá ser substituída por outras tabelas quando necessário. As tabelas, desta forma, guardam informações que formam os cenários da aplicação e, a medida que as condições mudam, a substituição dos cenários poderá ser necessário. Esse mecanismo pode ser melhor visualizado por meio da Figura 3.7.

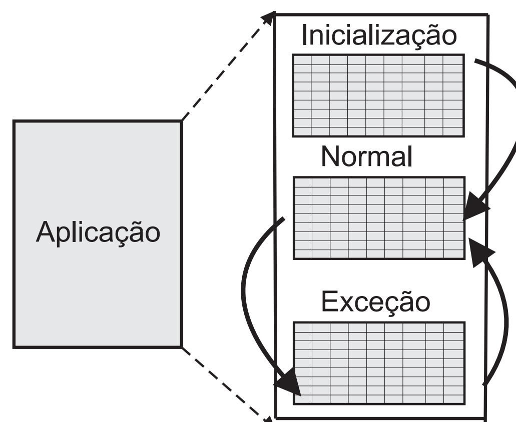


Figura 3.7: Reconfiguração Dinâmica no Modelo *Bare*

A reconfiguração *Dinâmica* de hardware pode ainda ser dividida em *Contexto Único* e *Contexto Múltiplo* [Skliarova, 2004]. Além do *Contexto Parcialmente Reconfigurável*, onde pequenas partes da camada ativa podem ser alterados sem a necessidade de reprogramar todo o dispositivo. Este modelo é interessante porque pode ser feito em tempo menor que os anteriores e, caso a zona alterada não esteja em uso no momento, a reconfiguração poderá ser feita em paralelo com a execução da aplicação. Os contextos de reconfiguração são mostrados no Capítulo 2, Figura 2.7. O Modelo *Bare* implementa as características de reconfiguração dinâmica de contexto múltiplo quanto faz uso dos cenários, onde cada cenário representa um contexto diferenciado para cada possibilidade prevista para a aplicação, e implementa também o contexto parcialmente reconfigurável, onde as linhas da tabela da aplicação podem ser substituídas independentemente umas das outras, além disso durante a execução do estado corrente apenas as linhas que representam esse estado estão sendo manipuladas pelo executor, com isso as linhas dos outros estados podem ser alteradas, ou reconfiguradas sem interferir com o processo de execução da aplicação.

Para justificar como a Reconfiguração de Aplicação executa no Modelo *Bare* os seguintes aspectos devem ser compreendidos:

- Toda aplicação é composta por um conjunto de linhas formando uma tabela e juntas executam uma função. As linhas que possuem o mesmo estado de origem (source) são agrupadas para formar as alternativas para o estado corrente, durante a execução da aplicação.

- Não há a necessidade da separação física entre linhas das tabelas de aplicações distintas, uma vez que durante a execução o próximo estado é definido pela última linha do estado corrente que foi executado com sucesso.

- Com isso pode-se ter mais de uma aplicação carregada pelo executor, porém somente as linhas correspondentes a aplicação em execução é que serão consideradas.

O esquema de reconfiguração será como descrito na Figura 3.8, neste caso os cenários com as aplicações ficam prontos para serem carregados, podem estar juntos no hardware de maneira completa ou parcialmente, a aplicação principal aguarda o evento denominado “Executor(App-Nome)”.

Ao disparar o evento para reconfiguração deverá ser passado o nome da aplicação selecionada para ser executada. Na transição a tabela da aplicação é carregada e o controle da execução é passado para a mesma. Quando a aplicação alcançar o estado

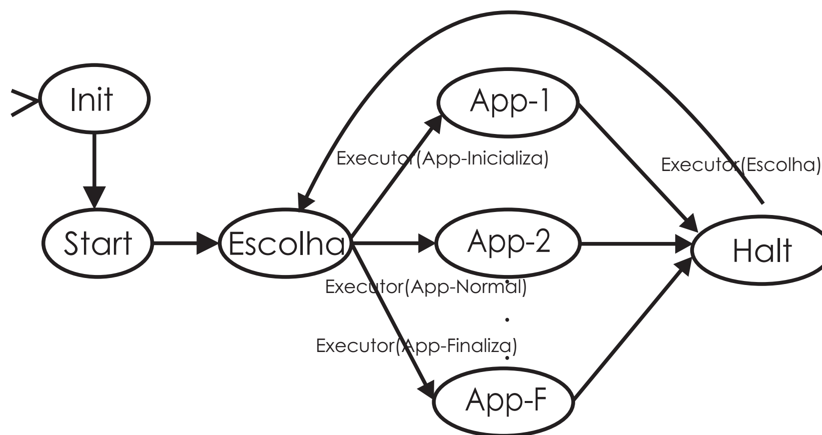


Figura 3.8: Esquema de Reconfiguração

"halt" este dispara o evento "Executor(Escolha)", assim a aplicação geral assume o controle e dispara outra aplicação para executar.

Capítulo 4

Implementação do Modelo *Bare*

Neste capítulo são apresentados os detalhes da implementação do Modelo de Desenvolvimento *Bare*. É apresentado uma definição formal para o Modelo de Execução, o qual servirá de base para a confecção do algoritmo do interpretador do modelo tabular *ExecutorXM*. São mostrado detalhes da implementação do interpretador do modelo tabular, tanto para *PC* como para o Robô NXT. É apresentado também detalhes da ferramenta *GeradorXM* para o desenvolvimento das aplicações no Modelo *Bare*.

4.1 Modelo de Execução *Bare*

Para a construção do modelo de execução *Bare* foi feita uma definição formal da execução da aplicação utilizando grafos direcionados, esta definição está baseada em [Chang & Lee, 1973].

Um grafo direcionado consiste de um conjunto V, não vazio, de elementos chamados vértices, de um conjunto A, disjunto de V, de elementos chamados arestas e, de um mapeamento D, de A em $V \times V$. O conjunto D é chamado de "mapeamento de incidência direta" associado com o grafo direcionado. Se $a \in A$ e $D(a) = (v, v\prime)$, então a possui v como vértice inicial e $v\prime$ como vértice terminal. O número de vértices é considerado finito. Um caminho em um grafo direcionado é uma seqüência de arcos $a_1, a_2, \ldots, a_n$, onde todo arco a_k tem v_{k-1} como seu vértice inicial e v_k como vértice final.

Neste contexto, uma aplicação é formada por um vetor de variáveis de entrada ou monitoradas $\bar{m} = (m_1, \ldots, m_t)$, um vetor de variáveis internas $\bar{i} = (i_1, \ldots, i_m)$ e um vetor de variáveis de saída ou controladas $\bar{c} = (c_1, \ldots, c_n)$ e um grafo finito direcionado (V, A), onde as seguintes condições são satisfeitas:

- No grafo (V, A) existe somente um vértice chamado "start", denotado por S, $S \in V$, que não é terminal de nenhum outro arco, exceto do vértice "init", denotado por I, $I \in V$; existe somente um vértice de finalização da aplicação chamado "halt", denotado por H, $H \in V$, que não é vértice inicial de nenhum outro arco; todo vértice v está em algum caminho do start(S) até halt(H).

- No grafo (V, A), cada arco a que não incide em H está associado com uma fórmula livre de quantificador do tipo $P_a(\bar{m}, \bar{i})$ e uma atribuição $\bar{i} \leftarrow f_a(\bar{m}, \bar{i})$; cada arco que incide no vértice H está associado com uma fórmula livre de quantificador do tipo $P_a(\bar{m}, \bar{i})$ e uma atribuição $\bar{c} \leftarrow f_a(\bar{m}, \bar{i})$, onde P_a é chamado de *predicado de teste* associado com o arco a e $P_a(\bar{m}, \bar{i})$ é chamado de *fórmula de teste* associado com o arco a.

- Para cada vértice v $(v \neq H)$, seja $a_1, a_2, \ldots, a_r$ todos os arcos deixando v e seja $P_{a1}, P_{a2}, \ldots, P_{ar}$ os predicados de teste associados com os arcos $a_1, a_2, \ldots, a_r$ respectivamente. Então para todo $\bar{m}$ e $\bar{i}$, um e, somente um, dos $P_{a1}(\bar{m}, \bar{i}), P_{a2}(\bar{m}, \bar{i}), \ldots, P_{ar}(\bar{m}, \bar{i})$ é verdadeiro (*true*).

Algumas vezes existe apenas um único arco deixando um certo vértice. Neste caso a fórmula de teste associada com o arco pode ser considerada "*true*" e convenientemente ignorada. A Figura 4.1 mostra o grafo direcionado da execução da aplicação Blink. Nele podemos verificar todas as propriedades necessárias para a execução da aplicação. O grafo possui somente um vértice "init"(I), "start"(S) e um nó "halt"(H). Todo vértice está num caminho de S até H. Cada arco está associado com uma fórmula que seleciona o arco a ser seguido e uma atribuição, quando o arco é escolhido, assim este grafo representa a execução da aplicação.

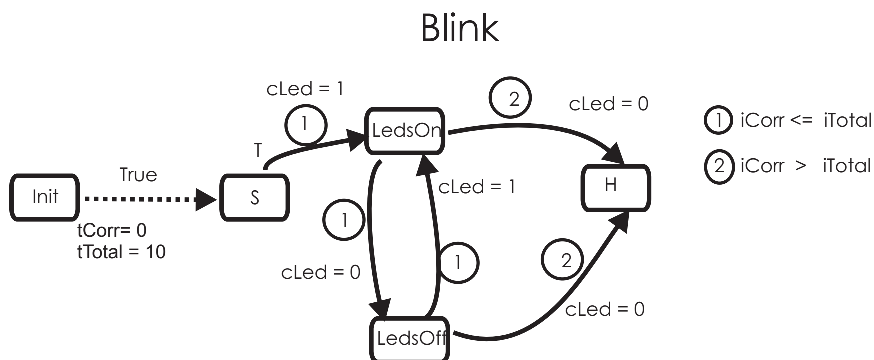


Figura 4.1: Grafo de Execução do Blink

Considerando o grafo direcionado da Figura 4.1, podemos estabelecer que dado uma entrada $\bar{m}$ para a aplicação, a execução da aplicação ocorrerá de acordo com o

seguinte algoritmo:

- Passo 1 - A execução inicia no vértice $init(I)$ e em seguida passa o controle para o vértice $start(S)$.
- Passo 2 - Seja $j = 0$, $v^j = S$ e seja $\bar{i}^j$ as variáveis internas.
- Passo 3 - Se $v^j = H$, então a execução termina, caso contrário vá para o próximo passo.
- Passo 4 - Seja a_k o arco no qual v^j é o vértice inicial e a fórmula de teste associada ao arco é verdadeira:

  $$P_{ak}(\bar{m}, \bar{i}^j) = true$$

  Seja v^{j+1} o vértice terminal de a_k. Então o controle se move por meio de a_k para v^{j+1} e uma das seguintes atribuições é executada:

  - $\bar{i}^{j+1} \leftarrow f_{ak}(\bar{m}, \bar{i}^j)$ se v^{j+1} não é H
  - $\bar{c} \leftarrow f_{ak}(\bar{m}, \bar{i}^j)$ se v^{j+1} é H
- Passo 5 - Seja $j = j + 1$ vá para o passo 3.

A execução é finita se, e somente se para algum k, $v^k = H$. Neste caso dizemos que a execução da aplicação termina para a entrada $\bar{m}$, produzindo como saída $\bar{c}$.

4.2 Executor do Modelo *Bare*

A partir do algoritmo formal para o modelo de execução foi construído o **ExecutorXM** para o Modelo *Bare*. O objetivo de especificar formalmente o modelo de execução é estudar uma maneira de implementar o *ExecutorXM* utilizando o menor código possível e, desvinculado da aplicação. A idéia é que o executor receba a aplicação descrita no formato tabular, para executá-la diretamente no hardware. Com isso a aplicação fica independente dos detalhes do hardware, e a aplicação passa para um nível de controle mais abstrato sobre o hardware. A independência da aplicação do mecanismo de execução permitirá que essa possa ser alterada em tempo de execução. Isso possibilita realizar inclusão, exclusão ou substituição de partes da aplicação. Assim a programação ou a reprogramação do hardware se torna viável.

Após a descrição da aplicação *Blink* no Modelo *Bare*, o modelo é transformado numa estrutura de tabelas. A Tabela 4.1 guarda informações dos estados, transições e

valores de entrada e saída. A Tabela 4.2 contém as variáveis da memória e os respectivos eventos e a Tabela 4.3 contém as condições de execução, conforme descritas no modelo. Essas estruturas tabulares devem ser capazes de armazenar todos os elementos necessários para executar uma aplicação.

S Current	**Source**	**Input**	**Mem_input**	**Target**	**Condition**	**Mem_output**	**Output**
true	init	_	_	start	c_0	$iCorr = 0; iTotal = 100$	_
false	start	$mTempo$	$(iCorr, iTotal)$	ledOn	c_1	$iCorr = iCorr + 1$	cLed=On
false	ledOn	$mTempo$	$(iCorr, iTotal)$	ledOff	c_1	$iCorr = iCorr + 1$	cLed=Off
false	ledOn	$mTempo$	$(iCorr, iTotal)$	halt	c_2	$iCorr = iCorr + 1$	cLed=Off
false	ledOff	$mTempo$	$(iCorr, iTotal)$	ledOn	c_1	$iCorr = iCorr + 1$	cLed=On
false	ledOff	$mTempo$	$(iCorr, iTotal)$	halt	c_2	$iCorr = iCorr + 1$	cLed=Off
false	halt	$mTempo$	$(iCorr, iTotal)$	halt	_	_	_

Tabela 4.1: Tabela de Execução para Aplicação Blink

Memory	**Events**	**Value**
mTempo	e_1	false
iCorr	e_2	false
iTotal	e_3	false
cLed	e_4	false
	$\bigvee E$	false

Tabela 4.2: Tabela de Eventos Monitorados

Conditions	**Expression**	**Value**
c_0	true	true
c_1	$iCorr \leq iTotal) \wedge e_1$	false
c_2	$(iCorr > iTotal) \wedge e_1$	false

Tabela 4.3: Tabela de Condições

A execução do Modelo *Bare* é descrita por meio da iteração do algoritmo de execução para demonstrar a sua mecânica. Todas as aplicações iniciam no estado "init" e em seguida passam para o estado "start", pela avaliação da condição c_0 que é sempre verdadeira. A partir deste ponto a aplicação fica sujeita a ocorrência dos eventos do sistema. Na ocorrência de um valor de entrada, no exemplo a variável $mTempo$, o valor é recebido na memória e um evento correspondente é disparado. Observando a Tabela

4.2, o evento vinculado a $mTempo$ é o evento "e_1". Com isso a expressão de disjunção dos eventos ($\bigvee E$) tem seu valor alterado para $true$, indicando a ocorrência do evento. Na tabela de condições 4.3, as condições vinculadas ao evento disparado ficam prontas para serem avaliadas, porém elas só serão avaliadas se fizerem parte do conjunto de estado corrente. O evento habilita apenas a verificação da condição ligada a ele, sendo que os elementos componentes da expressão são quem realmente determinam o valor final da condição. Em seguida, apenas as linhas que fazem parte do conjunto de estado corrente podem ter as sua condições avaliadas, seguindo o exemplo temos apenas uma linha no estado $start$ de acordo com a tabela de execução 4.1. Na ocorrência do evento e_1 a condição c_1 fica habilitada (vide Tabela 4.1) para ser avaliada e depende dos valores dos componentes da expressão, neste caso ela tem seu valor $true$, pois "$iCorr \leq iTotal$" também é $true$. Caso a condição esteja habilitada, então as variáveis de memória "$iCorr = iCorr + 1$" (Mem_output) e saída "$cLeds = On$" ($Output$) tem seus valores atualizados, o que pode leva ao disparo de novos eventos, os quais podem estar vinculados ou não a condições, no exemplo apenas $mTempo$ é tratada. Depois de atualizar os valores das variáveis o estado corrente é atualizado para o valor do estado alvo ($ledsOn$) que está na coluna $Target$, no exemplo o novo estado será "LedsOn". Em seguida o todo o processo é reiniciado até que o estado "halt" seja alcançado. Esse procedimento é executado pelo interpretador conforme o algoritmo apresentado na Figura 4.2, o qual demonstra como o executor faz a manipulação das tabelas para executar a aplicação.

```
FUNCTION Executor(Maquina)
Curr = "Init";
WHILE (Curr != "Halt"){
       Current = GetCurrentSet(Maquina, Curr)
       DO Current.Input
       FOREACH line IN Current DO
           DO Line.MemoryIn;
           Condition = RETURN Line.Condition;
           IF Condition is true
             THEN
                DO Line.Output;
                DO Line.MemoryOut;
                Curr = Line.Target
                BREAK
             END
}
```

Figura 4.2: Algoritmo do *ExecutorXM*

4.3 Implementação do Executor do Modelo *Bare*

Com o objetivo de testar o Modelo de Desenvolvimento *Bare* na geração e execução das aplicações para Sistemas Reativos Autônomos, foram desenvolvidas duas versões para o *ExecutorXM*. Ambas foram implementadas na linguagem *Lua* [Lua, 2009] [Ierusalimschy et al., 1996], sendo que uma versão foi criada como um *simulador* para executar no ambiente do PC convencional e a outra versão foi desenvolvida para executar no hardware do robô NXT da Lego [Lego NXT, 2009]. A versão para PC foi desenvolvida como um ambiente que executa o algoritmo do ExecutorXM, com a simulação dos eventos do meio ambiente por meio de 4 botões disponíveis, onde o usuário pode selecionar (disparar) um dos eventos para indicar a ocorrência do respectivos eventos para o sistema. A simulação está restrita para tratar apenas os eventos capturados pelo robô NXT real, no caso: tempo, presença, toque e som. A saída da versão para PC é realizada por meio de uma janela de mensagens. A escolha pela utilização de um robô NXT para o hardware real foi pelo fato de que o robô NXT possui elementos sensores simples e que estão prontos para serem usados sem muita complicação, tais sensores serão usados na detecção dos eventos do meio ambiente. O robô também possui alguns elementos que podem ser usados como atuadores para demonstrar a evolução de uma aplicação executando num hardware real, este elementos são os motores ou o display. Os elementos que compõem o robô NXT podem ser vistos na Figura 4.3

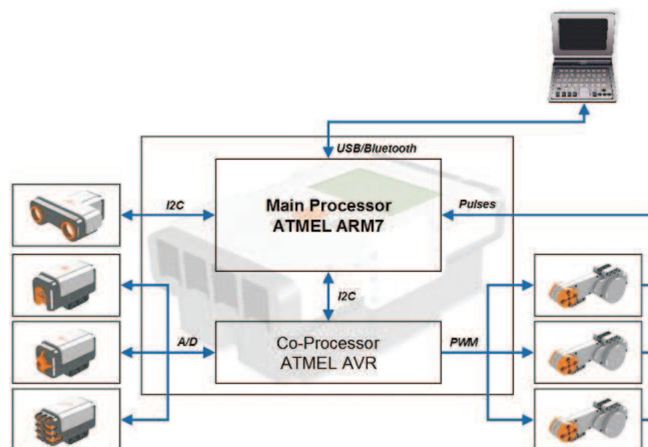


Figura 4.3: Interfaces do Robô NXT

Na implementação do ExecutorXM no PC foi utilizada uma biblioteca de interface gráfica para a linguagem *Lua* chamada **murgaLua** [murgaLua, 2009]. Esta interface disponibiliza uma *API* para manipulação das janelas gráfica no PC usando a linguagem *Lua*. Para a implementação no hardware do robô foi utilizada uma versão da linguagem

Lua para sistemas embarcados chamada *pbLua* [PbLua, 2009]. Com isso o mesmo algoritmo do *ExecutorXM* que roda no PC também roda no robô NXT sem alteração.

4.3.1 O Robô NXT

A utilização do Robô NXT, mostrado na Figura 4.4, foi definida para a validação da execução do interpretador de aplicações no formato tabular num ambiente de hardware real. Com ele é possível isolar somente a *X-machine* que trata da aplicação principal, ou seja, o núcleo do sistema reativo, fazendo uso dos sensores do robô NXT. Neste caso os sensores de toque, de movimento, de som e de luminosidade, serão a máquina de entrada para a aplicação e a máquina do comunicador será simulada pelos atuadores do robô, neste caso os motores de movimento e o Display. O Hardware do robô lego é composto por um microcontrolador de 32 bits ARM7 (AT91SAM7S256), com 256 Kb de FLASH e 64 Kb de RAM rodando a 48 MHz. Além de um microcontrolador de 8 bits AVR com 4 Kbytes de FLASH e 512 Byte RAM usado exclusivamente pelo processador principal para controlar o motor de passo dos motores de movimento. Possui 4 portas de entrada para receber os sensores e 3 portas de saída para controlar os motores de movimento. Possui também um display gráfico LCD de 100 x 64 píxels, e quatro botões que podem ser capturados como entrada.

Figura 4.4: Sensores e Atuadores do Robô NXT

4.3.2 A Linguagem LUA

A linguagem *Lua* foi escolhida por ser uma linguagem *script* pequena, com capacidade de ser executada diretamente dentro de um hardware pequeno, como no caso de um sistema embarcado ou de um nó sensor. Ela suporta programação procedural em geral e possui facilidades de descrição de dados. Além de ser dinamicamente tipada,

possui escopo léxico e interpreta *bytecodes*. Possui gerenciamento automático de coleta de lixo na memória. Lua é implementada como uma pequena biblioteca de funções na linguagem ANSI C, e pode ser compilada virtualmente sem qualquer modificação em todas as plataformas de hardware disponível. Por isso o seu núcleo de execução cabe diretamente em hardwares menores, criando assim um ambiente de execução para as aplicações. Nas implementações foram testada duas versões da linguagem *Lua*, juntamente com o versão oficial *Lua 5.1*, a qual está disponível em [Lua, 2009]. A segunda é uma versão que foi reescrita para executar nos Robôs MindStorm NXT da Lego chamada *pbLua* [PbLua, 2009]. A versão utilizada para o experimento com o robô Lego NXT foi a "*pbLua - Beta 18a*". A principal diferença entre pbLua e a versão oficial da linguagem *Lua* são os comandos para leitura e escrita de arquivos, além dos comandos de acesso aos sensores do Robô, os quais não fazem parte da versão oficial. Uma outra versão testada foi a *murgaLua*, que é a linguagem *Lua* com a inclusão de *API* gráficas para manipulação de janelas tanto no ambiente *Windows* como no *Linux*.

4.3.3 Implementação do Executor em LUA

O programa completo do *ExecutorXM* escrito na linguagem *Lua* para executar o Modelo *Bare* é composto por duas funções principais e outras funções de fazem a interface do hardware do robô com ambiente exterior:

Executor é a função principal (Figura 4.5). Ela recebe um arquivo com a aplicação no formato tabular, inicia sempre o processamento pelo estado "init"e executa a aplicação até que o estado de parada ("halt") seja alcançado.

LeCorr recebe como entrada um estado corrente e devolve todas as linhas da aplicação que possuem como estado fonte (*source*) o estado corrente informado (Figura 4.6).

A função *Executor* recebe a aplicação num formato de tabela interna e executa a aplicação seguindo a ordem do algoritmo. Todas as aplicações tem como estado inicial o estado "init". O processamento inicia com a busca das linhas que formam o estado corrente na tabela da aplicação, usando a função "LeCorr". Em seguida a entrada (Input) é "executada". A execução ocorre pela composição das chamadas de duas funções da linguagem *Lua*: *assert*() e *loadstring*(). A função *loadstring* recebe uma cadeia de caracteres como entrada e devolve como resultado a interpretação da cadeia no formato de *bytecodes* Lua. A função *assert* recebe um trecho de código Lua e o executa no mesmo contexto de execução do *ExecutorXM*. Desta forma cada parte da linha da tabela que for submetida para as funções será transformada em *bytecode* Lua

```
1 function Executor()
2         corrente={}
3         corr = "init"
4         while corr ~= "halt"do
5                 corrente=LeCorr(corr)
6                 assert(loadstring (corrente[1][2]) )()
7                 for i,lCorr in pairs(corrente) do
9                         assert(loadstring(lCorr[3]))()
10                          val1 = tostring(lCorr[5])
11                          val2 = tostring(lCorr[4])
12                          local exp = "return ".. lCorr[5]
13                          local Fnc = loadstring(exp)
14                          if (Fnc() ~= false) then
16                                  assert(loadstring(lCorr[7]))()
17                                  assert(loadstring(lCorr[6]))()
18                                  corr = lCorr[4]
19                                  break
20                          end
21                  end
22                  ClearEvent()
23          end – while
24 end – function
```

Figura 4.5: Função *ExecutorXM*

```
function LeCorr(c)
        local cor = {}
                for _, t in ipairs(aplicacao) do
                if t[1] == c then table.insert(cor, t) end
                end
        return cor
end
```

Figura 4.6: Função *LeCorr*

e executado. Para capturar os eventos de entrada, cada linha da tabela da aplicação tem como texto uma chamada para a função “WaitEvent()” (Figua 4.4). A função é ativada e executada pelo ambiente. Com a ativação da função, a captura dos eventos é disparada e a máquina fica aguardando pela sinalização de algum evento. Quando um evento ocorre, ele é capturado e o próximo passo é verificar qual a condição das linhas do estado corrente que será habilitada. Apenas uma condição será habilitada para o evento capturado. A memória de entrada é executada para capturar os elementos utilizados na avaliação das condições. As condições das linhas que formam o estado corrente são avaliadas sequencialmente. A primeira condição verdadeira habilita a execução

da saída (Output) e da memória de saída (Mem_Output), com isso, os valores das variáveis internas são atualizados e próximo estado corrente é definido. Caso nenhuma condição seja avaliada como verdadeira, o evento é descartado e todo o processo é iniciado novamente. O laço principal é repetido até que o estado corrente "halt" seja alcançado.

