

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

**SILVANA MADEIRA ALVES DAL-BÓ**

**Um Ambiente Baseado em Componentes para  
Desenvolvimento de *Software* de Sistemas Embutidos**

Luis Fernando Friedrich

Florianópolis, Julho de 2004.

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

**SILVANA MADEIRA ALVES DAL-BÓ**

**Um Ambiente Baseado em Componentes para  
Desenvolvimento de *Software* de Sistemas Embutidos**

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

Luis Fernando Friedrich

Florianópolis, Julho de 2004.

# Um Ambiente Baseado em Componentes para Desenvolvimento de *Software* de Sistemas Embutidos

**SILVANA MADEIRA ALVES DAL-BÓ**

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração (Computação Paralela e Distribuída) e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

---

Luis Fernando Friedrich, Dr.  
Orientador

---

Raul S. Wazlawick, Dr.  
Coordenador do Programa de Pós-Graduação em Ciência da Computação

Banca Examinadora:

---

Prof. Ricardo Villarroel Dávalos

---

Carlos Barros Montez

---

José Mazzucco Júnior

---

Luiz Cláudio Villar dos Santos

*Dedico este trabalho à minha filha **Giovanna**,  
que nasceu com vida e saúde, onde encontrei  
forças para continuar meu trabalho.*

## AGRADECIMENTOS

Agradeço a Deus, pela oportunidade, pela fé e pelas pessoas que colocou em meu caminho. Pessoas estas que contribuíram de alguma forma para que aflorasse a dedicação e o amor pela pesquisa e que eu jamais desistisse na conclusão deste trabalho.

Como não poderia citar a todos, deixo aqui a minha gratidão a todos que fazem parte de minha vida e que de alguma maneira me ajudaram, com idéias, com atitudes, com uma palavra de apoio e com seu conhecimento. Porém não poderia deixar de citar meu agradecimento especial ao meu Orientador, Luis Fernando Friedrich, por sua paciência, pelos seus ensinamentos e todo seu apoio.

Aos alunos Charles Konig e Michel Silveira, pela contribuição na implementação da Ferramenta proposta neste trabalho.

Sou eternamente grata a minha querida amiga e cunhada Marlize Dal-Bó de Oliveira, que muito me ajudou, muitas vezes, substituindo no meu papel de mãe, quando não pude estar presente com meus filhos, durante as viagens de aula do mestrado.

Aos meus amigos, pessoas maravilhosas presentes em minha vida, muito mais que amigos... pelas palavras certas, pelo apoio e amizade.

Meu profundo agradecimento ao meu querido marido, Giovani Dal-Bó, por todo seu apoio e amor dedicado a mim e aos nossos filhos, Bruno e Giovanna.

Aos meus pais, Miguel Fernandes Alves e Santina Madeira Alves, que sempre me incentivaram profissionalmente, me apoiaram e me auxiliaram a superar todos os obstáculos de minha caminhada.