As funções que realizam a interface para a captura dos eventos e a atuação das respostas que formam o *ExecutorXM* são apresentadas a seguir, os respectivos códigos estão no Anexo A:

WaitEvent - Esta função de interface de entrada que recebe como parâmetro a indicação de qual evento deverá ser monitorado, dentre os eventos possíveis de serem monitorados pelo robô NXT. Nesta implementação os seguintes eventos poderão ser monitorados: evento sonoro, evento de detecção de presença, evento de seleção por toque, evento de temporização. Quando o evento monitorado ocorre a função devolve a indicação da ocorrência do evento pela atribuição do valor lógico "true" na tabela de eventos (PrxMem) e um valor numérico correspondente a cada evento é devolvido na variável *evento*. O valor atribuído para cada evento é o seguinte: mSom = 5, mPresenca=9 , mToque=3, mTempo=10. A função $WaitEvent()$ usa uma função auxiliar chamada $criaCorotina()$, essa função é chamada para criar uma corotina para tratar cada um dos evento monitorado. Um escalonamento circular é realizado para permitir que os eventos possam ser monitorados por meio de um esquema simples de pseudo-paralelismo.

Scan - Esta é uma função de interface de entrada que é ativada para realizar a detecção de presença de objetos na rota do robô. Ela executa uma varredura com incremento de 45 graus para esquerda e depois para direita, o objetivo é encontrar uma posição para desviar do obstáculo. Se houve uma rota de desvio o seu valor é devolvido em graus para ser tomado em seguida.

Move - Esta é uma função de interface de atuação, ela recebe como parâmetros a direção e o número de giros para movimentar as rodas do robô, caso o movimento seja para frente ou para trás. Se o movimento for para um dos lados, o segundo parâmetro é interpretado como graus para posicionar o robô. Os valores dos parâmetros são 0 - para frente, 1 - para trás, 3 - para os lados. O segundo parâmetro depende do valor do primeiro, se for para frente ou para trás, o segundo parâmetro significa numero de voltas das rodas. Se for para os lados e o segundo parâmetro for positivo o movimento é para direita, se for negativo o movimento é para esquerda, em grau de posicionamento da direção do robô.

4.3.4 Gerador de Aplicações *Bare*

Para a geração do modelo tabular de execução a partir da especificação da aplicação, foi desenvolvida uma interface de alto nível chamada **GeradorXM**, assim como o ExecutorXM, esta interface gráfica também foi desenvolvida em *murgaLua*. Para iniciar o *GeradorXM* é necesário fazer uma chamada ao ambiente murgaLua, passando a aplicação do gerador como parâmetro: "murgaLua.exe GeradorXM.lua". O *GeradorXM* é carregado e a tela inicial é apresentada na Figura 4.7 (a). Nesta figura a aplicação Blink já foi carregada, e o detalhe da transição $t4$ é mostrado.

O GeradorXM é composto por quatro telas, que agregam os passos para o desenvolvimento das aplicações. A ferramenta é utilizada a partir da especificação da X-machine para aplicação. Nesta fase do desenvolvimento não existe uma ordem fixa a ser seguida no processo de preenchimento dos detalhes da aplicação. Na tela inicial as seguintes tarefas podem ser realizadas:

- incluir ou excluir estados para a aplicação, sendo que os estados init, start e halt são incluídos pela ferramenta como parte da interface.
- carregar uma aplicação já desenvolvida ou salvar a aplicação corrente.
- verificar os detalhes das transição da aplicação, a transição do estado "init" para o estado "start" já está definida.
- gerar o modelo do sistema para a ferramenta NuSMV.
- gerar a aplicação no modelo tabular para executar no PC ou no robô NXT.
- avançar para a próxima tela para cadastrar as novas transições para a aplicação.

Para inserir uma nova transição deve ser chamada a próxima tela para cadastro de transição, que é mostrada na Figura 4.7 (b)

Na tela de cadastro transições podem ser incluídas ou excluídas novas transições entre os estados cadastrados. Os detalhes de cada função de avaliação de transição também podemos ser verificados. A Figura 4.7 (b), mostra detalhes da função de avaliação de transição de estados $f3$. Nesta tela é possível fazer a ligação entre os estados fontes e estados destinos por meio das funções de avaliação de transição já cadastradas. Para cadastrar novas funções de avaliação de transição de estados a tela de cadastro de função deve ser chamada, esta tela é apresentada na Figura 4.7 (c).

Cada função f é composta de dois elementos de entrada, que são os eventos e os elemento da memória de entrada, e produz como resultado dois elementos de saída, que

(a) Tela Principal (b) Cadastro da Transição dos Estados

(c) Cadastro das Funções de Transição (d) Cadastro das Condições e Eventos

Figura 4.7: Gerador de Aplicações *Bare*

são os eventos disparados ou mensagem, e novos elementos para a memória de saída. Essa transformação é controlada por uma condição de avaliação de transição, que está vinculada a um evento. Na tela da Figura 4.7 (c) podemos inserir ou remover novas funções f que ligam os elementos de entrada aos elementos de saída, vinculando-os a uma condição previamente cadastrada, estando esta ligada a um evento. Também podem ser incluídos novos elementos de memória de entrada ou saída. Os elementos de saída podem ser eventos que estão ligados aos atuadores de movimentos do Robô NXT ou uma mensagem para informação.

Caso ocorra a necessidade da construção de novas condições de avaliação de transição, ou a vinculação da condição a um evento diferente é necessário acessar o cadastro de condições por meio da tela mostrada na Figura 4.7(d). Nesta tela podem ser cadastrados novos eventos, os quais podem estar ligados a qualquer variável do sistema, seja ela monitorada, controlada ou interna, e podem ser feitos relacionamentos entre os elementos da memória por meio dos operadores relacionais. Toda condição deve obrigatoriamente estar vinculada a um evento, com excessão da condição "true", explicada anteriormente. Existem casos em que a condição é o próprio evento, então a opção "none" deve ser escolhida juntamente com um evento.

O processo de construção da aplicação por meio do GeradorXM é intuitivo e interativo, depois que a aplicação foi especificada no Modelo de Desenvolvimento *Bare* e pode ser feito por fases, até que toda a aplicação esteja concluída. Como último passo, a aplicação deve ser gerada para o formato tabular para, em seguida, ser executada pelo ExecutorXM. A aplicação pode ser gerada para ser executara no PC ou no robô NXT. A diferença principal entre as aplicações geradas são os comandos de tratamento de arquivos e os mecanismos de interação com os sensores e atuadores.

Na Tabela 4.4 é apresentado o trecho inicial da aplicação gerada para o "Blink". Este trecho mostra o resultado da geração do estado "init" para o "start" e do "start" para "ledsOn". Mostra também a inicialização que é realizada para construir a tabela de eventos ($Event$), a tabela de memória (Mem), bem como a definição da função de limpeza dos eventos ($ClearEvent$) e a tabela de procuração (*proxy*) para a memória ($PrxMem$). A tabela *PrxMem* é uma meta-tabela que tem a finalidade de ligar os valores da tabela de memória simultaneamente com os elementos da tabela de eventos. Com isso todas as vezes que ocorrer alguma alteração em uma variável na memória, um evento correspondente à variável será disparado na tabela de eventos. Esses detalhes de ligação entre tabelas independentes facilita a integração e sincronização dos eventos durante a execução da aplicação. A meta-tabela é uma tabela *Lua* comum que define o comportamento do valor original com relação a certas operações especiais.

```
[[$ init $
$Event={}; ClearEvent = function() Event={mTempo=false} end $
$Mem = { mTempo=false,iCorr=0,iTotal=5 }$
$start$
$true$
$PrxMem={}; mt = { __index = function (PrxMem,k) return Mem[k] end,$
$__newindex = function (PrxMem,k,v) Event[k]=true; Mem[k] = v end }; $
$setmetatable(PrxMem, mt)$
$Output=nil$
]]
[[$start$
$Input=WaitEvent()$
$MemIn=PrxMem.iCorr$
$ledsOn$
$PrxMem.iCorr <= PrxMem.iTotal and Event.mTempo$
$PrxMem.iCorr=PrxMem.iCorr+1$
$outputWindow:value('Tempo Corrente!')$
]]
```

Tabela 4.4: Trecho Inicial da Aplicação Blink

Na Tabela 4.5 são mostrados os demais estados que compõem a aplicação "Blink" gerada. Cada linha da tabela é definida como uma cadeia de caracteres onde os elementos internos são separados pelo caractere "$". Os elementos constituintes de cada linha são os seguintes, em ordem de disposição:

- Estado Inicial - $Source$
- Entrada da máquina - $Input$
- Entrada da memória - Mem_input
- Condição de Avaliação ligada a um evento - $C_i \wedge E$
- Estado Final - $Target$
- Saída da máquina - $Output$
- Saída para a memória - Mem_output

Para o estado "Halt" todos os elementos são nulos.

```
[[$ledsOn$
$Input=WaitEvent()$
$MemIn=PrxMem.iCorr$
$ledsOff$
$PrxMem.iCorr <= PrxMem.iTotal and Event.mTempo$
$PrxMem.iCorr = PrxMem.iCorr+1 $
$outputWindow:value('Tempo Corrente!')$
]]
[[$ledsOff$
$Input=WaitEvent() $
$MemIn=PrxMem.iCorr $
$ledsOn$
$PrxMem.iCorr <= PrxMem.iTotal and Event.mTempo$
$PrxMem.iCorr=PrxMem.iCorr+1 $
$outputWindow:value('Tempo Corrente!')$
]]
[[$ledsOn$
$Input=WaitEvent()$
$MemIn=PrxMem.iCorr$
$halt$
$PrxMem.iCorr > PrxMem.iTotal and Event.mTempo$
$PrxMem.iCorr=PrxMem.iCorr+1$
$outputWindow:value('Tempo Esgotado!')$ ]]
[[$ledsOff$
$Input=WaitEvent()$
$MemIn=PrxMem.iCorr$
$halt$
$PrxMem.iCorr > PrxMem.iTotal and Event.mTempo$
$PrxMem.iCorr=PrxMem.iCorr+1$
$outputWindow:value('Tempo Esgotado!')$ ]]
[[$halt$ $nil$ $nil$ $halt$ $false$ $nil$ $nil$]]
```

Tabela 4.5: Continuação da Tabela para a Aplicação Blink

Capítulo 5

Aplicações do Modelo *Bare*

Como definido no Capítulo 3, os sistemas reativos são unidades programáveis e autônomas que executam a aplicação para atingir um objetivo. Assim, para modelar uma aplicação em RSSF, o nó sensor é considerado como um sistema reativo autônomo para desempenhar um comportamento reativo na interação com o meio ambiente. Na modelagem desse comportamento reativo num ambiente dinâmico os elementos identificados no processo são os seguintes:

- o conjunto de estímulo ambiental ou entradas.
- o conjunto de estado internos ao nó.
- o conjunto de regras que relacionam os estímulos com as mudanças de estados internos.

No desenvolvimento das aplicações em RSSF usando o Modelo de *Bare*, a aplicação é dividida em três componentes: um componente sensor, um componente aplicação e um componente comunicador/atuador. Cada componente é, em essência, uma *X-machine* independente e, como tal, pode ser modelada de forma individual, necessitando somente de um mecanismo de comunicação entre os três componentes. Depois de modelada, a aplicação é transformada no formato tabular e depositada no nó sensores, onde ela será executada pelo *ExecutorXM*, previamente carregado no nó. O *ExecutorXM* é responsável por interpretar a tabela da aplicação e executar as instruções no formato tabular. O formato tabular executável para a aplicação traz os seguintes benefícios:

- Simplificação do mecanismo de execução das aplicações em RSSF, pois o único código executável é o do *ExecutorXM*, o qual será depositado no nó sensor previamente e não necessita receber qualquer alteração para a execução das aplicações.

A aplicação, por sua vez, poderá ser enviada para ser executada somente quando for necessário ou poderá receber alterações durante a sua execução, que é chamada de reconfiguração dinâmica da aplicação.

- A programação e, consequentemente a reprogramação, serão simplificadas devido ao formato tabular das aplicações, sendo possível enviar a aplicação no formato de uma tabela completa de uma única vez ou pelo envio das linhas da tabela em separado. Isso permite um uso mais racional da baixa taxa de transferência entre os nós. Essa característica é conhecida como reconfiguração dinâmica da aplicação com contexto parcialmente reconfigurável.

- O papel de alguns nós sensores pode ser especializado, por meio do envio de linhas extras para a aplicação que produzirá um comportamento diferente. Além de possibilitar a migração desses papeis para outros nós, apenas pelo envio das partes especializadas da aplicação.

- É possível armazenar várias aplicações pré-definidas no nó, de tal maneira que estas serão ativadas em resposta a situações específica, os chamados cenários de aplicações [Caldas et al., 2005b], permitindo que o nó possa ser reconfigurado totalmente para atender a situações não previstas ou para se recuperar de situações de erros. Essa característica é conhecida como reconfiguração dinâmica com contexto múltiplo parcialmente reconfigurável.

Em seguida são apresentados 3 (três) aplicações desenvolvidas utilizando o Modelo de Desenvolvimento *Bare*. A primeira é uma aplicação de detecção de intruso, esta aplicação é direcionada para a área de Redes de Sensores Sem Fios e tem como objetivo mostrar a viabilidade da programação de uma aplicação em RSSF por meio de um sistemas reativo, utilizando um hardware real. A segunda aplicação é a aplicação de caminhamento autônomo de um robô, que tem por objetivo aplicar o Modelo *Bare* no desenvolvimento de um sistema reativo completo, utilizando um hardware real e completando, assim, o teste final de conceito para o Modelo proposto. A terceira aplicação é a união da área de robótica com RSSF, onde é proposto uma aplicação para realizar o posicionamento inteligente de nós sensores sem fio utilizando robôs autônomos. Esta aplicação foi desenvolvida em parte devido a problemas de hardware. Com estas 3 aplicações é pretendido mostrar a viabilidade da utilização do Modelo de Desenvolvimento *Bare* na construção de aplicações práticas.

5.1 Detecção de Intruso

Como exemplo de aplicação geral em RSSF, foi utilizado como estudo de caso uma aplicação encontrada na literatura chamada *envirotrack* [Abdelzaher et al., 2004]. Envirotrack é um *framework* para detecção e acompanhamento de alvos móveis numa rede de sensores. Baseado na aplicação *envirotrack* foi feito um modelo para a aplicação de detecção de intruso. No modelo de detecção de intruso um nó sensor pode estar em um dos seguintes estados: *livre*, *seguidor*, *membro* ou *líder*. Um nó, inicialmente, está no estado *livre*, neste caso é um nó que não detectou nenhum alvo. Um nó *livre* passa a ser um nó *membro* quando detecta a aproximação de um alvo por meio do sensor de presença. Um nó *livre* torna-se um nó *seguidor* quando não detecta nenhum alvo, mas está na vizinhança de um nó membro e recebe um aviso (*heartbeat*) que um alvo foi detectado. No estado livre o nó sensor não responde aos eventos de temporização e eleição de líder. Um nó *líder* é um nó que estava no estado *membro* e foi eleito por uma definição direta, sem a ocorrência de uma eleição. Todos os nós *membros* enviam sua localização para o nó *líder*, que realiza uma fusão das posições para obter uma estimativa da posição do alvo detectado. Se o nó *líder* deixar de detectar o intruso ele passa para o estado de *seguidor* e outro nó *membro* deve ser eleito *líder*. Os nós *membros* enviam sinais específicos (*heartbeat*) para que os nós *livres* que estão na sua vizinhança tornem-se *seguidores*. Os nós *seguidores* possuem um temporizador e, caso não detectem um alvo neste intervalo, voltam a ser nós *livres*. A Figura 5.1 mostra o esquema descrito para a máquina de *detecção de intruso*. Neste modelo, a ocorrência de alguns eventos não modificam a situação do nó sensor em determinados estados, isto é, caso ocorra um evento não tratado pelo estado corrente o evento é simplesmente descartado.

5.1.1 Detecção de Intruso no Modelo *Bare*

O primeiro passo é descrição da aplicação de *detecção de intruso* por meio da *X-machine* estendida pelo Modelo *Bare*. Depois de modelada a aplicação tem a seguinte definição:

- $T = (Booleano = \{\ true,\ false\ \})$
- $\Sigma = (mSom = \{Booleano\},\ mIntruso = \{Booleano\},\ mEleito = \{Booleano\},$ $mTempo = \{Booleano\},\ mPosXY = \{Int, Int\})$
- $\Gamma = (cSom = \{Booleano\},\ cPosXY = \{Int, Int\})$
- $\Lambda = (\ iPosXY = \{Int, Int\},\ iCorr = \{Tempo\},\ iTotal = \{Tempo\})$

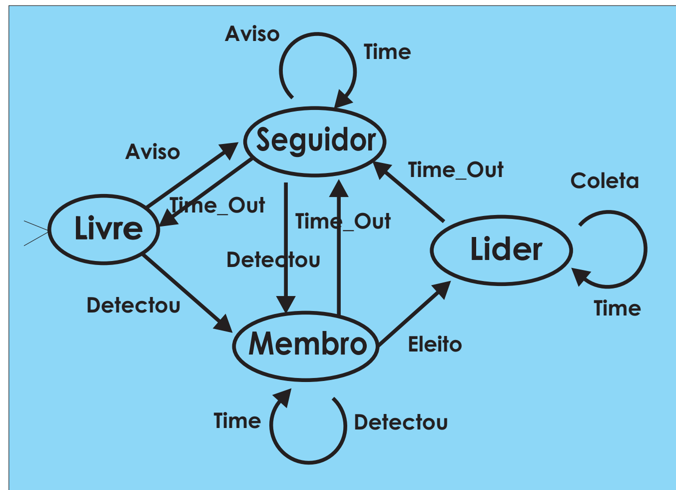


Figura 5.1: Aplicação de Detecção de Intruso

- $Q \cup \{I, S, H\} = \{Livre,\ Seguidor,\ Membro,\ Lider,\ Init,\ Start,\ Halt\}$
- $M = (mSom,\ mIntruso,\ mEleito,\ mPosXY,\ mTempo,\ iCorr,\ iTotal,\ iPosXY,\ cSom,\ cPosXY)$
- $E = \{e_1 = mSom\ \vee\ e_2 = mIntruso\ \vee\ e_3 = mEleito\ \vee e_4 = mTempo\}$ $= \{e_1 = false \vee e_2 = false \vee e_3 = false \vee e_4 = false\}$
- $C = \{$
  - $[c_0 = true]$
  - $[c_1 = e_1]$
  - $[c_2 = e_2]$
  - $[c_3 = e_3]$
  - $[c_4 = iCorr > iTotal \wedge e_4]$
  - $[c_5 = iCorr <= iTotal \wedge e_4]$

  $\}$
- Φ
  - $\phi_0 = (_, _, c_0, _, _)$
  - $\phi_1 = (mSom,\ iCorr,\ c_1,\ iCorr = 0,\ "Recebeu\ Aviso")$
  - $\phi_2 = (mIntruso,\ iCorr,\ c_2,\ iCorr = 0,\ "Detectou\ Intruso")$
  - $\phi_3 = (mEleito,\ iCorr,\ c_3,\ iCorr = 0,\ "Eleito\ Lider")$

  - $\phi_4 = (mTempo,\ _,\ c_4,\ _,\ "Tempo\ Esgotado")$
  - $\phi_5 = (mTempo,\ (iCorr, iTotal),\ c_5,\ iCorr = iCorr + 1,\ "Coletando")$
  - $\phi_7 = (mEleito,\ _,\ c_3,\ _,\ "Finalizando")$

- F
  - $(Init, \phi_0, Start)$
  - $(Start, \phi_0, Livre)$
  - $(Livre, \phi_1, Seguidor)$
  - $(Livre, \phi_2, Membro)$
  - $(Seguidor, \phi_1, Seguidor)$
  - $(Seguidor, \phi_2, Membro)$
  - $(Seguidor, \phi_4, Livre)$
  - $(Seguidor, \phi_5, Seguidor)$
  - $(Membro, \phi_2, Membro)$
  - $(Membro, \phi_3, Lider)$
  - $(Membro, \phi_5, Membro)$
  - $(Membro, \phi_4, Seguidor)$
  - $(Lider, \phi_4, Seguidor)$
  - $(Lider, \phi_5, Lider)$
  - $(Lider, \phi_7, Halt)$
- $q_0 = \{Init\}$
- $m_0 = (mSom = false,\ mIntruso = false,\ mEleito = false,$
  $mPosXY = 0,\ mTempo = false,\ iCorr = 0,\ iTotal = 10,$
  $iPosXY = 0,\ cPosXY = 0,\ cSom = false)$

5.1.2 Gerador da Aplicação de Detecção de Intruso

Para a geração do modelo tabular de execução a partir da especificação da aplicação é utilizado o **GeradorXM**, conforme descrito na seção 4.3.4. O *GeradorXM* é carregado e a tela está apresentada na Figura 5.2(a). Os estados da aplicação são incluídos sendo que os estados "init", "start" e "halt" já vem inclusos. Após o cadastramento dos estados é necessário fazer a inclusão das transições entre os estados. Uma transição é

a ligação entre um estado fonte e um estado alvo por meio de uma função de transição de estados. A função de transição com a condição sempre verdadeira (true) já vem cadastrada como f[1], bem como a transição entre o estado "init" e o estado "start". A Figura 5.2(b) mostra o resultado do cadastro da transição entre o estado "start" e o estado "livre", que utiliza a função f[1]. Para as transições entre os outros estados, novas funções de transição devem ser cadastradas, isso é feito por meio da próxima tela de cadastro de função.

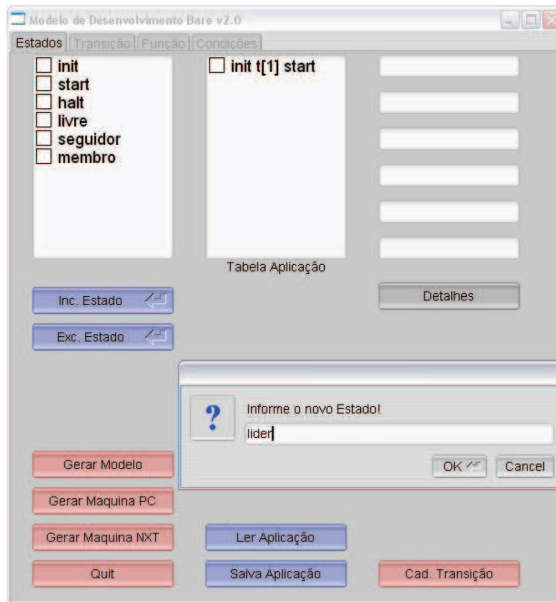
As funções são formadas pelos eventos detectados e um elemento da memória de entrada, produzindo como resultado novos eventos ou mensagens e um elemento da memória de saída. Cada função é controlada por uma **condição de avaliação de transição**, que é vinculada a detecção de um evento. Na tela da Figura 5.2(c) podem ser criadas novas funções com os eventos de entrada e de saída previamente definidos pelo modelo e podem ser inseridos novos elementos de memória interna. A condição "true" é uma condição previamente cadastrada, que não está ligada a ocorrência de nenhum evento. Para criar uma nova função de transição é necessário apenas indicar os elementos de entrada e de saída. A Figura 5.2(c) mostra todas as funções utilizadas para a aplicação de detecção de intruso.

Cada **condição** que é utilizada pelas funções de transição é formada por uma expressão lógica dos elementos de memória usando os operadores relacionais e está vinculada a um evento detectado. Na tela da Figura 5.2(d) podem ser cadastrados novos eventos para a aplicação que podem estar ligados a qualquer variável do sistema. Toda condição deve estar vinculada a um evento, com exceção da condição "true" e existem casos em que a condição é somente o próprio evento.

O processo de construção da aplicação por meio do GeradorXM é facilitado depois que a aplicação foi especificada no Modelo de Desenvolvimento *Bare* e pode ser feito por fases, até que toda a aplicação esteja concluída. Depois de cadastrada a aplicação um "Modelo do Sistema" é gerado automaticamente para ser verificado usando a ferramenta NuSMV. Depois de verificada as propriedades da aplicação, o próximo passo é gerar a tabela da aplicação para ser executada pelo ExecutorXM no PC ou no robô NXT.

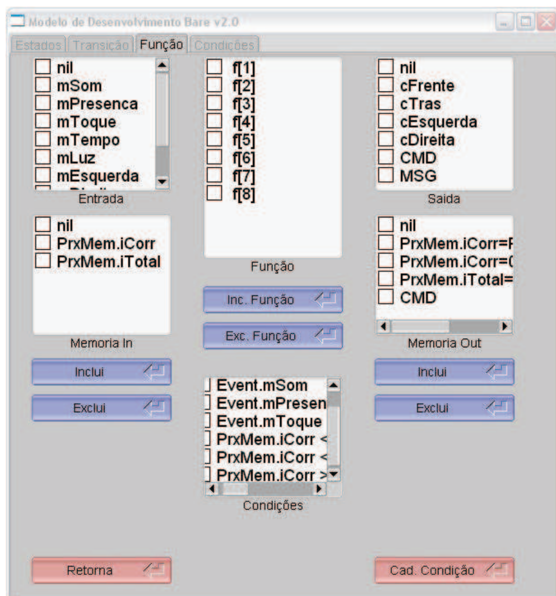
5.1.3 Mapeamento para o Modelo Tabular

Depois de concluída a descrição da aplicação utilizando o GeradorXM é feita a transformação automática para o modelo de execução no formato tabular. Na Seção 3.3 é descrito o algoritmo do mapeamento para o Modelo Tabular. A tabela resultante para a aplicação de *detecção de intruso* é mostrada na Figura 5.1. Nela estão resumidos todos os componentes necessários para a execução da aplicação.

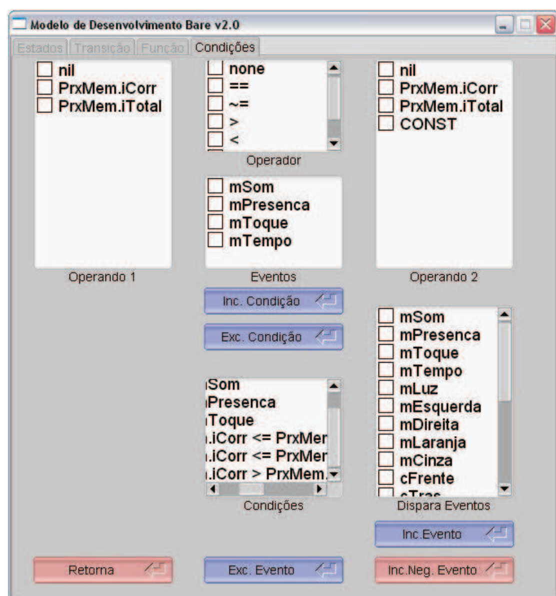


(a) Inclusão de Estados

(b) Cadastro da Transição dos Estados



(c) Cadastro das Funções de Transição



(d) Cadastro das Condições e Eventos

Figura 5.2: Geração da Aplicação de Detecção de Intruso

Source	**Input**	**Mem_input**	**Target**	**Condition**	**Mem_output**	**Output**
init	nil	nil	start	$true$	nil	*"Init"*
start	nil	nil	livre	$true$	$iCorr = 0$, $iTotal = 10$	*"Start"*
livre	$mSom \vee mIntruso$	$iCorr$	seguidor	c_1	$iCorr = 0$	*"Recebeu Aviso"*
livre	$mSom \vee mIntruso$	$iCorr$	membro	c_2	$iCorr = 0$	*"Detectou Intruso"*
seguidor	$mSom \vee mIntruso \vee mTempo$	$iCorr$	seguidor	c_1	$iCorr = 0$	*"Recebeu Aviso"*
seguidor	$mSom \vee mIntruso \vee mTempo$	$iCorr$	membro	c_2	$iCorr = 0$	*"Detectou Intruso"*
seguidor	$mSom \vee mIntruso \vee mTempo$	$iCorr, iTotal$	livre	c_4	$iCorr = 0$	*"Tempo Esgotado"*
membro	$mIntruso \vee mEleito \vee mTempo$	$iCorr$	membro	c_2	$iCorr = 0$	*"Detectou Intruso"*
membro	$mIntruso \vee mEleito \vee mTempo$	$iCorr$	lider	c_3	$Corr = 0$	*"Eleito Lider"*
membro	$mIntruso \vee mEleito \vee mTempo$	$iCorr, iTotal$	membro	c_5	$iCorr = iCorr + 1$	*"Temporizador"*
membro	$mIntruso \vee mEleito \vee mTempo$	$iCorr, iTotal$	seguidor	c_4	$iCorr = 0$	*"Tempo Esgotado"*
lider	$mTempo$	$iCorr, iTotal$	seguidor	c_4	$iCorr = 0$	*"Tempo Esgotado"*
lider	$mTempo$	$iCorr, iTotal$	lider	c_5	$iCorr = iCorr + 1$	*"Coletando"*
lider	$mToque$	nil	Halt	c_6	nil	*"Finalizando"*
halt	nil	nil	halt	nil	nil	nil

Tabela 5.1: Tabela para a Aplicação Detecção de Intruso

As Tabelas 5.3a e 5.3b apresentam, respectivamente, a relação dos eventos monitorados pela aplicação e as expressões que formam as condições relacionadas com os eventos, tais condições serão avaliadas apenas na ocorrência do evento.

Memory	Events	Value
mSom	e_1	false
mIntruso	e_2	false
mEleito	e_3	false
mTempo	e_4	false
	$\bigvee E$	false

(a) Eventos

Conditions	Expression	Value
c_1	$true \wedge e_1$	false
c_2	$true \wedge e_2$	false
c_3	$true \wedge e_3$	false
c_4	$iCorr > iTotal \wedge e_4$	false
c_5	$iCorr <= iTotal \wedge e_4$	false

(b) Condiçoes

Figura 5.3: Tabelas Auxiliares para Detecção de Intruso

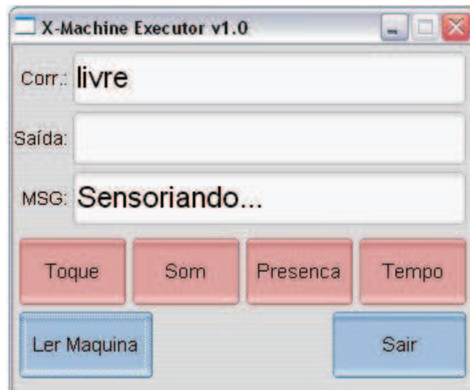
5.1.4 Execução da Aplicação no PC

Para a execução da aplicação no ambiente PC foi construído um simulador de sistemas reativos chamado **ExecutorXM**, onde os eventos monitorados são acionados por botões que indicam a ocorrência do respectivo evento e a saída é apresentada no formato de mensagens. Para a construção do simulador foi utilizada a biblioteca de interface gráfica para a linguagem *Lua* chamada *murgaLua* [murgaLua, 2009]. A biblioteca disponibiliza uma *API* para manipulação de janelas gráfica usando a linguagem *Lua*. Para iniciar o simulador é necessário fazer uma chamada ao ambiente murgaLua, passando o executor como parâmetro, da seguinte forma: "murgaLua.exe ExecutorXM.lua". O *ExecutorXM* então é carregado e as telas são apresentadas na Figura 5.4.

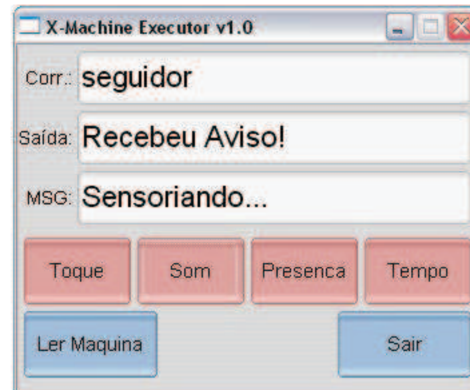
Em seguida a aplicação deve ser carregada pela opção "Ler Maquina". Depois de carregada para a memória, a aplicação inicia a sua execução sempre pelo estado inicial "init", isso é necessário para executar as inicializações das tabelas internas (Mem, PrxMem, Event) e em seguida o estado corrente passa para o estado "start". A partir deste momento o controle da execução da aplicação fica sujeito a ocorrência dos eventos, neste caso a máquina fica esperando a ocorrência dos eventos, como pode ser observado na Figura 5.4 (a).

Para a execução da aplicação, os seguintes eventos serão monitorados como entradas e a interpretação para a ocorrência deles será:

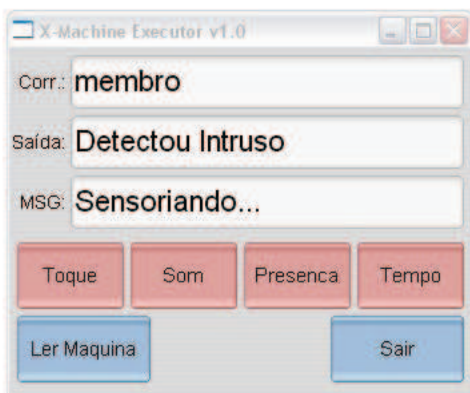
- toque - significa uma eleição para líder do cluster.
- som - significa que um sinal de aviso foi recebido pelo nó sensor.
- presença - significa que um intruso foi detectado.
- tempo - significa que um "tick" de passagem do tempo foi recebido.



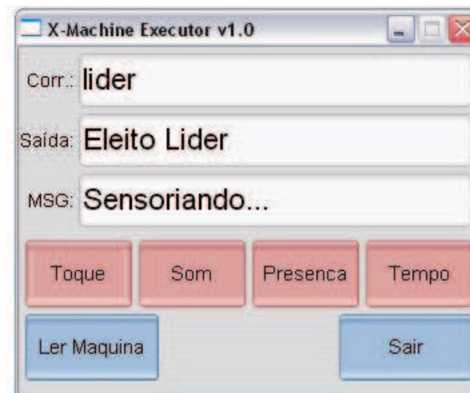
(a) Aplicação de Detecção de Intruso no estado "livre"



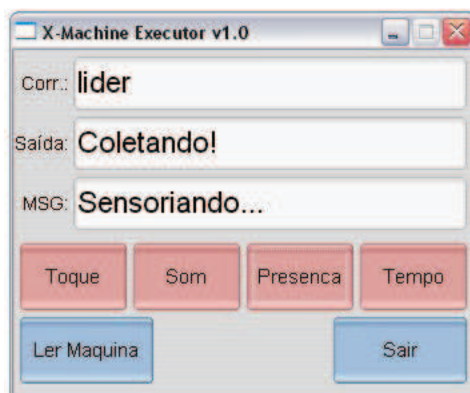
(b) Executor: Recebeu Sinalização



(c) Executor: Detectou Intruso



(d) Executor: Eleição de Líder



(e) Executor: Recebimento de dados

(f) Finalização do Executor

Figura 5.4: Execução da Aplicação de Detecção de Intruso no PC

Como estabelecido pela aplicação de detecção de intruso, quando o nó está no estado "livre" apenas dois eventos são considerados: a sinalização de outros nós sensores (som) e a detecção de intrusos pelo próprio nó (presença). Caso uma sinalização de um outro nó seja recebida o nó sensor passa para o estado de "seguidor" como apresentada na Figura 5.4 (b).

No estado "seguidor" os eventos tratados são: a sinalização dos outros nós sensores (som), a detecção de intrusos (presença) que faz o nó passar para o estado "membro" e o "tick" de tempo (tempo) que após um intervalo definido pela aplicação, se não houver detecção de intruso, faz com que o nó retorne para o estado "livre". Na Figura 5.4 (c) é mostrado apenas a detecção de intruso que faz o nó passar para o estado "membro".

Estando no estado "membro" os eventos tratados são: a detecção de intruso (presença), o "tick" de tempo (tempo) que após um intervalo, se não houver detecção de intruso, faz com que o nó retorne para o estado "seguidor" e o evento de eleição de líder (toque), que faz com que o nó passe para o estado de "líder" do cluster. Na Figura 5.4 (d) é mostrado a sinalização de toque e o nó passa para o estado "lider".

No estado "lider" os eventos tratados são: o "tick" de tempo (tempo) que faz com que o nó armazene os dados enviados pelos demais membros e faz a fusão destes para gerar uma possível posição do intruso, e o evento de eleição de líder (toque), que faz com que o nó transmita os dados da posição do intruso e passe para o estado "halt", finalizando a execução da aplicação. Na Figura 5.4 (e) é apresentado a captura de um eventos de detecção de intruso que demonstra o recebimento das posições dos outros nós.

Em seguida é feito o recebimento de um evento de eleição de lider (toque) para demonstrar a finalização da aplicação, e esta passa para o estado "halt" e encerra, conforme mostra a Figura 5.4 (f).

5.1.5 Execução da Aplicação no Robô

Para executar a aplicação de detecção de intruso no robô NXT foi utilizada uma versão da linguagem *Lua* reescrita para executar nos Robôs NXT da Lego, chamada *pbLua* [PbLua, 2009]. A implementação desta aplicação num hardware real tem por objetivo demonstrar a aplicabilidade do Modelo de Desenvolvimento *Bare* no desenvolvimento aplicações para sistemas embarcados utilizados num ambiente real, onde existem sérias restrições de processamento, de memória e de bateria, como aqueles encontrados nos nós sensores sem fio. Para executar a aplicação no Robô NXT faremos uso das "rotinas de interface", cuja finalidade é fazer a ligação entre o núcleo do sistema reativo e o ambiente exterior, as rotinas de interface são proposta em [Berry & Gonthier, 1992].

Para a detecção dos eventos externos foi projetada uma rotina de interface de entrada que detecta a ocorrência de um evento no mundo real e sinaliza para a aplicação, por meio da atualização das variáveis de eventos na memória. Para a interface de saída foi utilizada a sinalização por meio da informação apresentada no display de LCD. O robô NXT da Lego foi configurado como apresentado na Figura 5.5, nele estão ativos como entrada os sensores de presença, de som, de toque e um temporizador interno e, como saída o display de LCD. Em outro experimento a saída foi configurada para executar sobre os motores que acionam as rodas, além do acionamento do sensor de presença fazendo uma varredura do ambiente.

Figura 5.5: Configuração Robô NXT para os Experimentos

As seguintes rotinas executam no Robô NXT:

- WaitEvent (Figura 5.6) - Essa rotina foi desenvolvida para utilizar os sensores disponíveis no robô NXT, os quais são: sensor de presença, sensor de som e sensor de toque. Ela utiliza, ainda, um temporizador interno cujo objetivo é disparar um sinal a cada intervalo de tempo programado para indicar a passagem de um intervalo de tempo (tique), esse intervalo é definido pela variável *tEspera* no código do ExecutorXM e é definido em segundos. A leitura dos eventos é feita por meio da execução de corotinas para simular uma detecção em paralelo dos eventos. Quando algum desses eventos ocorre ele é sinalizado para a memória da máquina e habilita a avaliação de uma condição. A implementação da função *WaitEvent* em *pbLua* para executar no robô NXT é apresentada na Figura 5.6.

```
function WaitEvent(eSom, ePresenca, eToque, eTempo, ePosXY)
    evento=0
    criaCorotinas()
    while evento==0 and nxt.ButtonRead()  = 1 do
        coroutine.resume(co1, tEspera)
        coroutine.resume(co2, 3)
        coroutine.resume(co4, 4)
        coroutine.resume(co3, 1, 25)
     end
    if evento == 1 then nxt.DisplayText("TQ",60,30)
        elseif evento == 2 then nxt.DisplayText("S ",30,30)
        elseif evento == 9 then nxt.DisplayText("P ",45,30)
        elseif evento == 10 then nxt.DisplayText("T ",80,30)
        else nxt.DisplayText("Sens: ",0,30)
    end
end
```

Figura 5.6: Interface para Detecção de Eventos

- Executor (Figura 5.7)- Essa é a rotina responsável pela execução da aplicação no formato tabular no robô NXT. A função recebe como parâmetro um arquivo que armazena a aplicação e passa a executá-la, a partir do estado inicial (“init”). Ela utiliza a função *LeCorrArq* (Figura 5.8), para coletar a partir do arquivo somente as linhas da aplicação que possuem como estado fonte (source) o estado corrente informado. A partir da execução padrão do estado corrente “init” a evolução da aplicação fica por conta do eventos detectados e das condições habilitadas pelos mesmos, até que o estado final “halt” seja alcançado.

- LeCorrArq (Figura 5.8)- Esta função trabalha de maneira diferente no robô NXT devido a restrição de memória do hardware. Para diminuir a utilização da memória são carregados apenas as linhas do arquivo da aplicação cujo o estado fonte é o estado corrente. Isso deixa a função um pouco mais complexa que a original, mas permite que o espaço de memória seja otimizado. A função recebe como parâmetros o nome do arquivo com a aplicação tabular e o estado corrente e devolve uma tabela com todas as linhas que tem como estado fonte o estado informado:

A aplicação é gerada automaticamente por meio do módulo *GeradorXM*, conforme descrito na Seção 4.3.4. O desenvolvimento da aplicação é independente do ambiente de execução. Uma vez definida a aplicação ela pode ser executada tanto no robô NXT quanto no simulador do PC, chamado *ExecutorXM*, a diferença fica por conta dos detalhes de leitura e escrita dos arquivos, da saída que é direcionada para o display de LCD e das restrições de espaço em memória e das rotinas dedicadas para acesso aos

```
function Executor(Arquivo)
    corr = "init"
    nxt.DisplayClear()
    nxt.DisplayText("ExXM 1.0",0,0)
    nxt.DisplayText("Curr:",1,10)
    nxt.DisplayText("Next:",1,20)
    nxt.DisplayText("Sens: ",1,30)
    while corr  = "halt"
    do
        corrente={}
        corrente=LeCorrArq(Arquivo,corr)
        nxt.DisplayText(,35,10)
        nxt.DisplayText(corr,35,10)
        assert(loadstring(corrente[1][2]))()
        for i,lCorr in pairs(corrente)
        do
            assert(loadstring(lCorr[3]))()
            nxt.DisplayText(,35,20)
            nxt.DisplayText(tostring(lCorr[4]),35,20)
            local exp = "return ".. lCorr[5]
            local Fnc = loadstring(exp)
            if (Fnc()  = false)
            then
                assert(loadstring(lCorr[7]))()
                assert(loadstring(lCorr[6]))()
                corr = lCorr[4]
                break
            end
        end
        ClearEvent()
        collectgarbage()
    end
end
```

Figura 5.7: Rotina do ExecutorXM em pbLua

sensores do robô NXT. A aplicação de detecção de intruso que será executada pelo *ExecutorXM* no robô NXT tem o formato apresentado na Figura 5.9. Esse código é o mesmo apresentado na Seção 4.3.3, ele é transferido no formato de texto para o interpretador *pbLua* via uma conexão USB. Durante a transferência o código é interpretado e executado gerando, com isso, um arquivo interno com a tabela da aplicação. A diferença entre este código específico para o robô NXT e o original é devido ao mecanismo de armazenamento de arquivos no robô NXT que é por meio de memória FLASH de 256 Kbytes. Existe, ainda, restrições no tamanho da memória RAM que é de 64 Kbytes. A RAM é necessária para executar o interpretador *Lua*, o código do *ExecutorXM* e a tabela com o código da aplicação. As memórias FLASH e RAM são utilizadas da seguinte forma:

```
function LeCorrArq(arquivo,corr)
    local a={}
    local b={}
    local line
    corrente={}
    mx = nxt.FileOpen(arquivo, "r")
    if mx ~= nil then
        line = nxt.FileRead(mx,"*l")
        inicio, fim = string.find(line,'%$.-%$')
        estado=string.sub(line,inicio+1,fim-1)
        while line ~= nil do
            b={}
            if estado == corr then
                for w in string.gmatch(line, '(%$.-%$)') do
                    table.insert(b,string.sub(w, 2, -2))
                end
            end
            line = nxt.FileRead(mx,"*l")
            if line ~= nil then
                inicio, fim = string.find(line,'%$.-%$')
                estado=string.sub(line,inicio+1,fim-1)
            end
        end
    end
    nxt.FileClose(mx)
    return a
end
```

Figura 5.8: Rotina para Capturar o Estado Corrente

- A imagem do Interpretador *LUA* ocupa 132 Kbytes da FLASH. Sobram 124 Kbytes da FLASH para o armazenamento dos scripts *Lua* e os dados.
- 4 Kbytes da RAM são utilizados para vários tipos de variáveis internas, buffers e acesso aos drivers.
- 3.5 Kbytes são alocados para a pilha de execução da aplicação.
- A execução do interpretador *Lua* ocupa 16 Kbytes da RAM (16.450K). Sobram 40,5 Kbytes para o *ExecutorXM* executar a tabela com o código da aplicação.
- O código do ExecutorXM carregado ocupa 9,7 Kbytes da RAM. Sobram 30,8 Kbytes para a tabela com a aplicação e a manipulação das variáveis dinâmica do interpretador *Lua*.

Para executar a aplicação no robô NXT, é necessário inicialmente fazer uma conexão via USB ou Bluetooth entre o robô NXT e o PC para estabelecer um canal

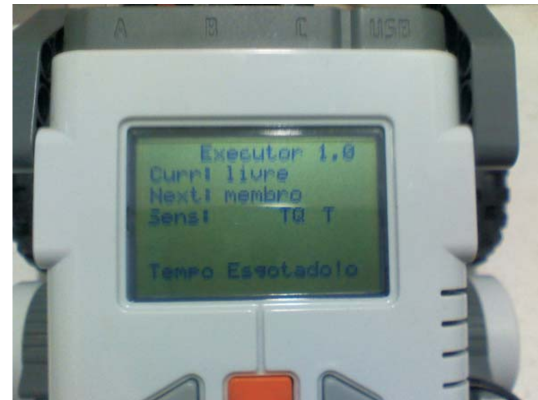
```
mx = nxt.FileCreate( "Intruso.nxt", 5122 )
Str= [[ $init$ $Event= { }; ClearEvent = function()
Event= { mSom=false,mPresenca=false,mToque=false,mTempo=false } end $
$ Mem = { mSom=false,mPresenca=false,mToque=false,mTempo=false,
iCorr=0,iTotal=10 } $
$ start $ $ true $
$ PrxMem= { }; mt = { __index = function (PrxMem,k) return Mem[k] end,
__newindex = function (PrxMem,k,v) Event[k]=true; Mem[k] = v end } ;
setmetatable(PrxMem, mt)$
$ Output=nil $ ]]
Str=string.gsub(Str,"\n", " ")
nxt.FileWrite(mx,Str.."\n")
collectgarbage()

Str=[[ $start$$Input=nil$
$ MemIn=nil$ $livre$
$ true$
$ MemOut=nil$ $ Output=nil$ ]]
Str=string.gsub(Str,"\n"," ")
nxt.FileWrite(mx,Str.."\n")
collectgarbage()

Str=[[$livre$$Input=WaitEvent(true,false,false,false,false)$
$MemIn=PrxMem.iCorr$$seguidor$
$Event.mSom$
$PrxMem.iCorr=0$$nxt.DisplayText('Recebeu Aviso!')$ ]]
Str=string.gsub(Str,"\n"," ")
nxt.FileWrite(mx,Str.."\n")
collectgarbage()
```

Figura 5.9: Tabela com Trecho da Aplicação de Detecção de Intruso

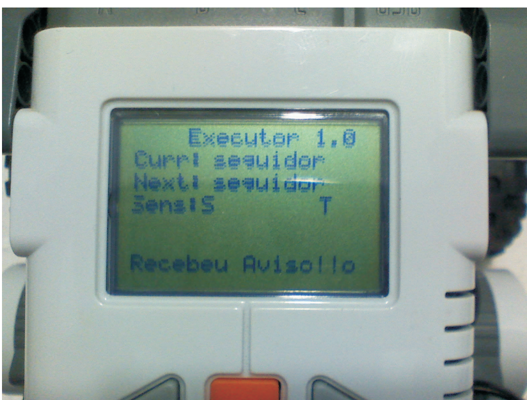
de comunicação. Depois usar algum software com conexão serial como terminal, nesse experimento foi utilizado o *HyperTerminal*. Depois de estabelecida a conexão é feita a transferência do arquivo com o *ExecutorXM*, esse arquivo é transferido uma única vez. Depois deve ser transferido o arquivo gerado pelo *GeradorXM* para o NXT, essa transferência pode ser feita de uma única vez com a aplicação completa, ou pode ser feita por meio do envio das linhas individualmente, conforme discutido na Seção 3.5.

Uma vez que toda a saída do processamento da aplicação é por meio de mensagens no display de LCD, a evolução da execução da aplicação será apresentada por meio das figuras mostrando somente o display de LCD. No display de LCD do robô são apresentados, além das mensagem resultantes da execução da aplicação, algumas informações sobre o estado da aplicação. São apresentados o estado corrente(Curr), o próximo estado (Next), que é definido pela condição que foi habilitada pelo último evento capturado, caso este seja processado pelo estado corrente, caso contrário o evento

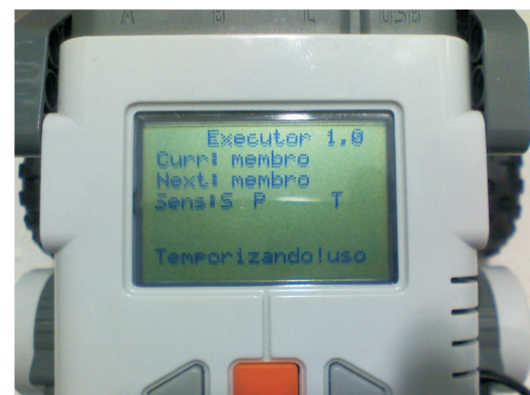
(a) Tela inicial do pbLua no robô NXT



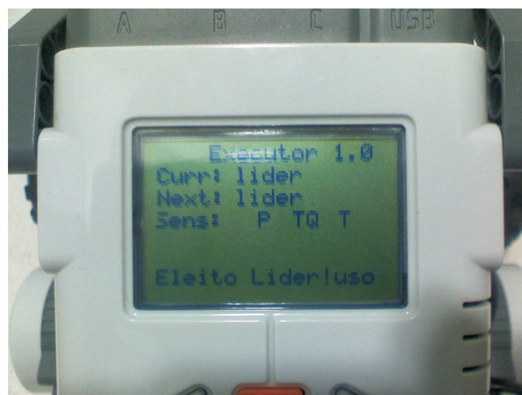
(b) Aplicação de detecção de intruso - livre



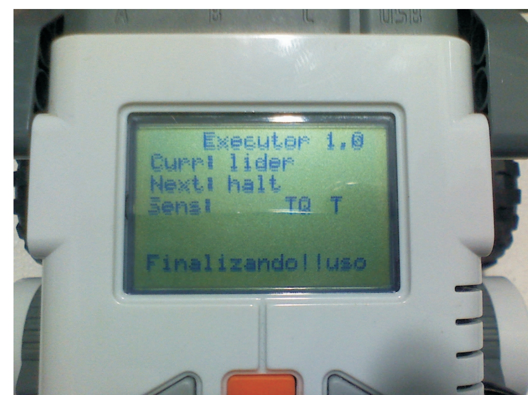
(c) Aplicação de detecção de intruso - seguidor



(d) Aplicação de detecção de intruso - membro



(e) Aplicação de detecção de intruso - lider



(f) Aplicação de detecção de intruso - halt

Figura 5.10: Execução da Aplicação de Detecção de Intruso no Robô NXT

será descartado. É apresentado também o evento sensoriado com a seguinte codificação: S indicando que foi detectado um sinal (som), P indicando que foi detectado um intruso (presença), TQ indicando que foi detectado um toque e T indicando que houve a passagem de um "tick" de tempo.