## LISTA DE ABREVIATURAS

|       |   |
|-------|---|
| ASCII | <i>American Standard Code for Information Interchange</i>           |
| CDL   | <i>Component Description Language</i>                               |
| CISC  | <i>Complex Instruction Set Computer</i>                             |
| CPU   | <i>Central Processor Unit</i>                                       |
| DE    | <i>Discrete Event</i>   |
| DEC   | <i>Descrição de Componente</i>                                      |
| ECOS  | <i>Embedded Configurable Operating System</i>                       |
| EPROM | <i>Erasable PROM</i>  |
| GPOS  | <i>General Purpose Operational Systems</i>                          |
| GUI   | <i>Graphic User Interface</i>                                       |
| IDL   | <i>Interface Definition Language</i>                                |
| IEEE  | <i>Institute of Electrical and Electronic Engineers</i>             |
| ITU   | <i>International Telecommunication Union</i>                        |
| MOC   | <i>Model of Computation</i>   |
| NFA   | <i>Nondeterministic Finite Automata</i>                             |
| OMG   | <i>Object Management Group</i>                                      |
| OMT   | <i>Object Modeling Technique</i>                                    |
| OO    | <i>Orientação a Objetos</i>   |
| OOA   | <i>Object Oriented Analysis</i>                                     |
| PCB   | <i>Process Control Block</i>  |
| PCB   | <i>Process Control Block</i>  |
| PROM  | <i>Programable ROM</i>  |
| RAM   | <i>Random Access Memory</i>   |
| RISC  | <i>Reduced Instruction Set Computer</i>                             |
| ROM   | <i>Read Only Memory</i>   |
| RTOS  | <i>Real Time Operation System</i>                                   |
| SDL   | <i>Specification and Description Language</i>                       |
| SFSM  | <i>Synchronous Finite State Machine</i>                             |
| UART  | <i>Universal Asynchronous Receiver</i>                              |
| UCP   | <i>Unidade Central de Processamento (tradução de CPU)</i>           |
| UML   | <i>Unified Modeling Language (Linguagem de modelagem Unificada)</i> |
| UML   | <i>Unified Modeling Language</i>                                    |
| VHDL  | <i>VHSIC Hardware Description Language</i>                          |
| VHSIC | <i>Very High Speed Integrated Circuit</i>                           |

## LISTA DE FIGURAS

|  |    |
|--|----|
| FIGURA 1 – Retorno Financeiro e Janelas de Tempo.....  | 06 |
| FIGURA 2 – Estrutura de software em camadas .....  | 11 |
| FIGURA 3 - Principais Níveis de Abstração no Processo de Projeto (Wolf, 2000) .....          | 13 |
| FIGURA 4 – Estrutura Hierárquica SDL .....   | 22 |
| FIGURA 5 - Design Utilizando SystemC.....  | 25 |
| FIGURA 6 – Notação do Diagrama de Classes.....   | 28 |
| FIGURA 7 – Notação dos Diagramas de Interação .....  | 29 |
| FIGURA 8 – Notação dos Diagramas de Estados.....   | 31 |
| FIGURA 9 – Plataforma de Tv no Modelo Koala .....  | 33 |
| FIGURA 10 – Ligação Koala .....  | 36 |
| FIGURA 11 - Exemplo de Camadas para um Sistema Embutido no eCos .....                        | 39 |
| FIGURA 12 – Interface da ferramenta de configuração (eCos, 2003) .....                       | 41 |
| FIGURA 13 – Arquitetura da Ferramenta Sigae .....  | 48 |
| FIGURA 14 – Estrutura de Diretório do Componente SO Sigae .....                              | 52 |
| FIGURA 15 – Estrutura do Interpretador LDEC .....  | 63 |
| FIGURA 16 - Diagrama de Atividades – desenvolvimento do <i>software</i> usando o Sigae ..... | 69 |
| FIGURA 17 - Diagrama de Classes da Ferramenta .....  | 70 |
| FIGURA 18 – Plataforma de <i>Hardware</i> .....  | 80 |
| FIGURA 19 – Janela Inicial da ferramenta .....   | 83 |
| FIGURA 20 – Janela de Arquitetura .....  | 84 |
| FIGURA 21 – Propriedades do Projeto .....  | 85 |
| FIGURA 22 – Componentes do Repositório .....   | 85 |
| FIGURA 23 – Janela de Edição das ligações dos componentes da aplicação .....                 | 86 |
| FIGURA 24 – Janela de descrição do projeto de <i>software</i> .....                          | 87 |
| FIGURA 25 – Janela da Linguagem de Descrição de Componente .....                             | 88 |
| FIGURA 36 – Janela de edição do <i>software</i> .....  | 89 |
| FIGURA 47 – Janela de especificação do compilador .....                                      | 90 |
| FIGURA 28 – Janela de inserção e descrição do componente para o repositório .....            | 91 |
| FIGURA 29 – Removendo um componente do repositório .....                                     | 92 |