Na Figura 5.10 (a) é mostrado o display de LCD do robo NXT indicando que o ambiente *pbLua* está pronto para executar. Para iniciar a aplicação deve ser chamada a função *Executor* passando como parâmetro o nome do arquivo com a tabela da aplicação da seguinte forma: "Executor("Intruso.nxt")". A partir daí o *ExecutorXM* passa a ler o estado corrente e a execução da aplicação é direcionada pelos eventos do mundo real que são capturados pelos sensores do robô NXT e pela evolução dos estados internos correntes da máquina. Da mesma forma que a aplicação executa no PC, ela também deve executar no robô NXT. A aplicação inicia no estado "init", executa as inicializações das tabelas internas (Mem, PrxMem, Event) e, em seguida, passa para o estado "start" e depois para o próximo estado que é o estado "livre". A Figura 5.10 (b) mostra a aplicação de detecção de intruso, no estado "livre", aguardando a ocorrência de algum evento.

Estando no estado "livre" dois eventos são considerados: a sinalização de outros robôs (som) e a detecção de intrusos pelo próprio robô (presença). Caso uma sinalização de um outro robô seja recebida, o robô passa então para o estado de "seguidor" como apresentada na Figura 5.10 (c).

No estado "seguidor" os eventos tratados são: a sinalização dos outros robôs (som), a detecção de intrusos (presença) que faz o robô passar para o estado "membro" e o "tick" de tempo (tempo) que após um intervalo definido pela aplicação, se não houver detecção de intruso, faz com que o robô retorne para o estado "livre". Na Figura 5.10 (d) é mostrado apenas a detecção de intruso que faz o robô passar para o estado "membro".

Estando no estado "membro" os eventos tratados são: a detecção de intruso (presença), o "tick" de tempo (tempo) que após um intervalo, se não houver detecção de intruso, faz com que o robô retorne para o estado "seguidor" e o evento de eleição de líder (toque), que faz com que o robô passe para o estado de "líder" do grupo de robôs.

Na Figura 5.10 (e) é mostrado a sinalização de líder e o robô passa para o estado "lider". No estado "lider" os eventos tratados são: o "tick" de tempo (tempo) tem duas finalidades após um intervalo de tempo faz com que o robô retorne para o estado "seguidor" e a medida que os tick de tempo vão correndo o robô vai coletando as posições dos demais robôs. O evento de eleição de líder (toque), que faz com que o robô transmita os dados colecionados da posição do intruso e passe para o estado "halt", finalizando a execução da aplicação, como mostra a Figura 5.10 (f).

5.1.6 Verificação para o Modelo da Detecção de Intruso

As propriedades que um sistema deve satisfazer fazem parte dos requisitos do sistema, sendo portanto impraticável tentar definir quais propriedade devam ser geradas automaticamente para o modelo. Essa tarefa é de pura responsabilidade do desenvolvedor do sistema. Porém existem definições ou regras gerais que podem auxiliar na descrição de algumas das propriedades que podem ser geradas automaticamente por meio do *GeradorXM*.

A categorização é importante para que dado um determinado modelo a ser verificado, seja possível começar o estudo do problema com o questionamento de quais devem ser as especificações de segurança, atingibilidade, vivacidade e outras que devem ser consideradas. Uma taxonomia aceita em verificação de sistemas utiliza duas propriedades principais [Loer, 2003]:

- Propriedade de Segurança (Safety)- Uma propriedade de segurança especifica que um determinado evento indesejado nunca deve ocorrer, dadas certas condições. Em qualquer sistema, a violação de uma propriedade de segurança tem que permitir que a mesma possa ser observada imediatamente, independentemente do comportamento do sistema no futuro.
- Propriedade de Vivacidade (Liveness) - Uma propriedade de vivacidade especifica que algum evento desejado do modelo deve ocorrer, dadas certas condições.

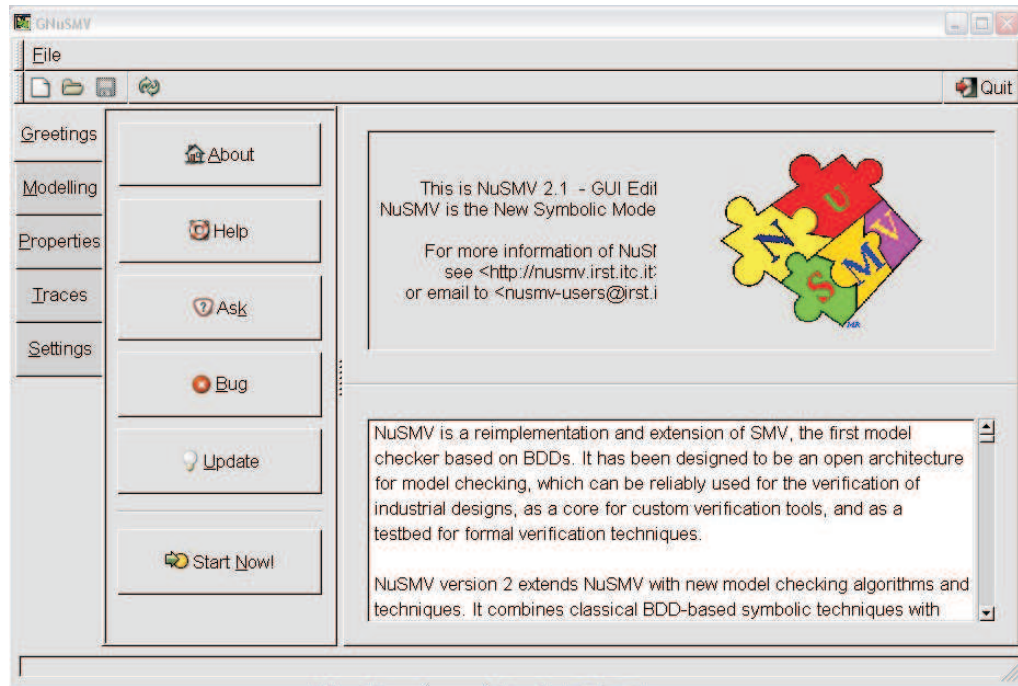
Na Figura 5.11 mostra a saída produzida pelo *GeradorXM* para a aplicação de *Detecção de Intruso*. Para realizar a verificação do Modelo do Sistema para a aplicação de detecção de intruso é utilizado o verificador de modelos *NuSMV*, versão 2.1 com uma interface gráfica. A tela inicial do NuSMV é apresentada na Figura 5.12(a). O modelo deve ser carregado e compilado o que esta apresentado na Figura 5.12(b). Para esta aplicação são verificadas as seguintes propriedade:

- Toda execução da aplicação de detecção de intruso, eventualmente, alcançará o estado "halt".
- Sempre que houver a detecção de intruso o nó passa para o estado de membro.
- Para se tornar líder um nó tem que ser membro e ser eleito (evento toque).

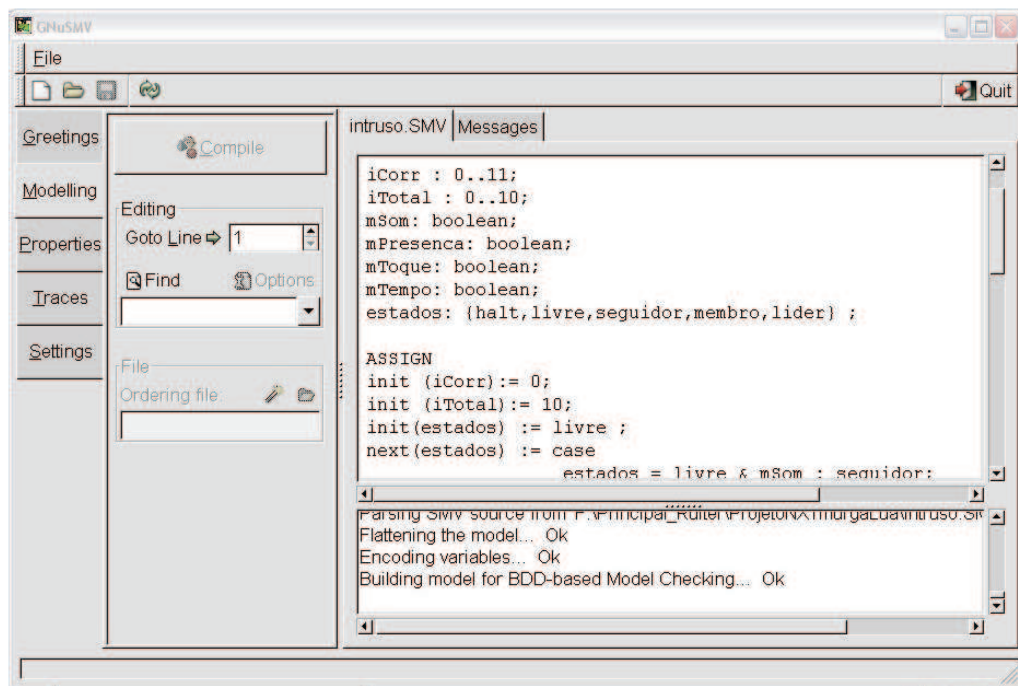
A Figura 5.13(a) mostra as propriedades ainda não verificadas. O resultado da verificação de modelos identifica que as propriedades são verdadeiras, como pode ser observado pelo resultado apresentado na Figura 5.13(b).

```
MODULE main

VAR
iCorr : 0..11;
iTotal : 1..10;
mSom: boolean;
mPresenca: boolean;
mToque: boolean;
mTempo: boolean;
estados: {halt,livre,seguidor,membro,lider} ;

ASSIGN
init (iCorr):= 0;
init (iTotal):= 5;
init(estados) := livre ;
next(estados) := case
                  estados = livre & mSom : seguidor;
                  estados = livre & mPresenca : membro;
                  estados = seguidor & mSom : seguidor;
                  estados = seguidor & mPresenca : membro;
                  estados = seguidor & iCorr > iTotal & mTempo : livre;
                  estados = seguidor & iCorr <= iTotal & mTempo : seguidor;
                  estados = membro & mPresenca : membro;
                  estados = membro & mToque : lider;
                  estados = membro & iCorr <= iTotal & mTempo : membro;
                  estados = membro & iCorr > iTotal & mTempo : seguidor;
                  estados = lider & iCorr > iTotal & mTempo : seguidor;
                  estados = lider & iCorr <= iTotal & mTempo : lider;
                  estados = lider & mToque : halt;
     1 : estados ;
     esac;
next(iCorr) := case
                  estados = livre & mSom : 0;
                  estados = livre & mPresenca : 0;
                  estados = seguidor & mSom : 0;
                  estados = seguidor & mPresenca : 0;
                  estados = seguidor & iCorr <= iTotal & mTempo : iCorr+1;
                  estados = membro & mPresenca : 0;
                  estados = membro & mToque : 0;
                  estados = membro & iCorr <= iTotal & mTempo : iCorr+1;
                  estados = lider & iCorr <= iTotal & mTempo : iCorr+1;
     1 :iCorr ;
     esac;
next(iTotal) := case
     1 :iTotal ;
     esac;

SPEC AG EF estados = halt
SPEC AG mPresenca = 1 -> estados = membro
SPEC AG (estados = membro & mToque = 1) -> AX estados = lider
```

Figura 5.11: Modelo do Sistema para Aplicação de Detecção de Intruso

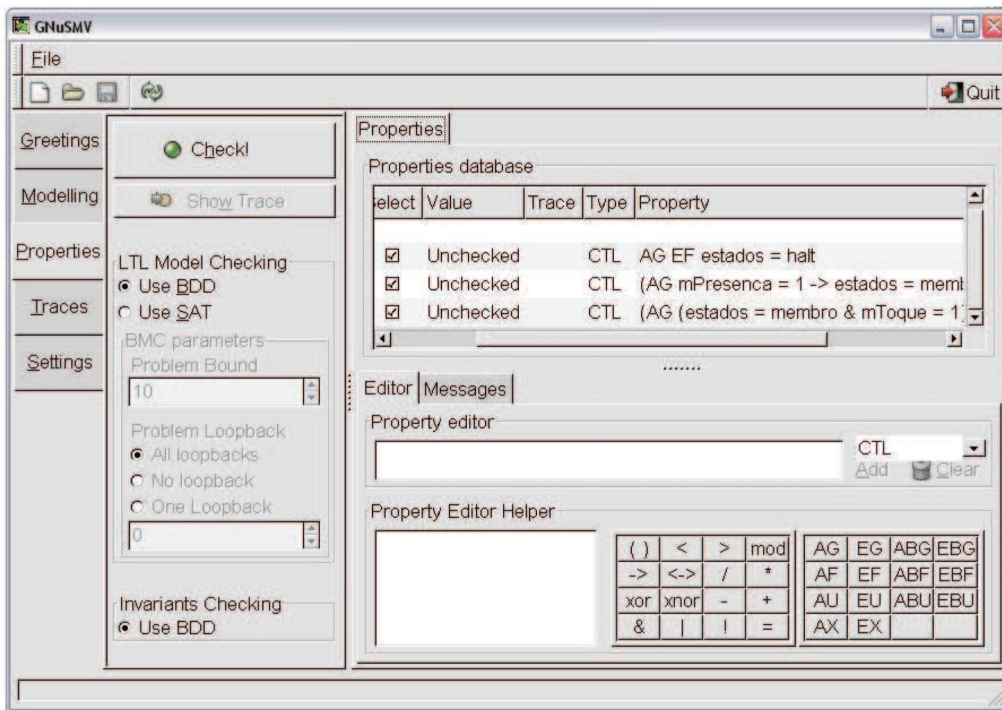


(a) Tela principal do NuSMV

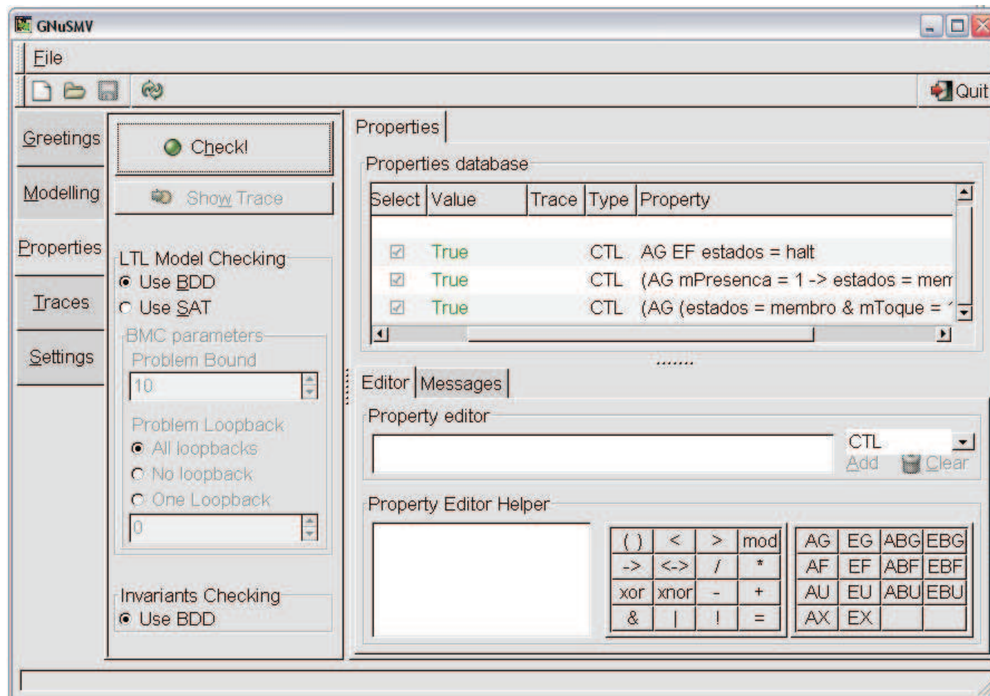


(b) Modelo carregado

Figura 5.12: Verificação da Aplicação de Detecção de Intruso



(a) Propriedades não verificadas



(b) Propriedades verificadas

Figura 5.13: Verificação da Aplicação de Detecção de Intruso (cont)

5.2 Caminhamento Autônomo de Robôs

Um outro exemplo de aplicação para o Modelo *Bare* é o caminhamento autônomo de robôs. Este exemplo é proposto para demonstrar a generalização da aplicação do Modelo *Bare* em problemas de sistemas reativos autônomos em robótica, onde a interface de sensoriamento é o detector de presença e a interface de atuação é o movimento de avanço e de desvio dos obstáculos detectados. O objetivo da aplicação é fazer com que o robô NXT caminhe livremente num ambiente não estruturado, evitando os obstáculos encontrados no caminho, sem a necessidade de uma programação antecipada sobre a existência dos obstáculos ou do seu posicionamento no terreno.

Como todos os detalhes do processo de Desenvolvimento do Modelo *Bare* já foi previamente explicado, neste exemplo o Modelo *Bare* será seguido, mas será apresentado apenas o resultado de cada fase com o comentário necessário. O projeto da aplicação para o caminhamento autônomo de um robô inicia com a identificação dos estados e das variáveis monitoradas e controladas pela aplicação. Os seguintes estados foram identificados para esta aplicação: *free*, *move*, *scan* e *way*. A Figura 5.14 apresenta os estados que compõem a aplicação de caminhamento autônomo e as transições entre os estados.

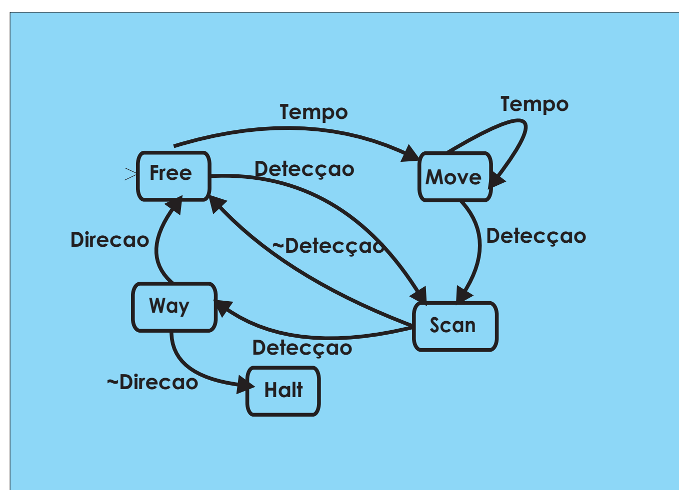


Figura 5.14: Aplicação de Caminhamento Autônomo

O robô inicia no estado *free*, neste estado o robô verifica se existe algum obstáculo para o seu avanço frontal, que é o definido pelo ângulo de ação do sensor de presença, com isso duas situações podem ocorrer:

- Se for detectado algum obstáculo o robô passa então para o estado de *scan*.

- Se não for detectado nenhum obstáculo durante um intervalo de tempo predefinido t, então o caminho está livre e o robô passa para o estado *move*.

No estado *move* o objetivo do robô é avançar no terreno até que algum obstáculo o impeça de continuar. A cada intervalo de tempo ele avança um giro completo das rodas por vez, usando a função "move(0,1)", e verifica se existe algum obstáculo. Se for detectado algum obstáculo, o robô passa para o estado de *scan*. Caso contrário ele permanece no estado *move* e faz um novo avanço.

No estado de *scan* é feito uma confirmação para determinar a posição do obstáculo em relação a rota que robô estava realizando. Duas situações podem ocorrer:

- O obstáculo é confirmado e uma função de verificação, chamada "scan()", é disparada para determinar a posição do obstáculo e fazer uma varredura lateral, da esquerda para a direita, para localizar uma rota alternativa para desviar do obstáculo. Após essa verificação a função devolve um valor indicando o resultado da verificação e passa para o estado *way*.

- Não houve a confirmação do obstáculo, neste caso o robô volta para o estado *free*.

No estado de *way* é feita a tomada de decisão com base na resposta da função "scan()". A função scan indica uma posição sem obstáculo numa escala de -90 até +90 graus de desvio a partir da posição da rota original do robô. Se a resposta tiver valor negativo o desvio é feito para a esquerda, se o valor for positivo o desvio será para a direita. O robô é redirecionado para a posição usando a função "move()" e passa para o estado *free*. Se a resposta for o valor 100 exato então não há rota de desvio para o obstáculo encontrado e o robô vai para o estado de *Halt* e termina a sua execução.

5.2.1 Caminhamento Autônomo no Modelo *Bare*

A descrição completa da aplicação de *caminhamento autônomo* por meio da X-machine estendida pelo Modelo *Bare* tem a seguinte definição:

- $T = (Booleano = \{\ true,\ false\ \})$

- $\Sigma = (mIntruso = \{Booleano\},\ mTempo = \{Booleano\})$

- $\Gamma = (cMove = \{Int, Int\})$

- $\Lambda = (\ iScan = \{Int\},\ iPos = \{Int\})$

- $Q \cup \{I, S, H\} = \{Init,\ Start,\ Halt,\ free,\ move,\ scan,\ way,\ \}$

- $M = (mIntruso,\ mTempo,\ \ iScan,\ iPos)$
- $E = \{e_1 = mIntruso\ \vee\ e_2 = mTempo\} = \{e_1 = false \vee e_2 = false\}$
- $C = \{$
  - $[c_0 = true]$
  - $[c_1 =\sim e_1 \wedge e_2]$
  - $[c_2 = e_1]$
  - $[c_3 = iScan == false]$
  - $[c_4 = iScan == true \wedge iPos \sim= 0 \wedge (e_1 \vee e_2)]$
  - $[c_5 = iScan == true \wedge iPos == 0]$
  - $[c_6 = iPos <= 90 \wedge iPos >= -90]$
  - $[c_7 = iPos == 100]$

  $\}$
- Φ
  - $\phi_0 = (_, _, c_0, _, _)$
  - $\phi_1 = (mTempo \vee mIntruso,\ _,\ c_1,\ _,\ move(0,1))$
  - $\phi_2 = (mIntruso,\ iScan,\ c_2,\ iScan = false,\ "Detectou\ Obstaculo")$
  - $\phi_3 = (_,\ \{iScan, iPos\},\ c_3,\ \{iScan, iPos = scan()\}, "Procura\ Saida")$
  - $\phi_4 = (mTempo \vee mIntruso,\ \{iScan, iPos\},\ c_4,\ _,\ "Encontra\ Saida")$
  - $\phi_5 = (mTempo,\ \{iScan, iPos\},\ c_5, _,\ "Sem\ Obstaculo")$
  - $\phi_6 = (mTempo, iPos, c_6, _, move(2, iPos)$
  - $\phi_7 = (mTempo, iPos, c_7, _, "Sem\ Caminho")$
- F
  - $(Init, \phi_0, Start)$
  - $(Start, \phi_0, free)$
  - $(free, \phi_1, move)$
  - $(free, \phi_2, scan)$
  - $(move, \phi_1, move)$
  - $(move, \phi_2, scan)$

- $(scan, \phi_3, scan)$
- $(scan, \phi_4, way)$
- $(scan, \phi_5, free)$
- $(way, \phi_6, free)$
- $(way, \phi_7, Halt)$

- $q_0 = \{Init\}$
- $m_0 = (mIntruso = false,\ mTempo = false,\ iPos = 0,\ iScan = false)$

5.2.2 Gerador da Aplicação de Caminhamento Autônomo

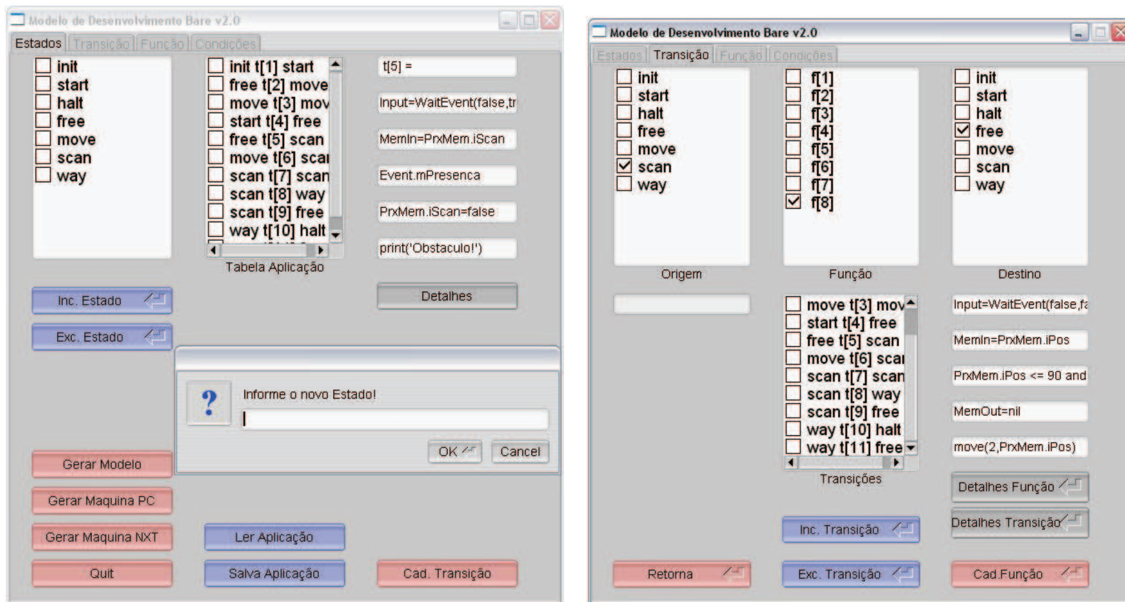
O *GeradorXM* é utilizado para fazer o cadastramento da aplicação no Modelo *Bare*. Na tela da Figura 5.15(a) são apresentados os estados da aplicação de caminhamento autônomo de robô. A Figura 5.15(b) apresenta as transições entre os estados e um exemplo da inclusão de uma nova transição, que é cadastrada pela indicação do estado origem (scan), usando uma função (f[8]) e um estado destino (free).

Para a inclusão de novas funções de transição a tela de cadastro de função é acionada. As funções devem relacionar os eventos detectados e um elemento da memória de entrada, e produzir como resultado os eventos disparados ou as mensagens, e um elemento de memória de saída. Toda função possui uma **condição** de avaliação de transição vinculada a um evento. A tela da Figura 5.15(c) mostra como exemplo a criação de uma funções para a aplicação de caminhamento autônomo que dispara o movimento do robô. Um dos eventos de saída importante é o comando "CMD" pois ele permite que, na habilitação da transição entre os estados, um evento seja disparado pela execução de um código, como a chamada a uma função para movimentar as rodas do robô.

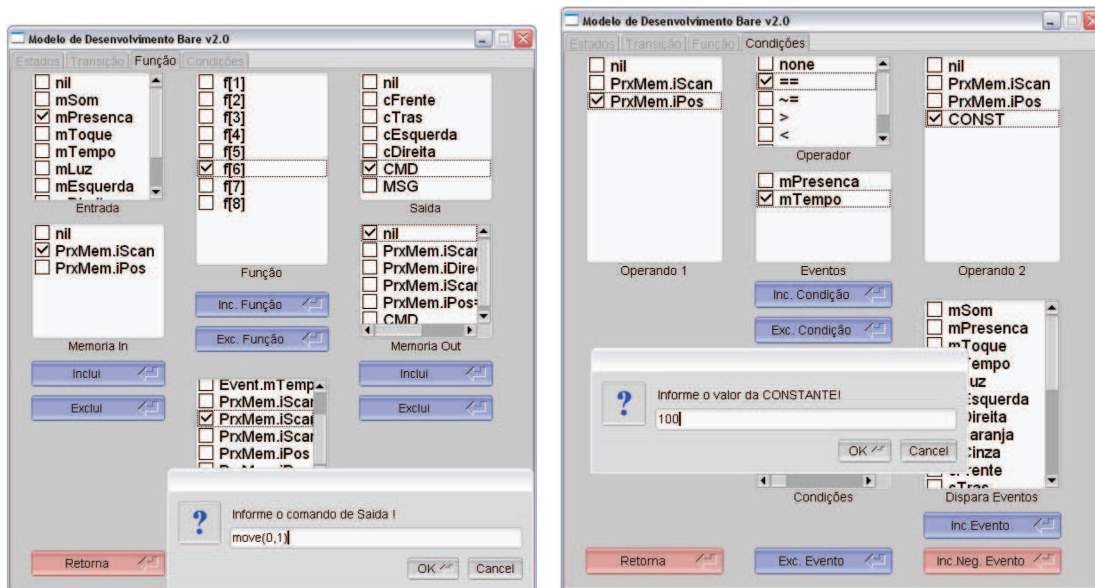
Cada **condição** é formada por uma expressão lógica dos elementos de memória usando os operadores relacionais e está vinculada a um evento. Na tela da Figura 5.15(d) mostra a construção de uma condição que faz uma verificação de igualdade da variável "iPos == 100" na presença do evento "tempo". Em seguida um "Modelo do Sistema" deve ser gerado para ser verificado pela ferramenta NuSMV. O próximo passo é gerar a tabela da aplicação para ser executada pelo ExecutorXM.

5.2.3 Mapeamento para o Modelo Tabular

Depois de concluída a descrição da aplicação o próximo passo é a transformação da especificação da aplicação para o formato tabular. O **GeradorXM** é utilizada para



(a) Inclusão de Estados (b) Cadastro da Transição dos Estados



(c) Cadastro das Funções de Transição (d) Cadastro das Condições e Eventos

Figura 5.15: Gerador da Aplicação de Caminhamento Autônomo

descrever a aplicação e realizar a transformação para o modelo de execução no formato tabular, como descrito na Seção 3.3. tabela resultante para a aplicação *caminhamento autônomo* é mostrada na Figura 5.2. Nela estão resumidos todos os componentes necessários para a execução da aplicação.

Source	**Input**	**Mem_input**	**Target**	**Condition**	**Mem_output**	**Output**
init	*nil*	*nil*	start	*true*	*nil*	"*Init*"
start	*nil*	*nil*	free	*true*	$iPos = 0$; $iScan = false$	"*Start*"
free	$mTempo \vee$ $mIntruso$	_	move	c_1	_	$move(0, 1)$
free	$mTempo \vee$ $mIntruso$	*iScan*	scan	c_2	$iScan = false$	"*Detectou Obstaculo*"
move	$mTempo \vee$ $mIntruso$	_	move	c_1	_	$move(0, 1)$
move	*mIntruso*	*iScan*	scan	c_2	$iScan = false$	"*Detectou Obstaculo*"
scan	*mTempo*	$\{iScan, iPos\}$	scan	c_3	$iScan, iPos = scan()$	"*Procura Saida*"
scan	$mTempo \vee$ $mIntruso$	$\{iScan, iPos\}$	way	c_4	_	"*Encontra Saida*"
scan	*mTempo*	$\{iScan, iPos\}$	free	c_5	_	"*Sem Obstaculo*"
way	*mTempo*	*iPos*	free	c_6	_	$move(2, iPos)$
way	*mTempo*	*iPos*	halt	c_7	_	"*Sem Caminho*"
halt	*nil*	*nil*	halt	*nil*	*nil*	*nil*

Tabela 5.2: Tabela de Execução para Aplicação de Caminhamento Autônomo

As Figuras 5.16a e 5.16b apresentam, respectivamente, a relação dos eventos monitorados pela aplicação e as expressões que formam as condições relacionadas com os eventos que são avaliadas na ocorrência do evento.

5.2.4 Execução da Aplicação de Caminhamento no Robô

Para executar a aplicação de caminhamento autônomo no robô NXT os mesmos passos do exemplo anterior são executados. O robô NXT é configurado como apresentado na Figura 5.5, nele estão ativos como entrada os sensores de presença, de som, de toque e um temporizador interno. Como interface de saída é usado o display de LCD e os motores que acionam as rodas. As rotinas básicas em *pbLua* do ExecutorXM e as rotinas de interface são as mesmas. Para a detecção dos eventos externos é utilizada a rotina de interface de entrada, chamada "WaitEvent()", que detecta a ocorrência

Memory	Events	Value
mIntruso	e_1	false
mTempo	e_2	false
	$\bigvee E$	false

(a) Eventos

Conditions	Expression	Value
c_1	$\sim e_1 \wedge e_2$	false
c_2	e_1	false
c_3	$iScan == false$	false
c_4	$iScan == true \wedge iPos \sim= 0 \wedge (e_1 \vee e_2)$	false
c_5	$iScan == true \wedge iPos == 0$	false
c_6	$iPos <= 90 \wedge iPos >= -90$	false
c_7	$iPos == 100$	false

(b) Condiçoes

Figura 5.16: Tabelas Auxiliares para Caminhamento Autônomo

de um evento no mundo real e sinaliza para a aplicação. Foi utilizada ainda uma rotina de interface chamada “scan()” para fazer um varredura de 180 graus do campo de visão do robô, procurando por uma rota para fazer o desvio do obstáculo. Para a interface de saída além da utilização do display de LCD, os motores são utilizados como atuadores em resposta aos eventos detectados, para tanto é utilizada uma rotina chamada “move()”.

As seguintes rotinas executam no Robô NXT:

- scan() (Figura 5.17) Essa rotina foi desenvolvida para executar uma varredura de 180 graus no espaço à frente do robô e procura uma posição sem obstáculo. Ela produz como resultado um valor *true* informado que a rotina finalizou e 3 tipos de respostas numéricas. A primeira resposta é 0(zero) indicando que não foi detectado nenhum obstáculo a frente do robô. Para a segunda resposta o sonar é movimentado para a esquerda e depois para a direita, em ângulos de 45 e 90 graus. A cada posicionamento é feito uma mova leitura para saber se existe um obstáculo naquela direção. Caso não seja detectado nenhum obstáculo ao posicionar o robô em uma direção, essa direção é devolvida em graus, sendo que o valor positivo ou negativo indica se é para direita ou esquerda, respectivamente. A terceira resposta é dado caso seja detectado que não existe uma região livre de obstáculos, neste caso o rotina devolve o valor 100(cem).

```
function scan()
    nxt.OutputResetTacho(3,1)
    nxt.OutputSetRegulation(3,1,1)
    setupI2C(1)
    nxt.I2CSendData(1, nxt.I2Ccontinuous, 0)
    if lerSonar() > 25 then return true, 0 end
    nxt.OutputSetSpeed(3,0x20,-50,45)
    if lerSonar() > 25 then nxt.OutputSetSpeed(3,0x20,50,45)
            return true, -45 end
    nxt.OutputSetSpeed(3,0x20,50,90)
    if lerSonar() > 25 then nxt.OutputSetSpeed(3,0x20,-50,45)
            return true, 45 end
    nxt.OutputSetSpeed(3,0x20,-50,135)
    if lerSonar() > 25 then nxt.OutputSetSpeed(3,0x20,50,90)
            return true, -90 end
    nxt.OutputSetSpeed(3,0x20,50,180)
    if lerSonar() > 25 then nxt.OutputSetSpeed(3,0x20,-50,90)
            return true, 90
    else
            nxt.OutputSetSpeed(3,0x20,-50,90)
            return true, 100
    end
    collectgarbage()
end
```

Figura 5.17: Interface para Detecção de Obstáculos

- move() (Figura 5.18) Esta rotina é uma rotina de interface de saída. Ela recebe como parâmetro a direção e um valor indicando o tipo de movimento. Se a direção for 0 (zero) ou 1(um), indica frente ou trás respectivamente, neste caso o valor indicará o número de giros completos de 360 graus sobre ambas as rodas. Se a direção for 3, indicará um movimento lateral para a esquerda ou direita em graus, a partir da posição frontal do robô. Neste caso o segundo valor informa o tipo de movimento da seguinte forma: se for um valor positivo será um movimento para a direita, caso seja um valor negativo será um movimento para a esquerda.

A aplicação de detecção de caminhamento autônomo que será executada pelo *ExecutoXM* no robô NXT tem o formato apresentado na Figura 5.19. Esse código é o mesmo apresentado na Tabela 5.2, ele é transferido no formato de texto para o interpretador *pbLua* via uma conexão USB. Durante a transferência o código é interpretado e executado gerando, com isso, um arquivo interno com a tabela da aplicação.

Neste experimento é necessário deixar o robô com liberdade de movimento para caminhar no ambiente, por causa disso não é usada a conexão USB com o *host*, em seu lugar é utilizado o arquivo padrão, chamado *pbLuaStartup*, o arquivo padrão é

```
function move(direcao,voltas)
    local velo=50
    nxt.OutputSetRegulation(1,1,1)
    nxt.OutputSetRegulation(2,1,1)
    if voltas ~=0 then
        if direcao == 0 then nxt.DisplayText("Para Frente ∧ ")
        elseif direcao == 1 then
            velo= -50
            nxt.DisplayText("Para Tras ∨ ")
        end
        if direcao == 0 or direcao == 1 then
            nxt.OutputSetSpeed(1,0x20,velo,voltas*360)
            nxt.OutputSetSpeed(2,0x20,velo,voltas*360)
        elseif direcao == 2 and voltas < 0 then
           nxt.OutputSetSpeed(1,0x20,-velo,nxt.abs(voltas*5))
            nxt.DisplayText("< Esquerda")
        else nxt.OutputSetSpeed(2,0x20,-velo,voltas*5)
            nxt.DisplayText("Direita >")
        end
    end
    Wait(50)
end
```

Figura 5.18: Interface para Atuação de Eventos

um conjunto de comandos na linguagem *Lua* que são carregados e executados. Este arquivo é criado uma única vez e executado sempre que o robô NXT é ligado e uma conexão USB/Bluetooth não é completada. A Figura 5.20 mostra o arquivo padrão que é construído e gravado no robô NXT para este experimento. Os comandos são gravados como cadeias de caracteres para o arquivo, desta forma é feita uma concatenação de todos os comandos e a cadeia final concatenada é gravada no arquivos *pbLuaStartup*. As rotinas $EX01$, $EX02$ e $EX03$, são os arquivos com as funções do ExecutorXM, em seguida tem uma chamada para inicializar os contadores dos giro das rodas e a chamada para o ExecutorXM passando como parâmetro a aplicação de caminhamento autônomo.

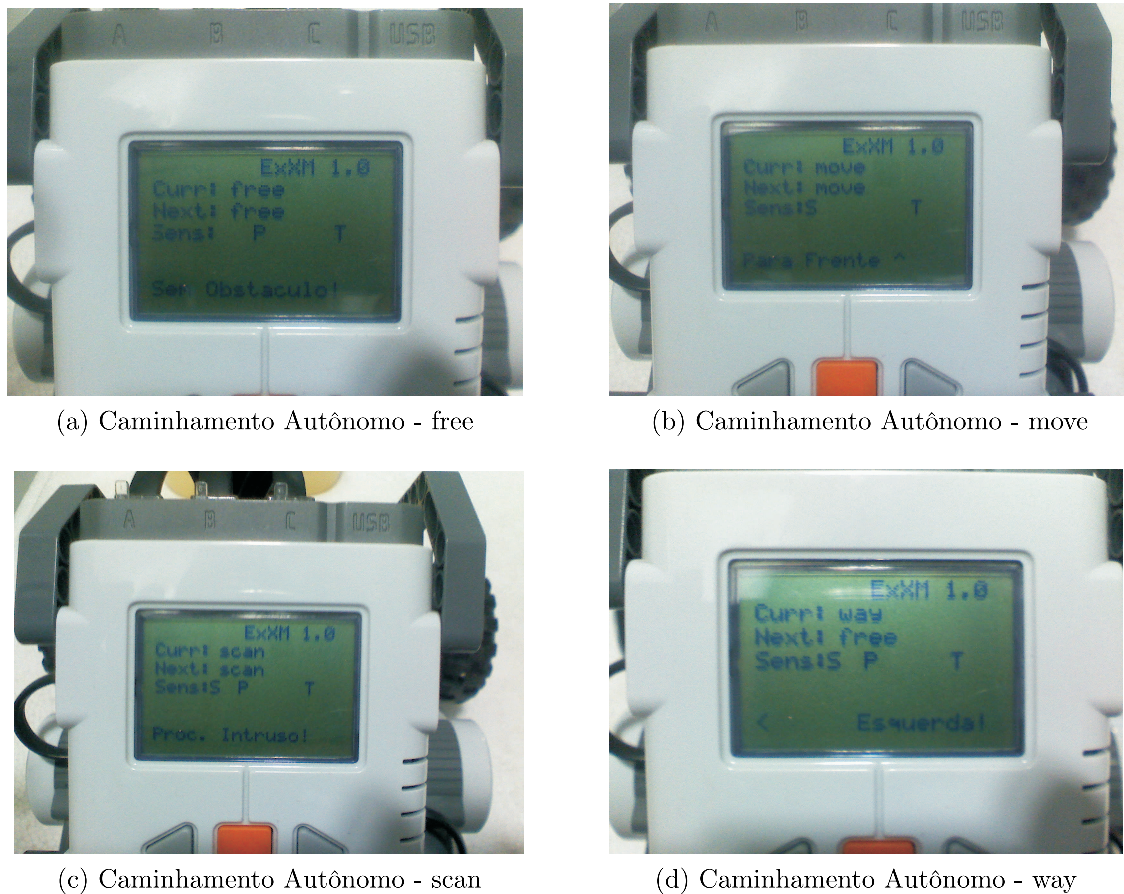
Quando a aplicação inicia o display de LCD mostra que o ExecutorXM está carregado e a partir daí começa a execução da aplicação. O estado corrente é definido e a execução da aplicação é direcionada pelos eventos do mundo real que são capturados pelos sensores do robô NXT e pela evolução dos estados internos correntes da máquina. A aplicação inicia no estado "init", executa as inicializações das tabelas internas (Mem, PrxMem, Event) e, em seguida, passa para o estado "start" e depois para o próximo estado que é o estado "free". A Figura 5.21 (a) mostra a aplicação de caminhamento au-

```
mx = nxt.FileCreate("caminha.nxt", 5122 )
Str = [[ $ init $ $ Event = { }; ClearEvent = function()
Event= { mPresenca=false , mTempo=false } end$
$ Mem = { mPresenca=false,mTempo=false,iScan=false,iPos=0 }$
$ start$ $ true$
$ PrxMem={}; mt = { __index = function (PrxMem,k) return Mem[k] end,
__newindex = function (PrxMem,k,v) Event[k]=true; Mem[k] = v end };
setmetatable(PrxMem, mt)$
$ Output=nil $ ]]
Str=string.gsub(Str," \n "," ")
nxt.FileWrite(mx,Str.."\n")

Str=[[$free$$Input=WaitEvent(false,true,false,false,false)$
$MemIn=nil$$move$
$Event.mTempo and not Event.mPresenca $
$MemOut=nil$$move(0,1)$ ]]
Str=string.gsub(Str,"\n",)
nxt.FileWrite(mx,Str.."\n")

Str=[[$move$$Input=WaitEvent(false,true,false,false,false)$
$MemIn=nil$$move$
$Event.mTempo and not Event.mPresenca $
$MemOut=nil$$move(0,1)$ ]]
Str=string.gsub(Str,"\n",)
nxt.FileWrite(mx,Str.."\n")
```

Figura 5.19: Tabela com Trecho da Aplicação de Caminhamento Autônomo

```
programString = [[
nxt.dofile("EX01")
nxt.dofile("EX02")
nxt.dofile("EX03")
nxt.OutputResetTacho(1,1)
nxt.OutputResetTacho(2,1)
nxt.OutputResetTacho(3,1)
Executor("caminha.nxt")
]]

h = nxt.FileCreate( "pbLuaStartup", 512 )

nxt.FileWrite( h, programString )

nxt.FileClose( h )
```

Figura 5.20: Arquivo de Inicialização para o robô NXT



(a) Caminhamento Autônomo - free

(b) Caminhamento Autônomo - move

(c) Caminhamento Autônomo - scan

(d) Caminhamento Autônomo - way

Figura 5.21: Execução da Aplicação Caminhamento Autônomo

tônomo, no estado “free”. Neste estado dois eventos são tratados, o tempo e a detecção de um obstáculo. Caso não seja detectado um obstáculo e o evento de passagem tempo $t = 3\ seg$ ocorra, a aplicação passa para o estado “move” e executa um movimento para frente, chamando a função “move(0,1)”. Caso seja detectado um obstáculo então a aplicação irá para o estado “scan”.

No estado “move” o robô avançará enquanto nenhum obstáculo atrapalhe seu caminho, neste estado dois eventos são considerados: O tempo que faz com que o robô avance sempre que ocorrer a passagem de um intervalo de tempo. A detecção de um obstáculo, que faz com que o robô passe para o estado de detecção “scan”. A Figura 5.21 (b) mostra a aplicação de caminhamento autônomo, no estado “move” fazendo o avanço pela passagem do tempo.

No estado “scan” o robô executa a função $scan()$ que confirma a existência de um obstáculo no caminho do robô e faz uma varredura lateral, da esquerda para direita, procurando por um caminho livre que sirva como uma rota de desvio para o obstáculo detectado. Ao executar a função $scan()$ três situações podem ocorrer: A evolução da

máquina só acontece quando a função $scan()$ termina a varredura, o que é indicado pela atualização da variável $iScan$, independente dos eventos ocorrerem e a máquina continua no estado "scan". Quando a função de varredura termina, duas situações podem ocorrer: Caso ocorra os eventos de passagem do tempo ou a detecção de um obstáculo, e o resultado da função $scan()$ seja diferente de 0, a máquina vai para o estado "way". Caso a função $scan()$ termine e o resultado seja igual a 0, então nenhum obstáculo foi confirmado e a máquina vai para o estado "free". A Figura 5.21 (c) mostra a aplicação de caminhamento autônomo, no estado "scan" avaliando o ambiente e procurando por uma rota para desviar do obstáculo.

No estado "way" o robô executa uma tomada de decisão com base no resultado devolvido pela função $scan()$. Chegar neste estado significa que foi confirmada a existência de um obstáculo no caminho do robô e foi executada uma busca por um caminho para uma rota de desvio do obstáculo detectado. Neste ponto duas situações podem ocorrer: A função $scan()$ detectou um rota de desvio e o valor é passado para a variável $iPos$, numa faixa de -90 até +90 graus. Neste caso a função $move()$ é chamada passando como parâmetros a direção lateral "3" e a direção do desvio "$iPos$". Caso o valor retornado pela função $scan()$seja o valor 100 exato, então não foi encontrado uma rota para o desvio e o próximo estado é o estado "halt". A Figura 5.21 (d) mostra a aplicação de caminhamento autônomo, no estado "way" chamando a função $move(3, iPos)$ para executar o desvio do obstáculo.

5.2.5 Verificação para o Modelo do Caminhamento Autônomo

Na Figura 5.22 mostra a saída produzida pelo $GeradorXM$ para a aplicação *caminhamento*. Na verificação do Modelo do Sistema para a aplicação de caminhameto é utilizado o verificador de modelos *NuSMV*, versão 2.1. O modelo é carregado e compilado como apresentado na Figura 5.23(a). Para a aplicação de caminhamento autônomo são verificadas as seguintes propriedade:

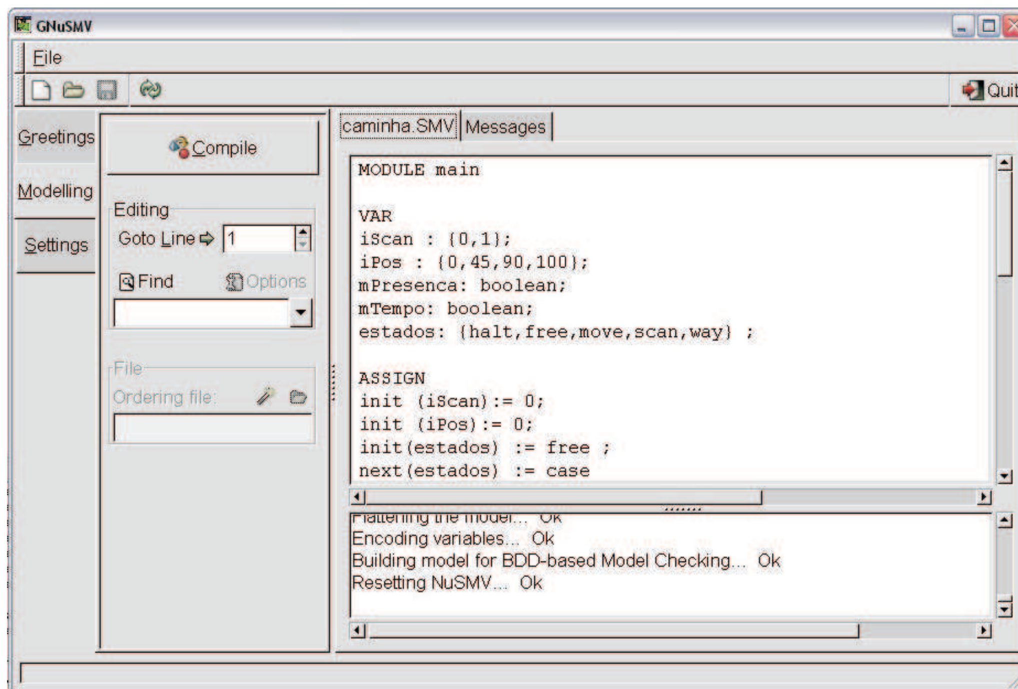
- Toda execução da aplicação de caminhamento autônomo, eventualmente, alcançará o estado "halt".
- Sempre que não houver obstáculo o robô se movimentará.
- O robô só para se não houver nenhuma opção de caminho (iPos=100).