|   |            |
|---|------------|
| <b>FIGURA 30 – Janela de Compilação .....</b>   | <b>93</b>  |
| <b>FIGURA 31 – Teste do código - Estudo de Caso Relógio .....</b>                         | <b>99</b>  |
| <b>FIGURA 32 – Janela gráfica de visualização dos componentes – estudo de caso 2.....</b> | <b>100</b> |
| <b>FIGURA 33 – Caixa de Listagem de Operações dos componentes – estudo de caso 2.....</b> | <b>101</b> |
| <b>FIGURA 54 – Teste do código - estudo de caso forno microondas .....</b>                | <b>104</b> |



## LISTA DE QUADROS

|   |     |
|---|-----|
| QUADRO 1 – Trecho Código System Calls .....                                   | 53  |
| QUADRO 2 – Trecho Código Kernel .....   | 55  |
| QUADRO 3 – Modelo de Descrição do Componente .....                            | 63  |
| QUADRO 4 – Exemplo arquivo makefile .....                                     | 68  |
| QUADRO 5 – Classe Projeto .....   | 72  |
| QUADRO 6 – Classe Componente .....  | 74  |
| QUADRO 7 – Classe Gerenciador .....   | 76  |
| QUADRO 8 – Classe Compilador .....  | 77  |
| QUADRO 9 – Código fonte gerado pela Ferramenta - Estudo de Caso Relógio ..... | 97  |
| QUADRO 10 - Arquivo DEC do projeto Clock .....                                | 98  |
| QUADRO 11 – Arquivo makefile – estudo de caso 2 .....                         | 103 |

## RESUMO

Sistemas Embutidos são sistemas direcionados a uma aplicação específica, com alta integração de seus componentes de *hardware* e *software* e uma interface de comunicação com o ambiente externo específica de cada aplicação. O perfil de desenvolvimento do projeto de um sistema embutido (*embedded system*) implica em uma metodologia que envolva tanto os componentes de *software* quanto os componentes de *hardware*, levando-se em conta especialmente o tempo e o custo de um projeto. Neste sentido, a especificação de uma ferramenta capaz de auxiliar na metodologia e desenvolvimento destes sistemas, vem com o objetivo de diminuir o tempo de desenvolvimento de um projeto e a reutilização de componentes de *software* e aplicações já desenvolvidas. Esta dissertação apresenta um estudo sobre metodologias de projeto para sistemas embutidos, linguagens de descrição de componentes e ferramentas de projeto existentes atualmente e finalmente um modelo proposto para uma ferramenta de *software* baseada em componentes. A ferramenta dispõe de um ambiente onde cada componente de *software* é descrito em uma linguagem aqui denominada *LDEC* (linguagem de descrição de componentes), própria da ferramenta e, inserido em um repositório de componentes. Os componentes presentes no repositório podem ser selecionados, através de uma interface gráfica, de modo a compor um *software* de aplicação embutida. Uma aplicação ou componente, podem ser reutilizados, desde que estejam descritos e inseridos no repositório da ferramenta. Através da ligação dos componentes e a configuração da aplicação, a ferramenta possibilita gerar os arquivos fontes, executáveis e de descrição da aplicação. A validação da ferramenta é apresentada através de estudos de caso descritos no final da dissertação.

**Palavras-chave:** Sistemas Embutidos, Metodologias para Desenvolvimento de Sistemas Embutidos, Ferramentas de Configuração para Sistemas Embutidos.

## ABSTRACT

**Title:** “Development Environment for Component-Based Embedded Systems”

Embedded systems are systems addressed to a specific computational application, with high hardware components and software integration and a communication interface with the real world. The development of the project of a embedded system follows a methodology that involves the software and hardware components, taken into account the time and the cost of a project. In this sense, the specification of a tool capable to help those involved in the development of these systems, comes with the reduction of the time of development of a project and the possibility of software reusability components from applications already developed. This work presents a study about project methodologies for embedded systems, languages of description of components and existing project tools. Based on the concepts studied in this work the software components environment is proposed. The environment has an interface where each software component is written in a language denominated LDEC (language for description of components), that comes with the environment and is inserted in a repository of components. The present components in the repository can be selected, through a graphic interface, in way to compose a software of built-in application. The application or component can be reused, since they are described and inserted in the repository. Through the connection of the components and the configuration of the application, a code generator generates the files sources, executable and of description of the application. The validation of the tool is presented through case studies described in the end of this work.

**Keywords:** Embedded systems, Methodologies for Development of Embedded Systems, Configuration Tools for Embedded Systems, Development Environment and Code Generation.

## SUMÁRIO

|   |           |
|---|-----------|
| <b>SUMÁRIO .....</b>  | <b>1</b>  |
| <b>SISTEMAS EMBUTIDOS.....</b>  | <b>1</b>  |
| 1.1. Motivação .....  | 3         |
| 1.2. Justificativa.....   | 5         |
| 1.3. Objetivos e contribuições .....                                  | 7         |
| 1.4. Organização do Texto .....                                       | 8         |
| <b>PROJETO DE SISTEMAS EMBUTIDOS.....</b>                             | <b>10</b> |
| 2.2.1. Componentes de <i>Hardware</i> .....                           | 15        |
| 2.2.2. Componentes de <i>Software</i> .....                           | 16        |
| 2.3. Linguagens de Descrição de Componentes.....                      | 17        |
| 2.3.1. A Linguagem SDL (Specification and Description Language) ..... | 20        |
| 2.3.2. A Linguagem VHDL (Very High Speed Integrated Circuit) .....    | 23        |
| 2.3.3. SystemC .....  | 25        |
| 2.3.4. A Linguagem UML (Unified Modeling Language).....               | 26        |
| 2.4. Ferramentas de Configuração Existentes .....                     | 32        |
| 2.4.1. Koala.....   | 33        |

|   |           |
|---|-----------|
| 2.4.3. eCos - Embedded Configurable Operating System .....            | 37        |
| 2.5. Conclusão.....   | 42        |
| <b>A FERRAMENTA SIGAE.....</b>  | <b>43</b> |
| 3.1. Visão Geral .....  | 45        |
| 3.2. Arquitetura da Ferramenta .....                                  | 48        |
| 3.2.1. Módulo Descrição.....  | 48        |
| 3.2.2. Módulo Ligação .....   | 50        |
| 3.2.3. Módulo Gerador .....   | 51        |
| 3.2.4. Núcleo de Componentes do Sistema Operacional .....             | 52        |
| 3.2.5. Biblioteca de Componentes (Repositório).....                   | 56        |
| 3.3. Descrição dos Componentes na Ferramenta.....                     | 57        |
| 3.3.1. A Linguagem de Descrição de Componentes .....                  | 58        |
| 3.3.2. O Interpretador da Linguagem de Descrição de Componentes ..... | 63        |
| 3.4. Geração do Código.....   | 67        |
| 3.5. O projeto da ferramenta.....                                     | 69        |
| 3.6. Conclusão.....   | 77        |
| <b>ESTUDOS DE CASO.....</b>   | <b>79</b> |
| 4.1. Plataforma de hardware .....                                     | 80        |
| 4.1.1. Microcontrolador ATmega103.....                                | 81        |
| 4.1.2. Memórias.....  | 81        |
| 4.1.3. Teclado .....  | 82        |
| 4.1.4. Display de Cristal Líquido – LCD .....                         | 82        |
| 4.2. Estudo de Caso 1 : Relógio Digital .....                         | 82        |

|   |            |
|---|------------|
| 4.3. Estudo de Caso 2 : Forno de Microondas ..... | 100        |
| 4.4. Análise dos resultados .....                 | 105        |
| <b>CONCLUSÕES E TRABALHOS FUTUROS.....</b>        | <b>108</b> |
| <b>ANEXO 1.....</b>                               | <b>112</b> |
| <b>ANEXO 2.....</b>                               | <b>124</b> |
| <b>BIBLIOGRAFIA .....</b>                         | <b>130</b> |