A Figura 5.23(b) mostra as propriedades ainda não verificadas. O resultado da verificação de modelos identifica que as propriedades são verdadeiras, como pode ser

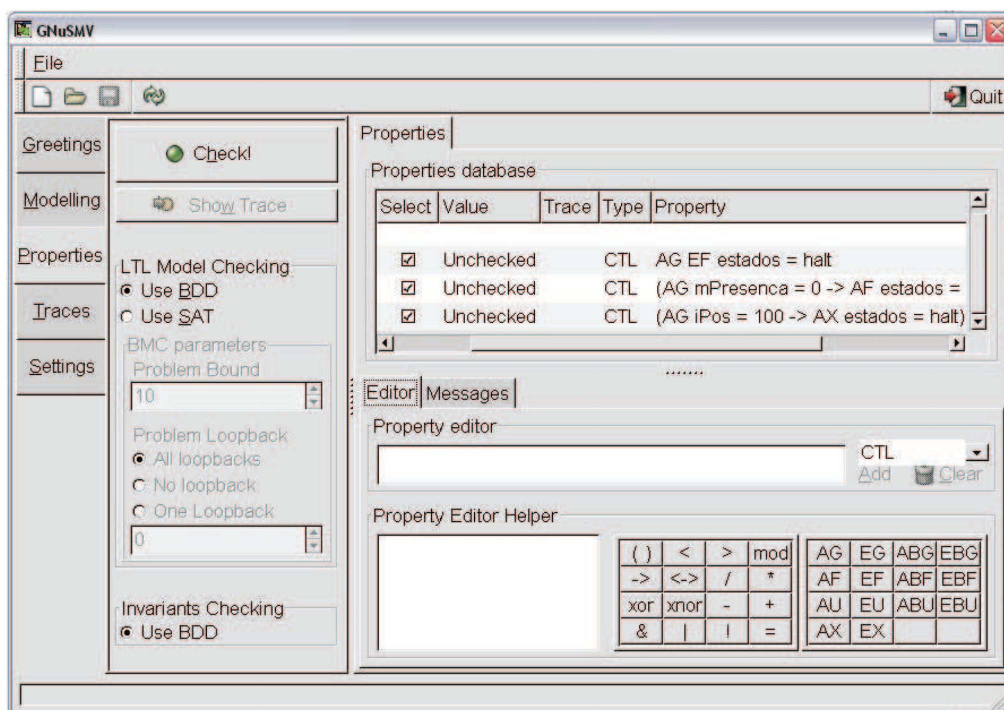
observado pelo resultado apresentado na Figura 5.24(a). Na Figura 5.24(b) é apresentado um trecho da execução da aplicação de caminhamento autônomo, mostrando os valores atribuídos pelo verificador NuSMV para as variáveis monitoradas, internas e para os estados do modelo.

```
MODULE main
VAR
iScan : {0,1};
iPos : {0,45,90,100};
mPresenca: boolean;
mTempo: boolean;
estados: {halt,free,move,scan,way} ;
ASSIGN
init (iScan):= 0;
init (iPos):= 0;
init(estados) := free ;
next(estados) := case
                  estados = free & mTempo &  !  mPresenca  : move;
                  estados = move & mTempo &  !  mPresenca  : move;
                  estados = free & mPresenca : scan;
                  estados = move & mPresenca : scan;
                  estados = scan & iScan =  0  : scan;
                  estados = scan & iScan =  1  & iPos != 0 & (mPresenca | mTempo) : way;
                  estados = scan & iScan =  1  & iPos = 0 : free;
                  estados = way & iPos = 100 : halt;
                  estados = way & iPos <= 90 & iPos >= -90 : free;
     1 : estados ;
     esac;
next(iScan) := case
                  estados = free & mPresenca :  0 ;
                  estados = move & mPresenca :  0 ;
                  estados = scan & iScan =  0  :  {0,1};
     1 :iScan ;
     esac;
next(iPos) := case
                  estados = scan & iScan =  0  :  {0,45,90,100};
                  estados = way & iPos = 100 : 0;
     1 :iPos ;
     esac;

SPEC AG EF estados = halt
SPEC AG mPresenca = 0 -> AF estados = move
SPEC AG iPos = 100 -> AX estados = halt
```

Figura 5.22: Modelo do Sistema para Aplicação de Caminhamento Autônomo

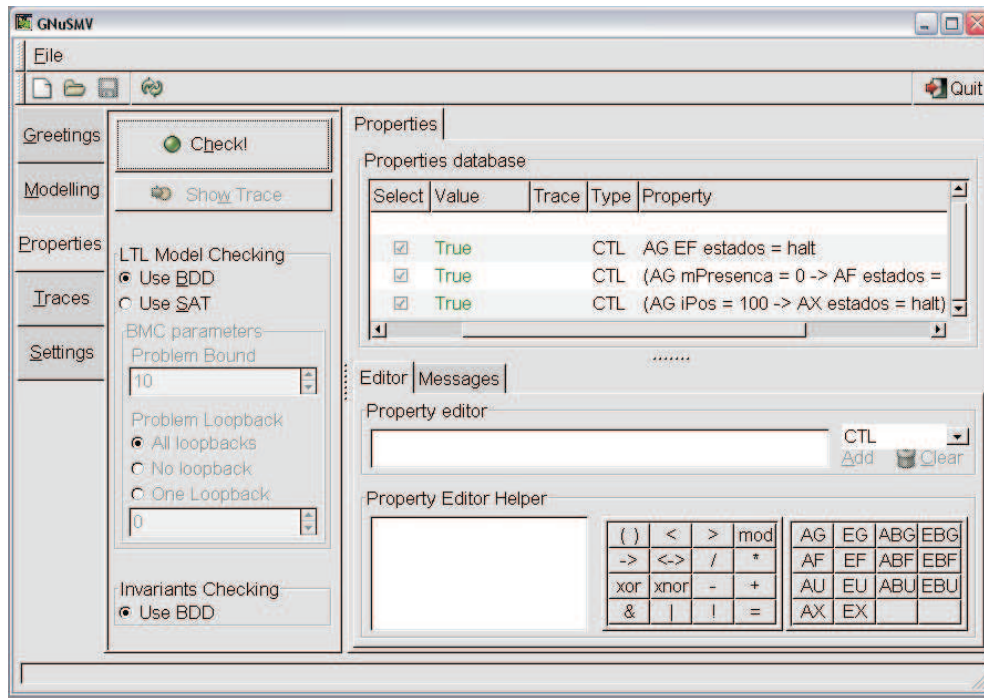


(a) Modelo carregado

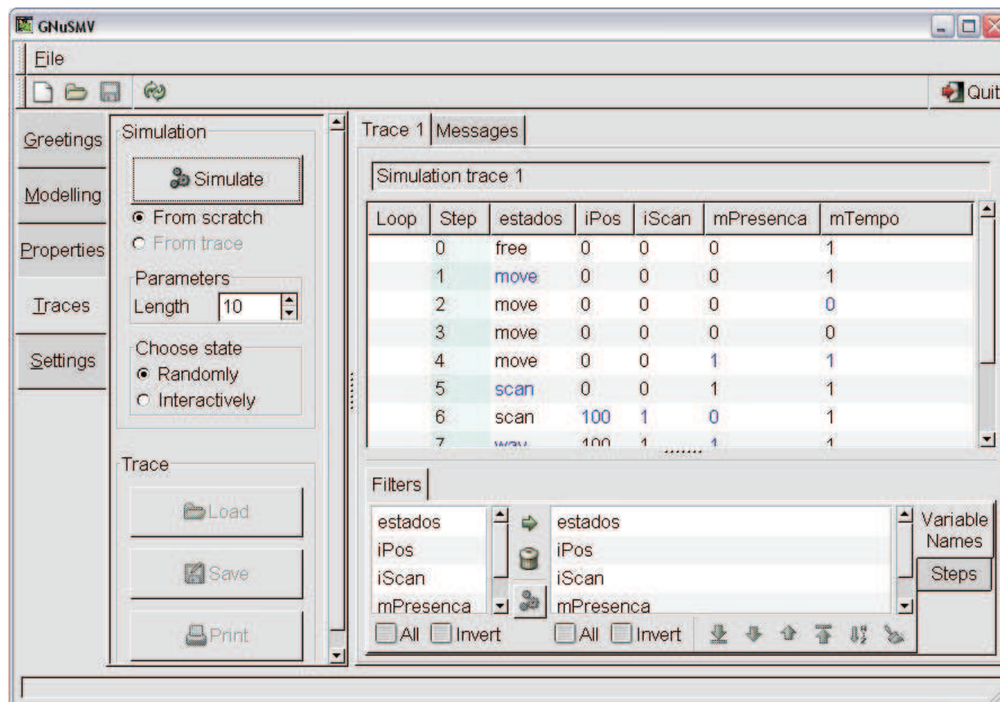


(b) Propriedades não verificadas

Figura 5.23: Verificação da Aplicação de Caminhamento Autônomo



(a) Propriedades verificadas



(b) Trace de execução

Figura 5.24: Verificação da Aplicação de Caminhamento Autônomo (cont.)

5.3 Posicionamento de Nós Sensores usando Robôs

Neste exemplo é feita uma união entre a área de Rede de Sensores Sem Fio e a Robótica, dotando o nó sensor com mobilidade. Um algoritmo foi elaborado para realizar o posicionamento dos nós sensores numa área de monitoramento. O algoritmo considera que um nó sensor está acoplado a cada robô NXT e este faz parte de um grupo de robôs com liberdade de movimentação no ambiente. O objetivo do grupo é fazer o posicionamento mais adequado do nó sensor para estabelecer uma rota de comunicação entre o nó sorvedouro (*sink*) e o ambiente onde está ocorrendo o evento a ser monitorado. O estabelecimento da rota de comunicação pode ser definido com uma maior inserção no interior da área monitorada, ou com uma maior abrangência de cobertura lateral da área monitorada. Com este exemplo é aberta uma nova possibilidade para depositar os nós sensores por meio de robôs, garantindo que ele alcançará, de maneira eficiente, uma maior área de cobertura para comunicação e transmissão dos dados coletados usando um número menor de nós. A seguir são apresentados exemplos para os tipos de abordagem citados para o posicionamento dos nós sensores:

- Em profundidade - se for necessário fazer uma cobertura de inserção em território inimigo, a cobertura em profundidade será utilizada. Os robôs penetrarão numa linha imaginária e tentarão chegar o mais longe possível dentro do território inimigo.

- Em amplitude - se for necessário fazer uma cobertura para realizar uma detecção de invasão ou proteção de uma reserva ambiental, então a forma de atuação deverá ser em amplitude. A disposição será mais lateral possível para alcançar uma maior área de cobertura e atuação.

- Híbrido - quando for necessário realizar uma busca numa grande área, uma mistura das duas abordagens será mais útil, iniciando com uma distribuição lateral e em seguida uma inserção em profundidade.

Na maioria das aplicações em RSSF os sensores são depositados aleatoriamente em ambientes sem nenhuma infraestrutura ou até mesmo jogados por via aérea na região de interesse e depois é executado um algoritmo de roteamento para definição das rotas de comunicação, onde alguns nós são ligados ou desligados para fazer o controle de densidade da rede [Tubaishat & Madria, 2003]. Um segunda forma é realizar previamente um cálculo sobre a área onde os nós serão depositados e em seguida definir as posições de melhor densidade para depositar os nós [Sitharama et al., 2001]. O

trabalho com maior relação, que temos conhecimento até o momento, sobre a disposição de nós sensores usando robôs é realizado em [Parker et al., 2003], porém nessa abordagem existe a necessidade do conhecimento *a priori* do ambiente, é necessário um cálculo prévio para otimizar o posicionamento dos nós, são necessários 3 tipos de robôs (*Leader Helper, Follower Helper, Sensor Nodes*) e os experimentos foram realizados em um ambiente simulado. Existem outros trabalhos que envolvem teorias mais elaboradas, tais como os campos potencias de [Howard et al., 2002] que maximiza a área de cobertura e o trabalho de [Payton et al., 2001] que utiliza o conceito feromônio virtual para atração e repulsão em enxame de robôs para distribuí-los numa área desconhecida. Uma comparação direta entre estes trabalhos esta fora do escopo deste exemplo. Porém pode ser considerada um vantagem intuitiva do algoritmo a utilização de um número menor de nós robôs/sensores, para realizar a tarefa de detecção de intruso, devido a estratégia de posicionamento dos nós de maneira lateral na fronteira da região. Uma outras vantagem vislumbrada é a possibilidade da realização de um reposicionamento (sensores móveis) do nós sensores para se adaptar as características de novas aplicações.

5.3.1 Algoritmo de Posicionamento de Nós Sensores

Na descrição do algoritmo de posicionamento usando robôs, as seguintes considerações devem ser observadas:

- Todo robô carrega um nós sensor e tem a mesma configuração.
- Os nós sensores transmitem e recebem a sua posição de forma radial.
- Os robôs tem liberdade para seguir em qualquer direção.
- O objetivo é distribuir os robôs de forma a alcançar a maior área possível em profundidade ou em amplitude.
- n = numero de robôs.
- ht = numero de *hosts* já posicionados.

Os passos do algoritmo para posicionamento de nós sensores usando robôs, definido na Figura 5.25, pode ser acompanhado pelos passos apresentados na Figura 5.26:

1. Os n robôs inicialmente estão juntos no raio de alcance do nó *sink*, chamando de host-corrente (ht=1), .
2. Os robôs são inicializados para seguir qualquer direção, mantendo-se dentro do alcance de outro nó e procurando se afastar o máximo possível do host corrente.
3. Repetição
   a) O primeiro robô que chega no limite do raio de comunicação com o nó host corrente (ht=1), para e solicita do host corrente para ser o novo host corrente (ht+1), o host corrente autoriza o primeiro que fizer a solicitação e atualiza o valor do host corrente para o novo valor (ht=2), o novo host fixa a sua posição e divulga o valor do host corrente (ht=2) na rede.
   b) Os robôs continuam andando e afastando um dos outros o máximo possível dentro do limite de comunicação, mantendo alcance de pelo menos um nó simples e procurando pelo host corrente divulgado pela rede (ht=2) até número do host corrente seja igual ao número de robôs (ht=n)
4. Quando todos os nós simples forem hosts teremos uma rota de comunicação definida com maior alcance possível em profundidade ou amplitude.

Figura 5.25: Algoritmo para Posicionamento de Nós Sensores Usando Robô

5.3.2 Posicionamento de Nós Sensores no Modelo *Bare*

Para a aplicação de posicionamento de nós sensores usando robôs, os seguintes estados foram identificados: *sleep*, *near*, *far* e *host*. Um robô inicialmente esta no estado *sleep*, neste estado o robô está inativo e aguarda o comando para iniciar a tarefa de posicionamento. No estado "sleep" somente o evento de "toque" é tratado, ao detectar o evento o robô acorde, faz uma verificação no ambiente e passa para o estado "near". No estado "near" é feita a descoberta dos outros nós, em busca do *host* corrente. Depois de descoberta a posição do host corrente a aplicação passa para o estado "far". No estado "far" o robô procura permanecer no limite do alcance de comunicação do host corrente e para isso faz um movimento de afastamento com maior precisão, chegando no limite de alcance ele para e passa para o estado de "host". No estado de "host" o robô deve requisitar do host corrente a autorização para se transformar no novo host, e então passa a transmitir essa informação na rede. A Figura 5.27 apresenta os estados que compõem a aplicação de posicionamento de nós sensores com as transições entre os estados.

A descrição da aplicação para o *posicionamento de nós sensores* usando a X-machine estendida pelo Modelo *Bare* é definido a seguir:

- $T = (Booleano = \{\ true,\ false\ \})$

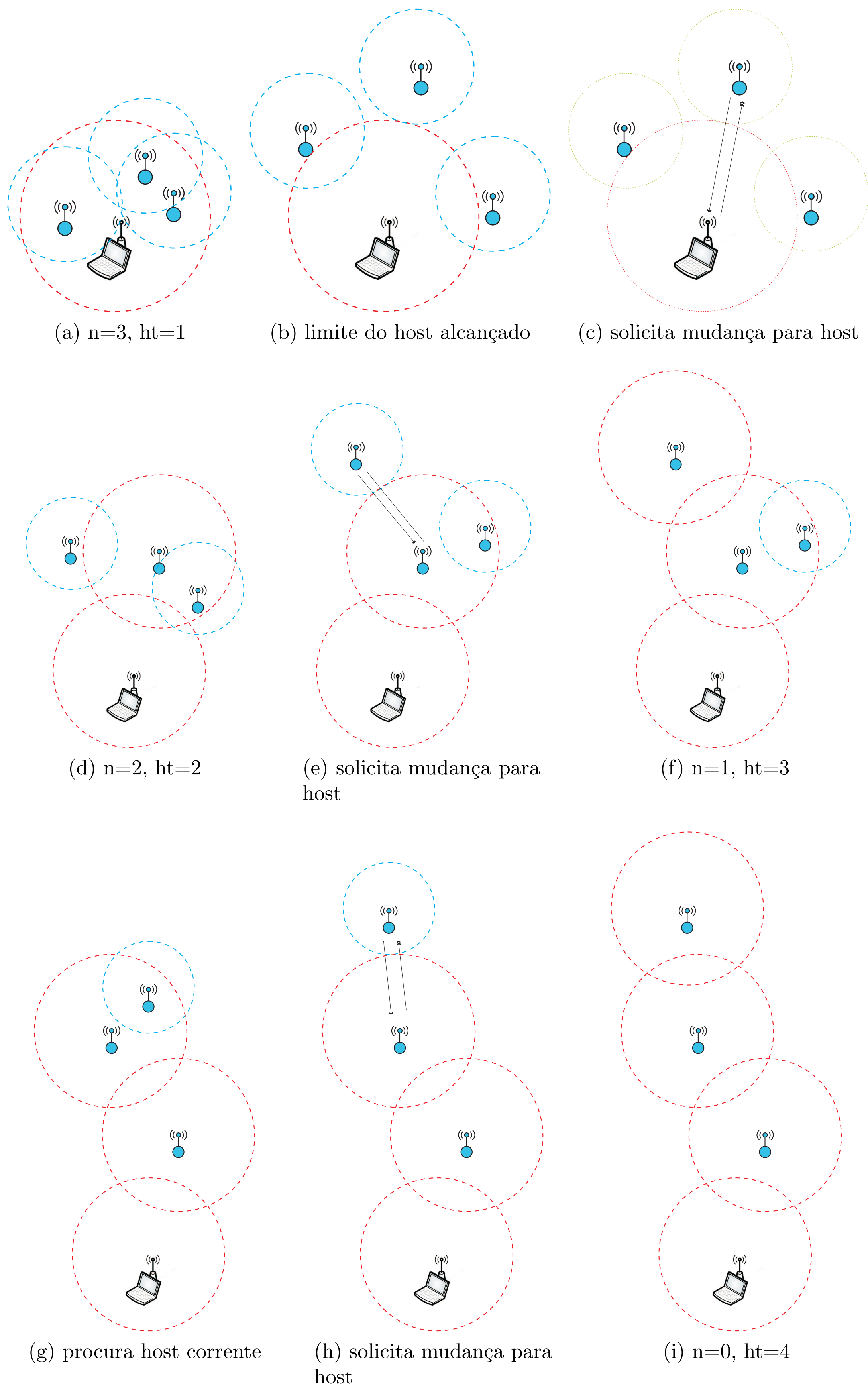


Figura 5.26: Posicionamento de Nós Sensores Usando Robô

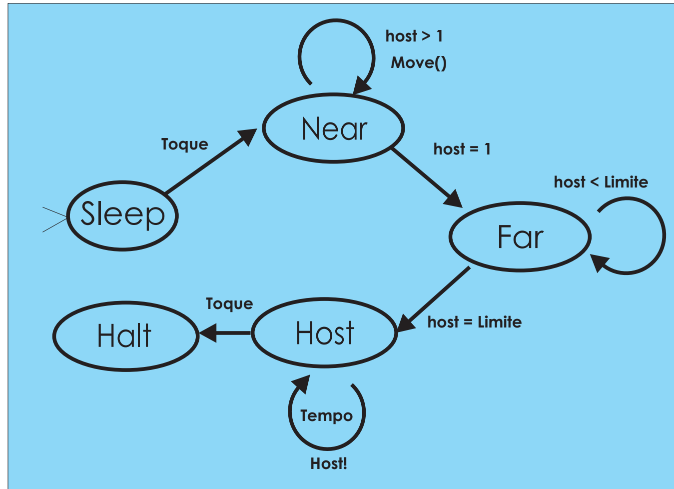
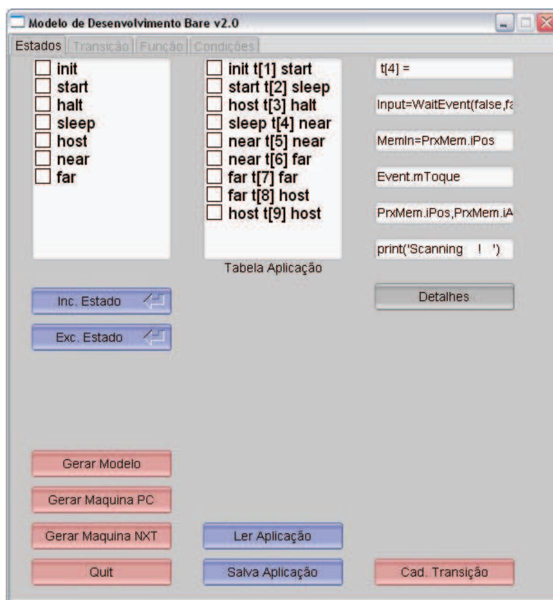


Figura 5.27: Aplicação de Posicionamento de Nós Sensores usando Robô

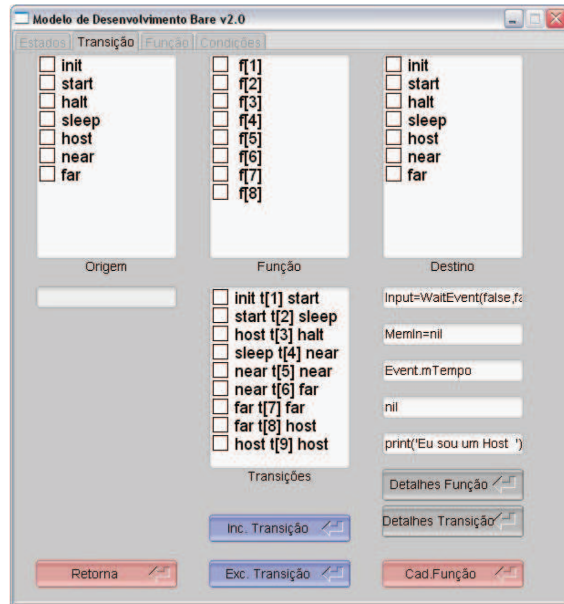
- $\Sigma = (mIntruso = \{Booleano\},\ mTempo = \{Booleano\},\ mToque = \{Booleano\})$
- $\Gamma = (move = \{Int, Int\})$
- $\Lambda = (iPos = \{Int\},\ iAng = \{Int\})$
- $Q \cup \{I, S, H\} = \{Init,\ Start,\ Halt,\ sleep,\ near,\ far,\ host,\ \}$
- $M = (mIntruso,\ mTempo,\ mToque,\ iPos,\ iAng)$
- $E = \{e_1 = mIntruso\ \vee\ e_2 = mTempo,\ \vee\ e_3 = mToque\} = \{e_1 = false \vee e_2 = false \vee e_3 = false\}$
- $C = \{$
  - $[c_0 = true]$
  - $[c_1 = e_1]$
  - $[c_2 = e_2]$
  - $[c_3 = e_3]$
  - $[c_4 = iAng \sim= 0 \wedge e_2]$
  - $[c_5 = iAng == 0 \wedge e_2]$
  - $[c_6 = iPos > 0 \wedge e_2]$
  - $[c_7 = iPos <= 0 \wedge e_2]$

  $\}$

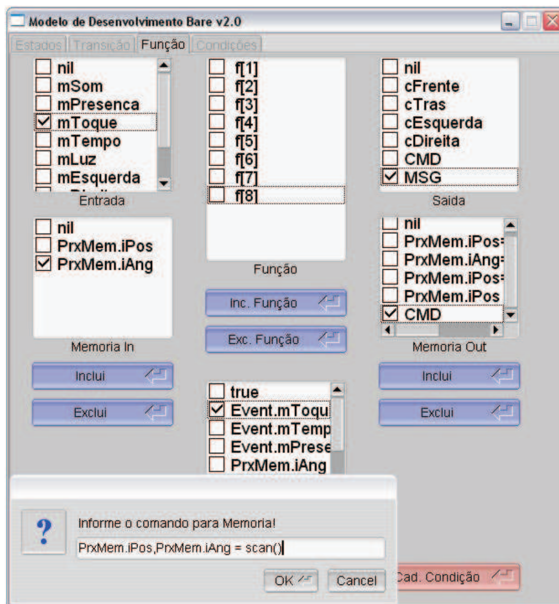
- Φ
  - $\phi_0 = (_, _, c_0, _, _)$
  - $\phi_1 = (mToque,\ \{iPos, iAng\},\ c_3,$
    $\{iScan, iPos = scan()\},\ "Procura\ Hosts!")$
  - $\phi_2 = (mTempo,\ iAng,\ c_4,\ iAng = 0,\ move(3, iAng))$
  - $\phi_3 = (mTempo,\ iPos\},\ c_5,\ _,\ move(0, iPos))$
  - $\phi_4 = (mTempo,\ iPos,\ c_6,\ iPos = iPos - 2,\ move(0, 2))$
  - $\phi_5 = (mTempo,\ iPos,\ c_7, iPos = 0,\ "Definindo\ Host")$
  - $\phi_6 = (mTempo, _, c_2, _, "Sou\ um\ Host!")$
  - $\phi_7 = (mPresenca, _, c_1, _, "Finalizando!")$
  - $\phi_8 = (mToque, _, c_3, _, "Reposicionando!")$
- F
  - $(Init, \phi_0, Start)$
  - $(Start, \phi_0, sleep)$
  - $(sleep, \phi_1, near)$
  - $(near, \phi_2, near)$
  - $(near, \phi_3, far)$
  - (far, ϕ_4, far)
  - $(far, \phi_5, host)$
  - $(host, \phi_6, host)$
  - $(host, \phi_7, halt)$
  - $(host, \phi_8, sleep)$
- $q_0 = \{Init\}$
- $m_0 = (mIntruso = false,\ mTempo = false,\ mToque = false\ iPos = 0,\ iAng = 0)$



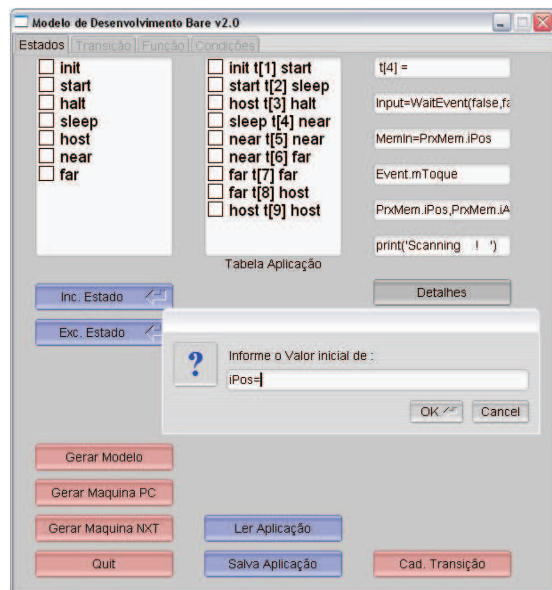
(a) Inclusão de Estados



(b) Cadastro da Transição dos Estados



(c) Cadastro das Funções de Transição



(d) Geração da Tabela da Aplicação

Figura 5.28: Gerador da Aplicação de Posicionamento de Nós com Robô

5.3.3 Gerador da Aplicação de Posicionamento de Nós com Robô

O *GeradorXM* é utilizado para o cadastramento da aplicação no Modelo *Bare*. A Figura 5.28(a) mostra os estados da aplicação de posicionamento de nós com robô. A Figura 5.28(b) apresenta as transições entre os estados.

Toda função possui uma **condição** de avaliação de transição vinculada a ocorrência de um evento. A tela da Figura 5.15(c) mostra como exemplo a criação de uma funções para a aplicação de posicionamento de nós com robô, que atualiza o conteúdo de dois elementos de memória (iPos, iAng), simultaneamente, com o resultado de uma função de interface (scan). Neste exemplo o comando "CMD" é utilizado para disparar um evento que coletar uma informação quando um outro evento de toque é detectado.

Na tela da Figura 5.15(d) mostra a solicitação da definição dos valores iniciais da memória quando a tabela da aplicação é gerada. Toda vez que a tabela da aplicação for gerada, um valor inicial para as variáveis é solicitado, conforme mostra o exemplo da Figura 5.15(d), para a variável "iPos". Em seguida um "Modelo do Sistema" deve ser gerado para ser verificado pela ferramenta NuSMV. O próximo passo é gerar a tabela da aplicação para ser executada pelo ExecutorXM.

5.3.4 Mapeamento para o Modelo Tabular

Depois de concluída a descrição da aplicação o próximo passo é a transformação da especificação da aplicação para o formato tabular por meio do **GeradorXM**. A tabela resultante para a aplicação de *posicionamento de nós sensores* é mostrada na Tabela 5.3. Nela estão resumidos todos os componentes necessários para a execução da aplicação pelo ExecutorXM.

As Tabelas 5.29a e 5.29b apresentam, respectivamente, os eventos monitorados pela aplicação de posicionamento de nós sensores e as condições relacionadas com os eventos que são avaliadas na ocorrência do evento.

5.3.5 Execução da Aplicação de Posicionamento no Robô

Na execução da aplicação de posicionamento de nós sensores os mesmos passos dos exemplos anteriores são executados. A configuração do robô NXT está apresentada na Figura 5.5, os sensores ativos como entrada são os sensores de presença, de som, de toque e um temporizador interno. A interface de saída é o display de LCD e os motores. As rotinas básicas em *pbLua* para a interface são as mesmas. Na implementação do algoritmo no robô foi feita uma adaptação para suprir a falta dos nós sensores reais,

Source	**Input**	**Mem_input**	**Target**	**Condition**	**Mem_output**	**Output**
init	nil	nil	start	c_0	nil	"$Init$"
start	nil	nil	sleep	c_0	$iPos = 0$ $iAng = 0$	"$Start$"
sleep	$mTempo$	$\{iPos, iAng\}$	near	c_3	$iPos, iAng =$ $scan()$	"$Procura\ Hosts!$"
near	$mTempo$	$iAng$	near	c_4	$iAng = 0$	$move(3, iAng)$
near	$mTempo$	$iPos$	far	c_5	_	$move(0, iPos)$
far	$mTempo$	$iPos$	far	c_6	$iPos = iPos - 2$	$move(0, 2)$
far	$mTempo$	$iPos$	host	c_7	_	"$DefinindoHost!$"
host	$mTempo$	_	host	c_2	_	"$Sou\ um\ Host!$"
host	$mToque$	_	sleep	c_3	_	"$Reposicionando!$"
host	$mPresenca$	_	halt	c_1	_	"$Finalizando!$"
halt	nil	nil	halt	nil	nil	nil

Tabela 5.3: Tabela de Execução para Aplicação Posicionamento de Nós com Robô

Memory	**Events**	**Value**
mIntruso	e_1	false
mTempo	e_2	false
mToque	e_3	false
	$\bigvee E$	false

(a) Eventos

Conditions	**Expression**	**Value**
c_1	e_1	false
c_2	e_2	false
c_3	e_3	false
c_4	$iAng \sim= 0 \land e_2$	false
c_5	$iAng == 0 \land e_2$	false
c_6	$iPos > 0 \land e_2$	false
c_7	$iPos <= 0 \land e_2$	false

(b) Condiçoes

Figura 5.29: Tabelas Auxiliares para o Posicionamento de Nós Sensores

que seriam usados para verificar o raio de alcance do host. Para simular a detecção do raio de alcance a rotina de interface de entrada, chamada "scan()", foi modificada para fazer uma varredura de 180 graus do campo de visão do robô e identificar o objeto mais distante no raio de ação, este então passa a ser o alvo do robô representando o host corrente. Na interface de saída é utilizado o display de LCD e os motores, como atuadores em resposta aos eventos detectados. A rotina de interface de saída "move()" foi alterada para avançar com maior precisão.

São descritas a seguir as alterações nas rotinas que executam no Robô NXT:

- scan() (Figura 5.30) Essa rotina executa uma varredura de 180 graus no espaço

à frente do robô procurando pelos objetos simulando os outros nós sensores e o objeto mais distante é considerado o host corrente. Ela produz como resultado dois valores: O primeiro valor indicar a distância do objeto mais afastado do robô. O segundo valor indica o ângulo que o robô deve desviar para se posicionar na direção do objeto. O robô usa a função “move()” para se posicionar na direção do host corrente.

```
function scan()
    local pos=0
    local posAux=0
    local ang=0
    nxt.OutputResetTacho(3,1)
    nxt.OutputSetRegulation(3,1,1)
    setupI2C(1)
    nxt.I2CSendData(1, nxt.I2Ccontinuous, 0)
    pos=lerSonar()
    nxt.OutputSetSpeed(3,0x20,-50,45)
    posAux=lerSonar()
    if posAux > pos then pos = posAux ; ang= -45 end
    nxt.OutputSetSpeed(3,0x20,50,90)
    posAux=lerSonar()
    if posAux > pos then pos = posAux ; ang= 45 end
    nxt.OutputSetSpeed(3,0x20,-50,135)
    posAux=lerSonar()
    if posAux > pos then pos = posAux ; ang= -90 end
    nxt.OutputSetSpeed(3,0x20,50,180)
    posAux=lerSonar()
    if posAux > pos then pos = posAux ; ang= 90 end
    nxt.OutputSetSpeed(3,0x20,-50,90)
    collectgarbage()
    return pos/2,ang
end
```

Figura 5.30: Interface para Detecção do Host Alvo

- move() (Figura 5.31) Esta rotina é uma rotina de interface de saída. Ela recebe como parâmetro a direção e um valor indicando o tipo de movimento. Se a direção for 0 (zero) ou 1(um), indica frente ou trás respectivamente. A alteração em relação a função “move” anterior é que o giro de 360 graus das rodas foi dividido por 10, com isso é possível avançar com precisão de 1/10 do giro completo($\approx 2cm$). Se a direção for 3, indicará um movimento lateral para a esquerda ou direita em graus, a partir da posição frontal do robô. Neste caso o segundo valor informa o tipo de movimento da seguinte forma: se for um valor positivo será um

movimento para a direita, caso seja um valor negativo será um movimento para a esquerda.

```
function move(direcao,voltas)
      local velo=50
      nxt.OutputSetRegulation(1,1,1)
      nxt.OutputSetRegulation(2,1,1)
      if voltas ~=0 then
           if direcao == 0 then nxt.DisplayText("Para Frente ∧ ")
           elseif direcao == 1 then
                velo= -50
                nxt.DisplayText("Para Tras ∨ ")
           end
           if direcao == 0 or direcao == 1 then
                nxt.OutputSetSpeed(1,0x20,velo,voltas*36)
                nxt.OutputSetSpeed(2,0x20,velo,voltas*36)
           elseif direcao == 2 and voltas < 0 then
               nxt.OutputSetSpeed(1,0x20,-velo,nxt.abs(voltas*5))
                nxt.DisplayText("< Esquerda")
           else nxt.OutputSetSpeed(2,0x20,-velo,voltas*5)
                nxt.DisplayText("Direita >")
           end
      end
      Wait(50)
end
```

Figura 5.31: Interface para Atuação com Movimento

A Figura 5.32 mostra o um trecho do código da aplicação de posicionamento de nós sensores que será executada pelo *ExecutoXM* no robô NXT que é transferido no formato de texto para o interpretador *pbLua* via uma conexão USB e armazenado para ser usado como uma tabela para aplicação.

No experimento o robô precisa se movimentar para seguir o host corrente, então a conexão USB não é utilizada, neste caso é utilizado o arquivo padrão (*pbLuaStartup*) que é executado sempre que o robô NXT é ligado e não ocorre uma conexão USB/Bluetooth. A Figura 5.33 mostra o arquivo padrão que é construído e gravado no robô NXT para este experimento.

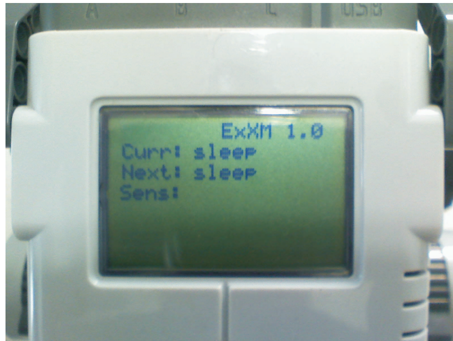
Os comandos são gravados como cadeias de caracteres para o arquivo, desta forma é feita uma concatenação de todos os comandos e a cadeia final concatenada é gravada no arquivos *pbLuaStartup*. As rotinas $EX01$, $EX02$ e $EX04$, são os arquivos com as funções do ExecutorXM, em seguida tem uma chamada para inicializar os contadores dos giro das rodas e a chamada para o ExecutorXM passando como parâmetro a aplicação de posicionamento dos nós sensores.

```
mx = nxt.FileCreate("posicao.nxt", 5122 )
Str=[[$init$ $Event={}; ClearEvent = function()
Event={mToque=false,mTempo=false,mPresenca=false} end$
$Mem = {mToque=false,mTempo=false,mPresenca=false,iPos=0,iAng=0}$
$start$ $true$
$PrxMem={}; mt = { __index = function (PrxMem,k) return Mem[k] end,
__newindex = function (PrxMem,k,v) Event[k]=true; Mem[k] = v end };
setmetatable(PrxMem, mt)$
$Output=nil$ ]]
Str=string.gsub(Str,"\n",")
nxt.FileWrite(mx,Str.."\n")
collectgarbage()

Str=[[$sleep$$Input=WaitEvent(false,false,true,false,false)$
$MemIn=PrxMem.iPos$$near$
$Event.mToque$
$PrxMem.iPos,PrxMem.iAng = scan()$$nxt.DisplayText('Scanning ! ')$ ]]
Str=string.gsub(Str,"\n","")
nxt.FileWrite(mx,Str.."\n")
collectgarbage()

Str=[[$near$$Input=WaitEvent(false,false,false,true,false)$
$MemIn=PrxMem.iAng$$near$
$PrxMem.iAng  = 0 and Event.mTempo$
$PrxMem.iAng=0$$move(3,PrxMem.iAng)$ ]]
Str=string.gsub(Str,"\n","")
nxt.FileWrite(mx,Str.."\n")
collectgarbage()
```

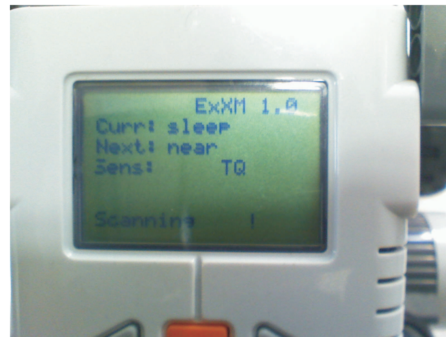
Figura 5.32: Tabela com Trecho da Aplicação do Posicionamento de Nós Sensores

```
programString = [[
nxt.dofile("EX01")
nxt.dofile("EX02")
nxt.dofile("EX04")
nxt.OutputResetTacho(1,1)
nxt.OutputResetTacho(2,1)
nxt.OutputResetTacho(3,1)
Executor("posicao.nxt")
]]

h = nxt.FileCreate( "pbLuaStartup", 512 )

nxt.FileWrite( h, programString )

nxt.FileClose( h )
```

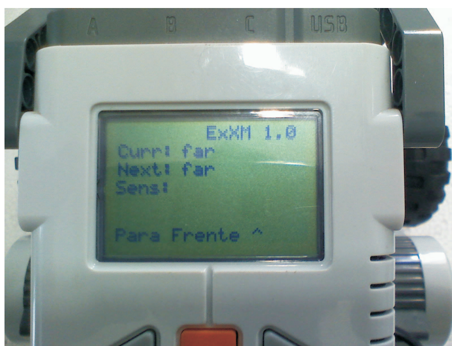
Figura 5.33: Arquivo de Inicialização



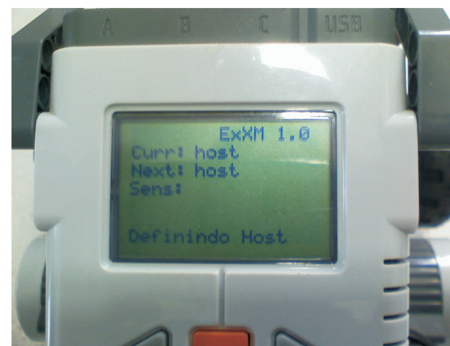
(a) Posicionamento de Nós com Robô - sleep



(b) Posicionamento de Nós com Robô - near



(c) Posicionamento de Nós com Robô - far



(d) Posicionamento de Nós com Robô - host

Figura 5.34: Execução da Aplicação Posicionamento de Nós Sensores usando Robô

Quando a aplicação inicia o display de LCD mostra que o ExecutorXM está carregado e a partir daí começa a execução da aplicação. A aplicação inicia no estado "init", executa as inicializações das tabelas internas (Mem, PrxMem, Event) e, em seguida, passa para o estado "start" e depois passa para o estado "sleep".

A Figura 5.34 (a) mostra a aplicação de posicionamento de nós sensores no estado "sleep". Neste estado apenas um evento é tratado. Quando ocorre o evento de "toque" a aplicação passa para o estado "near" e executa a função "scan()", para fazer uma varredura e identificar o objeto mais distante que será considerado o host corrente.

No estado "near" a finalidade do robô é avançar até alcançar o host corrente o mais próximo possível, o espaço para a aproximação é calculado com base na distância encontrada pela função "scan()". O objetivo do estado "near" é, após identificado a posição do host corrente, se aproximar dele para em seguida se afastar até o seu limite. O avanço é disparado pela passagem do tempo, assim o robô passa para o estado de afastamento "far". A Figura 5.34 (b) mostra a aplicação de posicionamento de nós sensores no estado "near".

No estado “far” o objetivo do robô é se afastar do host corrente até o limite do alcance de comunicação, para isso faz uso da função $move()$ de forma mais precisa, avançando em intervalos menores, da ordem de $\approx 2cm$ por vez a cada intervalo de tempo detectado. Após chegar no limite do alcance de comunicação o robô para e passa para o estado de “host”. A Figura 5.34 (c) mostra a aplicação de posicionamento de nós sensores no estado “far”.

No estado de “host” o robô fixa a sua posição e passa a avisar aos demais nós que ele agora é o host corrente, isso é feito a cada intervalo de tempo. Além do evento de tempo, dois outros eventos são tratados: se o evento de “toque” for detectado o robô termina a execução e passa para o estado “halt”. Caso seja necessário um novo posicionamento do nó sensor, então o robô pode passar para o estado “sleep” se o evento de presença for detectado, reiniciando todo o ciclo. A Figura 5.34 (d) mostra a aplicação de posicionamento de nós sensores no estado “host”.

5.3.6 Verificação para o Modelo do Posicionamento de Nós Sensores

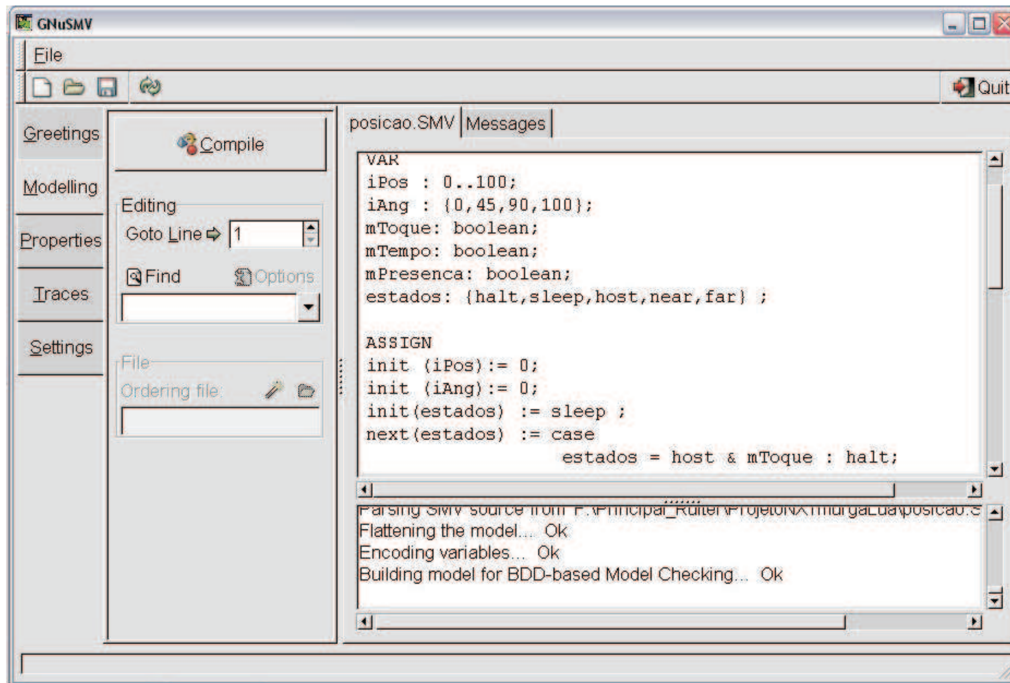
Na Figura 5.35 mostra a saída produzida pelo *GeradorXM* para a aplicação *posicionamento*. Na verificação do Modelo do Sistema para a aplicação de posicionamento é utilizado o verificador de modelos *NuSMV*, versão 2.1. O modelo é carregado e compilado como apresentado na Figura 5.36(a). Para a aplicação de posicionamento de nós sensores são verificadas as seguintes propriedade:

- Toda execução da aplicação de caminhamento autônomo, eventualmente, alcançará o estado “halt”.
- Sempre que o robô for despertado ele se transformará num host.

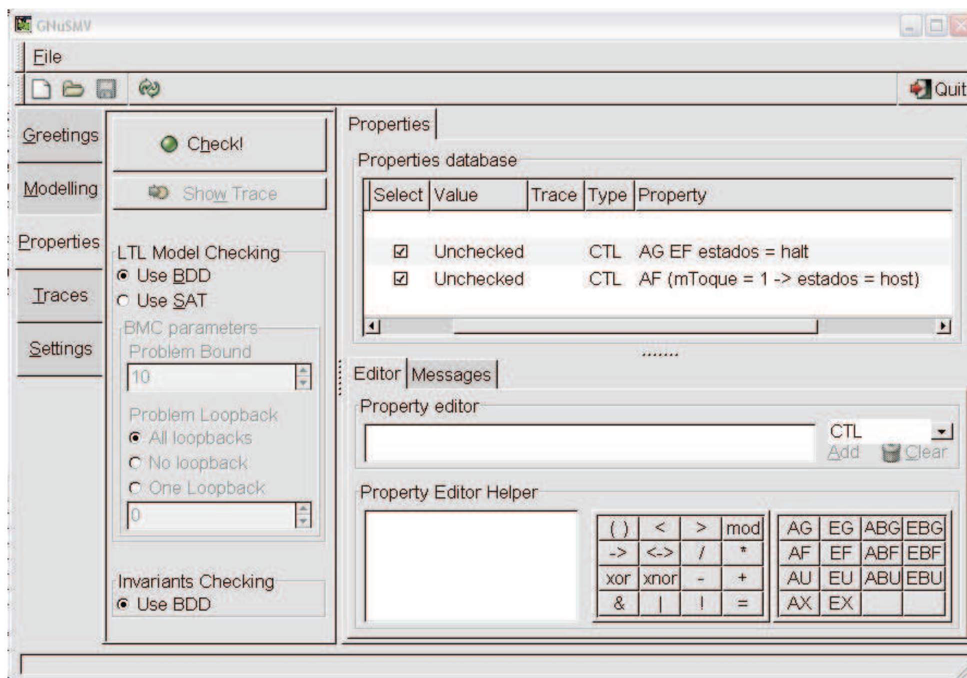
A Figura 5.36(b) mostra as propriedades ainda não verificadas. O resultado da verificação de modelos identifica que a primeira propriedades como verdadeira e a segunda como falsa, como pode ser observado pelo resultado apresentado na Figura 5.37(a). Quando uma propriedade é identificada como falsa, é possível fazer um trace da situação que nega a propriedade. Na Figura 5.37(b) é apresentado um trecho da execução da aplicação de posicionamento de nós sensores que produz um laço que não permite a aplicação evoluir e chegar ao estado de “host”. Ao analisar este caso específico descobrimos que essa situação é contornada, uma vez que o que leva ao laço é a possibilidade do evento tempo não ocorrer, mas na prática o evento tempo sempre ocorre.

```
MODULE main

VAR
iPos : 0..100;
iAng : {0,45,90,100};
mToque: boolean;
mTempo: boolean;
mPresenca: boolean;
estados: {halt,sleep,host,near,far} ;

ASSIGN
init (iPos):= 0;
init (iAng):= 0;
init(estados) := sleep ;
next(estados) := case
                  estados = host & mToque : halt;
                  estados = sleep & mToque : near;
                  estados = near & iAng != 0 & mTempo : near;
                  estados = near & iAng = 0 & mTempo : far;
                  estados = far & iPos > 0 & mTempo : far;
                  estados = far & iPos <= 0 & mTempo : host;
                  estados = host & mTempo : host;
     1 : estados ;
     esac;
next(iPos) := case
                  estados = sleep & mToque :  2..100;
                  estados = far & iPos > 2 & mTempo :  iPos - 2;
                  estados = far & iPos <= 2 & mTempo : 0;
     1 :iPos ;
     esac;
next(iAng) := case
                  estados = sleep & mToque :  {0,45,90,100};
                  estados = near & iAng != 0 & mTempo : 0;
     1 :iAng ;
     esac;

SPEC AG EF estados=halt
SPEC AF (mToque = 1 -> estados = host)
```

Figura 5.35: Modelo do Sistema para Aplicação de Posicionamento de Nós Sensores

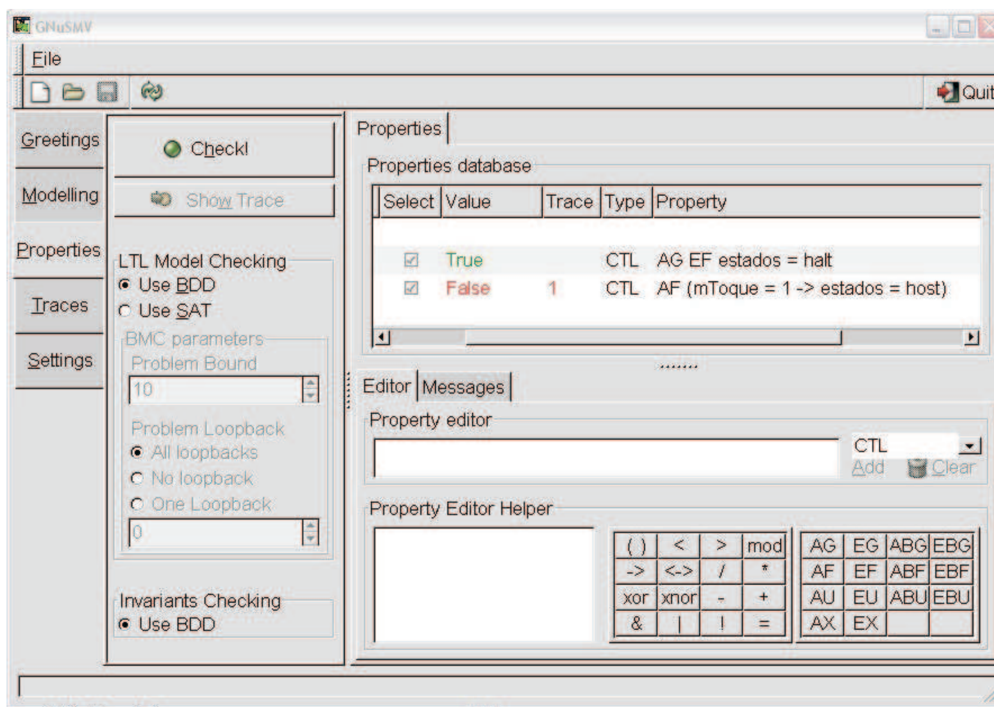


(a) Modelo carregado

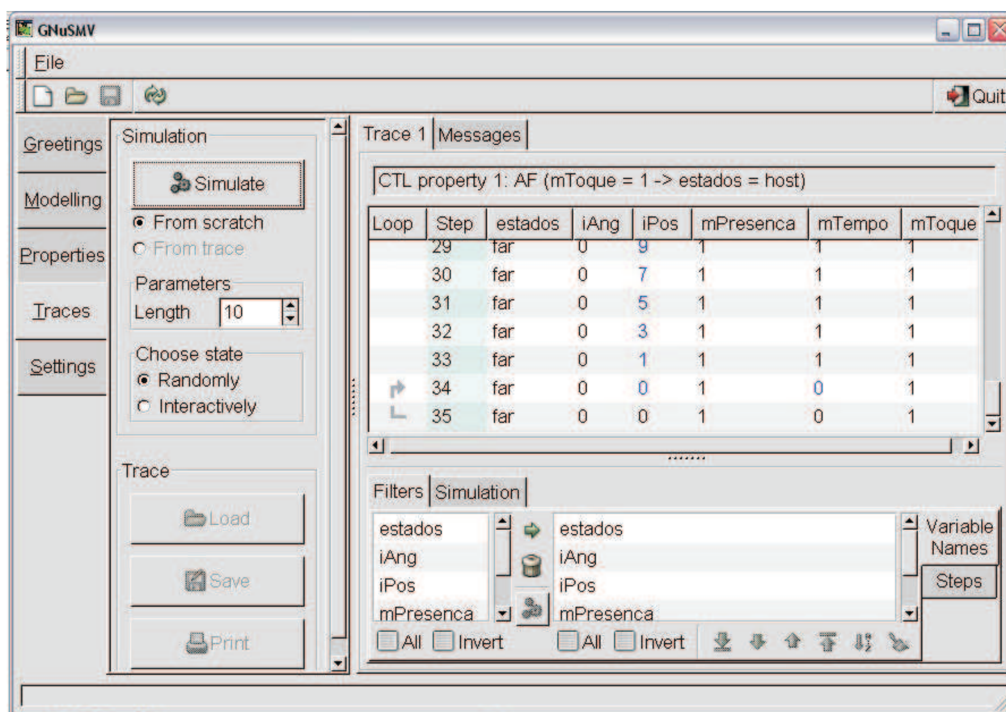


(b) Propriedade não verificada

Figura 5.36: Verificação da Aplicação de Posicionamento de Nós Sensores



(a) Propriedade verificada



(b) Trace do Modelo

Figura 5.37: Verificação da Aplicação de Posicionamento de Nós Sensores

Capítulo 6

Conclusão e Trabalhos Futuros

Este trabalho apresenta o Modelo *Bare* para o Desenvolvimento e Execução das Aplicações em Sistemas Reativos Autônomos, com exemplos de aplicações na área de Rede de Sensores Sem Fio e Robótica. O modelo é chamado "Modelo de Desenvolvimento *Bare*". O Modelo *Bare* inicia com construção da especificação da aplicação num modelo formal de alto nível, chamado *X-machine*. Uma ferramenta chamada **GeradorXM**, é utilizada para construir os detalhes da especificação e conectar os elementos projetados. A mesma ferramenta é utilizada para transformar a especificação da aplicação para um formato *tabular*. A especificação da aplicação, no formato tabular, será executada diretamente no hardware hospedeiro por meio de um interpretador de tabelas, chamado **ExecutorXM**. O ExecutorXM é uma aplicação escrita na linguagem *Lua*, e foi desenvolvida para funcionar com o menor tamanho de código possível. Uma vez que o executor é depositado no hardware não há a necessidade de receber qualquer alteração ou modificação. Sua função é ler as aplicações na forma de tabelas e executá-las.

Uma contribuição deste trabalho é apresentar uma alternativa de solução para o problema de programação e reprogramação dos sistema autônomos em dispositivos que estejam fora do alcance físico, por meio do envio da aplicação num formato simplificado e com uma separação lógica bem definida. A aplicação projetada é transformada no formato de tabela, onde as linhas podem ser enviadas separadamente porque são independentes. Com isso a tabela pode ser transmitida de maneira completa ou em partes, uma linha por vez, sendo que as partes acrescidas podem agregar novas funcionalidades ou substituir as funções originais da aplicação. A aplicação passa a ter uma estrutura de execução dinâmica que pode ser reconfigurada sempre que houver a necessidade.

A programação de aplicações em Redes de Sensores Sem Fio ainda é uma das maiores dificuldades para a adoção mais ampla dessa tecnologia. Nas aplicações no mundo real a programação é feita num nível muito próximo ao hardware e requer do pro-

gramador trabalhar com detalhes de baixo nível, desviando o foco do desenvolvimento da lógica da aplicação. Isso demanda a necessidade de abstrações de programação de alto nível, capaz de simplificar a programação sem perder a eficiência. A capacidade de manipular a aplicação num nível mais alto é muito útil na área de RSSF, onde a quantidade de nós utilizados é muito grande (da ordem de milhares) e depois de lançados torna-se inviável ou a um custo financeiro muito elevado fazer a reprogramação dos nós sensores. A utilização do Modelo *Bare* na área de RSSF é viabilizado da seguinte forma: cada nós sensores é carregado com o código do ExecutorXM uma única vez antes de ser feita a disposição dos nós sensores na área a ser monitorada. Com relação ao carregamento das aplicações que serão executadas no nó sensor, ou programação do nó, poderá ser feito das seguintes formas: 1) a aplicação é carregada no nó sensor junto com o executor, 2) a aplicação pode ser enviada após a disposição do nó sensor de uma única vez ou 3) pode ser enviada em pequenos trechos de cada vez, fazendo uso racional da banda da rede. No decorrer do processo de execução da aplicação no nó sensor, pode surgir a necessidade de realizar uma atualização da aplicação, chamado de reprogramação. A reprogramação pode ser realizada com a substituição dos trechos antigos da aplicação ou pelo acréscimo de novas funções (acréscimo de novas linhas) para as tabelas da aplicação.

O Modelo de Desenvolvimento de Execução *Bare* torna possível executar uma troca de contextos das aplicações de forma controlada de maneira análoga ao modelo da reconfiguração de hardware, o qual foi denominado de “Reconfiguração de Aplicação”. O Modelo *Bare* implementa as características de reconfiguração dinâmica de contexto múltiplo, onde cada tabela da aplicação representa um cenário diferenciado com o seu contexto próprio, sendo que ao substituir uma tabela de aplicação corrente por outra temos um novo contexto, similar a reconfiguração de hardware. O modelo também permite implementar a reconfiguração parcial, onde as linhas da tabela da aplicação podem ser substituídas independentemente umas das outras durante a execução da aplicação, uma vez que no estado corrente apenas as linhas que representam esse estado estão sendo manipuladas pelo executor, com isso as linhas dos outros estados podem ser alteradas, ou seja reconfiguradas, sem interferir com o processo de execução da aplicação.

A capacidade de reprogramar um dispositivo a distância sem ter que realizar o desligamento completo do dispositivo também é muito útil na Robótica, para realizar a reprogramação do robô em missão num terreno hostil ou com restrição de acesso, como no caso do robô explorador *Sojourner* usado na Missão “Mars Pathfinder”. Alguns dispositivos eletrônicos que executam missão crítica podem ser beneficiados com a possibilidade de reprogramação em tempo de execução. Na área Tecnologia da Informação,

missão crítica é a expressão utilizada para descrever o conceito de aplicações, serviços e processos com alta disponibilidade, cuja paralisação ou perda de dados importantes podem gerar grandes transtornos, não apenas econômicos e operacionais, mas também sociais, tanto para grandes corporações, como para pequenas empresas.

Como resultado final deste trabalho de pesquisa foi criado o Modelo de Desenvolvimento *Bare* para o desenvolvimento de Aplicações em Sistemas Reativos Autônomos, com aplicações na área de Redes de Sensores Sem Fio e Robótica, usando um modelo formal, que resulta na geração automática dos código para execução, juntamente com um ambiente de execução. Por meio deste modelo é proposto a solução para o problema da programação e reprogramação das aplicações em hardware com difícil acesso por meio de um canal de comunicação com baixa taxa de velocidade usando o modelo de reconfiguração de aplicação.

Uma outra contribuição do trabalho é a geração automática do "Modelo do Sistema" a partir da tabela da aplicação. O modelo gerado é escrito na linguagem da ferramenta de verificação formal de modelos (Model Checking) "NuSMV". As máquinas de estados finitos são usadas muitas vezes como uma ferramenta para a construção de modelos dos sistemas, nessas máquinas os estados guardam informações sobre os valores correntes das variáveis e as transições descrevem como os estados fontes evoluem para os estados destino. Uma vez que a transformação da especificação da aplicação de X-machine para o modelo tabular mantem a relação entre os estados e as transições que habilitam a evolução da aplicação, torna-se quase imediata a extração do Modelo do Sistema. Essa abordagem tem uma vantagem que o modelo, que descreve o comportamento do sistema, é um modelo real do sistema, portanto é uma descrição precisa e sem ambiguidade.

Para a tarefa de verificação formal da aplicação as propriedades desejadas para a aplicação devem ser definidas usando a lógica temporal CTL. Com a lógica temporal CTL é possível especificar uma ampla variedade de propriedades relevantes para a aplicação, tais como correção funcional ("a aplicação faz o que deve fazer?"), safety ("uma coisa indesejada nunca ocorre"), liveness ("uma coisa desejada eventualmente ocorre"), reachability ("uma determinada situação pode ser atingida?"). Apesar da definição das propriedades que a aplicação deve satisfazer, ser parte dos requisitos da aplicação, e a sua definição ser uma tarefa de responsabilidade do desenvolvedor da aplicação, existem algumas regras gerais que auxiliam na descrição das propriedades que são geradas automaticamente junto com o Modelo do Sistema.

Duas outras contribuição foram alcançadas no desenvolvimento deste trabalho. A primeira delas diz respeito a definição de um Modelo Dinâmico de Execução para as X-machines. Por ser uma entidade matemática abstrata utilizada apenas para modelar a

especificação e as características dos sistemas, não existe ainda um mapeamento direto entre a maquina abstrata e uma implementação eficiente. Temos conhecimento na literatura de duas implementações para X-machine, uma usando a linguagem Prolog outra usando a linguagem Java, porém ambas fazem uma simulação da execução do código. Com a inclusão dos *Eventos* e as *Condições* no modelo é possível construir um mecanismo para executar a especificação diretamente numa máquina real.

A segunda contribuição é com relação a programação de aplicações sem a utilização de um sistema operacional dedicado. Para executar as aplicações é necessário somente o *ExecutorXM*, que está escrito na linguagem *Lua* e, neste caso, precisa do interpretador *Lua* para o hardware alvo. Em outubro deste ano foi apresentado na PUC/Rj, no Lua Workshop 2009, o projeto *Embedded Lua - eLua*[eLua, 2009], que tem por objetivo portar a linguagem *Lua* para dispositivos embarcados. Com a portabilidade da linguagem *Lua* o *ExecutorXM* e consequentemente o Modelo *Bare* poderá executar nas seguintes plataforma de microcontroladores: ARMCortex-M3, ARM7TDMI RISC, ARM966E-S 16/32, AVR32-UCcore, ARM-based 32-bit. Viabilizando desta forma a aplicação deste modelo na solução em sistemas embarcados de maneira mais geral.

A adoção do Modelo de Desenvolvimento *Bare* para o desenvolvimento das aplicações traz os seguintes benefícios para a área de RSSF:

- Simplificação do mecanismo de execução das aplicações em RSSF, pois o único código executável fixo é o do *ExecutorXM*, o qual é depositado nos nó sensor previamente e não precisa receber nenhuma alteração para a execução de qualquer nova aplicação.

- A programação e, consequentemente a reprogramação, das aplicações em RSSF fica simplificada, uma vez que o formato tabular, como que as aplicações são manipuladas, torna possível o envio de trechos da aplicação. Isso permite um uso mais racional da baixa taxa de transferência entre os nós.

- O papel de alguns nós sensores pode ser especializado, por meio do envio de algumas linhas extras para modificar a aplicação que produzirá um comportamento diferente. Isso possibilita a migração desses papeis para outros nós, por meio do envio das partes especializadas da aplicação (linha modificadas).

- É possível armazenar várias tabelas de diferentes aplicações nos nó sensores, de tal maneira que estar serão ativadas em resposta a situações críticas. Essa característica é conhecida como reconfiguração dinâmica com contexto múltiplo parcialmente reconfigurável.

- Definição das aplicações em RSSF como parte da classe de aplicações de Sistemas Reativos Autônomos.

Benefício do Modelo de Desenvolvimento *Bare* para o desenvolvimento das aplicações de maneira geral:

- Utilização de um modelo formal para a especificação das partes componentes das aplicações dos Sistemas Reativos Autônomos.
- Geração automática de código a partir do modelo formal da aplicação.
- Modelo simples para programação das aplicação baseado em tabelas de alto nível com instruções de execução.
- Utilização de um interpretador de tamanho mínimo para executar as tabelas com as aplicações.
- Capacidade de evoluir a aplicação em tempo de execução.
- Utilização da especificação formal das aplicações como entrada para um sistema de verificação formal de modelos.
- O Modelos do Sistema é uma descrição precisa do sistema real e não um protótipo.
- O *ExecutorXM* está implementado na linguagem de programação *LUA* e executa num hardware real.
- Definição de um Modelo Dinâmico para a executar o modelo abstrato da X-Machine.
- Programação de Sistemas Reativos sem a utilização de um Sistema Operacional.

6.1 Trabalhos Futuros

Apesar de uma grande parte do trabalho está pronto, é possível realizar mais experimentos para avançar na construção desta solução. Durante a execução do trabalho aparecem mais dúvidas que respostas e são listadas a seguir alguns dos desdobramentos para trabalhos futuros:

- Comparação do Modelo *Bare* com outros trabalhos relacionados. Como por exemplo com o *Maté* e o *Impala*, comparando características como Modularidade, Correção, Facilidade de Atualização, Eficiência de Energia e possibilidade de execução em hardwares de nós sensores reais.

- Descrição do *Sensor* e do *Atuador* por meio de *X-machines*. Por exemplo, assumindo que o protocolo de comunicação "A" é um protocolo muito seletivo que usa uma faixa curta do radio para transmitir para um vizinho próximo, enquanto o protocolo "B" é um protocolo mais indiscriminado de difusão, que usa uma faixa longa para transmitir. O protocolo "A" consome menos energia e gera menos tráfego na rede, porém só pode trabalhar quando conhece a existência do outro nó, para utilizar essa transmissão com razoável frequência. Por outro lado quando um nó sensor está isolado numa posição remota, mas tem energia suficiente para gastar ele pode usar o protocolo "B", para se conectar com outros nós de maneira mais eficiente. A decisão de utilizar um ou outro pode ser programada por meio de uma aplicação projetada como uma *X-machines*, utilizando o mecanismo de reconfiguração da aplicação considerando a descoberta de vizinhos próximos ou não. Um outra proposta seria a substituição do protocolo de comunicação em intervalos de tempo.

- A construção de um código para o ExecutorXM nativo em linguagem *assembler* ou na linguagem *C*, possível de executar no nó sensor real. Neste trabalho foi utilizada a linguagem *Lua* para implementar o código do executor, um caminho alternativo seria implementar o executor diretamente no nó sensor.

- Na alteração dinâmica da aplicação não há garantias que a inclusão ou exclusão das linhas da tabela mantenha a aplicação consistente e sem a presença de erros. Uma solução para o problema seria fazer uma verificação de consistência mínima na tabela da aplicação antes de executá-la.

- O modelo de reconfiguração de aplicação deve permitir a troca de contexto entre as aplicações, para que o nó execute uma nova aplicação quando for detectada uma situação crítica. Uma opção é utilizar o mesmo esquema de troca de estado para executar a troca de contexto. Estabelecendo uma função de transição chamada "reconf" que fará com que, na passagem de um estado para outro, seja executada o carregamento da nova aplicação implicando na troca de contexto.

- A aplicação "GeradorXM" apesar de grande utilidade para o projeto pode ser melhorada com a inclusão de uma interface gráfica baseada em máquinas de estados, o que permitirá a construção da X-machine de maneira mais natural.

- Acrescentar na aplicação "GeradorXM" um mecanismo para sugestão de propriedades para o "Modelo do Sistema". Dotar um conjunto de propriedades padrão

adaptável a estrutura da aplicação gerada, que forneça sugestão sobre as propriedades desejáveis para a aplicação.

- Incrementar a modelagem por meio da inclusão no modelo de especificação e geração para X-machine com comunicação, e mecanismos de abstração como composição e hierarquia. Para que aplicações mais complexas possam ser desenvolvidas.

Referências Bibliográficas

Abdelzaher, T.; Blum, B.; Cao, Q.; Chen, Y.; Evans, D.; George, J.; George, S.; Gu, L.; He, T.; Krishnamurthy, S.; Luo, L.; Son, S.; Stankovic, J.; Stoleru, R. & Wood, A. (2004). Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. *24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 582–589.

Arampatzis, T.; Lygeros, J. & Manesis, S. (2005). A survey of applications of wireless sensors and wireless sensor networks. *In Proc. of the Mediterranean Control Conference (Med05), Limassol Cyprus, 27-29 June.*

Baier, C. & Katoen, J.-P. (2008). *Principles of Model Checking.* The MIT Press.

Barnard, J.; Whitworth, J. & Woodward, M. (1996). Communicating x-machines. *Information and Software Technology*, 38:401–407.

Berry, G. & Gonthier, G. (1992). The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87--152.

Breen, M. (2005). Experience of using a lightweight formal specification method for a commercial embedded system product line. *Requirements Engineering Journal*, 10:161--172.

Caldas, R. B.; Corrêa, F. L.; Nacif, J. A.; Roque, T. R.; Ruiz, L. B.; da Mata, J. M.; Fernandes, A. O. & Junior, C. J. N. C. (19-22 Sept. 2005a). Low power/high performance self-adapting sensor node architecture. *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, 2:4 pp.

Caldas, R. B.; Corrêa, F. L.; Nacif, J. A.; Roque, T. R.; Ruiz, L. B.; Fernandes, A. O. & Junior, C. J. N. C. (2005b). Reconfigurable sensor node for low power-high performance applications. *In Proceedings of the 13th IFIP International Conference on Very Large Scale Integration Systems (VLSI-SOC 2005).*

Chang, C.-L. & Lee, R. C. (1973). *Symbolic Logic and Mechanical Theorem Proving*. Computer Science Classics. Academic Press.

Chang, E.; Manna, Z. & Pnueli, A. (1991). The safety-progress classification. *Logic and Algebra of Specifications (F.L. Bauer, W. Brauer, and H. Schwichtenberg, eds.)*, NATO Advanced Science Institutes Series:143–202.

Cimatti, A.; Clarke, E. M.; Giunchiglia, F. & Roveri, M. (1999). Nusmv: A new symbolic model verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pp. 495--499, London, UK. Springer-Verlag.

Clarke, E. M. & Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pp. 52--71, London, UK. Springer-Verlag.

Clarke, E. M.; Emerson, E. A. & Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244--263.

Compton, K. & Hauck, S. (2002). Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171--210.

Dwyer, M. B.; Avrunin, G. S. & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pp. 411--420, New York, NY, USA. ACM.

Edmund M. Clarke, J.; Grumberg, O. & Peled, D. A. (2000). *Model Checking*. The MIT Press.

Eilenberg, S. (1974). *Automata, Languages, and Machines*. Academic Press, Inc., Orlando, FL, USA.

Eleftherakis, G. (2003). *Towards Formal Development of Computer-Based Systems*. PhD thesis, University of Sheffield.

eLua (2009). Disponível em: <http://elua.berlios.de/en_overview.html>. [Acessada em: 15-Novembro-2009].

Ferreira, N. F. G. (2005). Verificação formal de sistemas modelados em estados finitos. Master's thesis, Escola Politécnica da Universidade de São Paulo.

Georgescu, H. & Vertan, C. (2000). A new approach to communicating x-machines systems. *Journal of Universal Computer Science*, 6(5):490--502.

Glaser, S. (2004). Some real-word applictaions of wireless sensor nodes. In *SPIE Symposium on Smart Structure & Material/NDE 2004*.

Glinz, M. (2000). Improving the quality of requirements with scenarios. In *Second World Congress for Software Quality*, pp. 55–60. unknown.

Halbwachs, N. (1998). Synchronous programming of reactive systems - a tutorial and commented bibliography. In *In Tenth International Conference on Computer-Aided Verification, CAVŠ98, Vancouver (B.C.), LNCS 1427*, pp. 1--16. Springer Verlag.

Harel, D. & Pnueli, A. (1985). On the development of reactive systems. *NATO ASI Series - Logics and Models of Concurrent Systems*, pp. 477--498.

Heitmeyer, C. L.; Jeffords, R. D. & Labaw, B. G. (1996). Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231--261.

Holcombe, M. (1988). X-machines as a basis for dynamic system specification. *Software Engineering Journal*, pp. 69–76.

Howard, A.; Mataric, M. J. & Sukhatme, G. S. (2002). Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem. pp. 299--308.

Ierusalimschy, R.; de Figueiredo, L. H. & Filho, W. C. (1996). Lua an extensible extension language. *Softw. Pract. Exper.*, 26(6):635--652.

Janicki, R. & Khedri, R. (2001). On a formal semantics of tabular expressions. *Science of Computer Programming*, 39(2–3):189--213.

Lamport, L. (1977). Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, SE-3(2):125–143.

Laycock, G. T. (1993). *The Theory and Practice of Specification Based Software Testing*. PhD thesis, University of Sheffield.

Lego NXT (2009). Disponível em: <www.mindstorms.lego.com>. [Acessada em: 10-Agosto-2009].

Loer, K. (2003). Model-based automated analysis for dependable interactive systems. Technical report, University of York.

Lua (2009). Disponível em: <http://www.lua.org>. [Acessada em: 02-Maio-2009].

McMillan, K. L. (1993). *Symbolic Model Checking.* Kluwer Academic Publishers.

Mottola, L. & Picco, G. P. (2008). Programming wireless sensor networks: Fundamental concepts and state of the art. accepted for publication on ACM Computing Surveys.

murgaLua (2009). Disponível em: <http://www.murga-projects.com/murgaLua/>. [Acessada em: 12-Junho-2009].

Newton, R. & Welsh, M. (2004). Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pp. 78--87, New York, NY, USA. ACM Press.

Parker, L. E.; Kannan, B.; Fu, X. & Tang, Y. (2003). Heterogeneous mobile sensor net deployment using robot herding and line-of-sight formations. In *Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems Las Vegas, Nevada.*

Payton, D.; Estkowski, R. & Howard, M. (2001). Compound behaviors in pheromone robotics. *Robotics and Autonomous Systems*, 44:229--240.

PbLua (2009). Disponível em: <http://www.hempeldesigngroup.com/lego/pbLua/>. [Acessada em: 10-Agosto-2009].

Pnueli, A. (1977). The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46--57, Washington, DC, USA. IEEE Computer Society.

Quielle, J.-P. & Sifakis, J. (1982). Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pp. 337--351, London, UK. Springer-Verlag.

Romer, K. & Mattern, F. (2004). The design space of wireless sensor networks. *Wireless Communications, IEEE*, vol.11, no.6:54– 61.

Rosson, M. B. & Carroll, J. M. (2001). *Usability Engineering: Scenario-Based Development of Human Computer Interaction.* Morgan Kaufmann Publishers.

Ruiz, L. B. (2003). *MANNA: A Management Architecture for Wireless Sensor Networks.* PhD thesis, Computer Science Department of the Federal University of Minas Gerais, Belo Horizonte, MG, Brazil.

Ryser, J. & Glinz, M. (1999). A practical approach to validating and testing software systems using scenarios. In *QWE Š99, 3 rd International Software Quality Week Europe*.

Sitharama, K. C.; Chakrabarty, K.; Iyengar, S. S.; Qi, H. & Cho, E. (2001). Coding theory framework for target location in distributed sensor networks. In *in International Symposium on Information Technology: Coding and Computing, 2001*, pp. 130--134.

Skliarova, I. (2004). *Reconfigurable architectures for problems of combinatorial optimization*. PhD thesis, University of Aveiro.

Toscani, S. S. (1993). *RS: Uma Linguagem para Programação de Núcleos Reactivos*. PhD thesis, Universidade Nova de Lisboa.

Tubaishat, M. & Madria, S. (2003). Sensor networks: an overview. *Potentials, IEEE*, 22(2):20--23.

Welsh, M. & Mainland, G. (2004). Programming sensor networks using abstract regions. *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*.