# CAPÍTULO 1

## SISTEMAS EMBUTIDOS

A inserção da eletrônica na evolução dos equipamentos contribuiu para o crescimento do mercado dos dispositivos embutidos, o que trouxe progressos nesta área. Hoje, existem telefones que discam por comandos de voz, televisores que se conectam a Internet, microondas operados remotamente, máquinas fotográficas digitais e até automóveis que estacionam sozinhos. (Zelenovsky et al., 2003).

As mudanças continuam ocorrendo e, como consequência, necessita-se produzir ferramentas de *software* que satisfaçam as características exigidas pelos componentes destinados a cada tipo de sistema que se esteja desenvolvendo, tais como, performance, custo, consumo de bateria, dentre outros.

Desta forma, o projeto de sistemas embutidos torna-se bastante desafiador, envolvendo diversos conceitos e técnicas de desenvolvimento não tradicionais, dentre requisitos de *hardware* e *software* até restrições temporais de uma determinada aplicação.

Wolf (2001) define que Sistemas Embutidos são dispositivos programáveis concebidos para realizar uma função específica, são sistemas com requisitos específicos para cada projeto, tais como área, velocidade e consumo de potência necessitando desta forma, de uma configuração especial para cada aplicação.

Segundo StanKovic (2000), não existe uma forma eficiente de construir *software* para sistemas embutidos, principalmente no que se refere à análise dos componentes em função dos requisitos de uma determinada aplicação.

O projeto de um sistema embutido torna-se bastante complexo na medida em que envolve aspectos tais como, consumo de potência, alto desempenho, baixa disponibilidade de memória, portabilidade e confiabilidade (Wolf, 2001). Neste sentido, o projetista enfrenta muitos desafios, desde a arquitetura de hardware, que pode envolver um ou mais processadores, tipos diversos de estrutura de comunicação que pode variar de um barramento a uma rede complexa até as exigências de *software* para um sistema operacional de tempo real (RTOS), contendo módulos como de escalonamento de processos e serviços de comunicação, dentre outros. (Wagner, 2001).

Neste contexto, diversos trabalhos tem sido desenvolvidos com o intuito de facilitar a tarefa do projetista destes sistemas, onde cita-se: Vest<sup>1</sup> Uma Ferramenta para Construção e Análise de Sistemas Operacionais Baseados em Componentes para Sistemas Embutidos e de Tempo Real), koala<sup>2</sup> (Um Modelo de Componentes para *software* de equipamentos eletrônicos) e eCos<sup>3</sup> (Um sistema Operacional configurável para Sistemas Embutidos), sendo alvo de estudo neste trabalho.

---

<sup>1</sup> Disponível em <http://www.cs.virginia.edu/brochure/profs/stankovic.html>

<sup>2</sup> Abordado por (Ommering,R.V, Linden, F.V.D, Kramer, J., Mager, J., 2000)

<sup>3</sup> Disponível em <http://sources.redhat.com/ecos>



O projeto de sistemas embutidos compreende um conjunto de componentes de *hardware* e *software*. Alguns sistemas podem ser implementados com um microcontrolador, processador de sinais digitais ou um microprocessador e um *software* associado. Os componentes podem ser descritos com linguagens como SystemC, SpecC, Esterel, VHDL, UML, SDL dentre outras, dependendo do nível de abstração.

Este trabalho tem como objetivo desenvolver uma proposta de ferramenta para projeto de sistemas embutidos, com o intuito de facilitar as etapas de desenvolvimento destes projetos. No decorrer da elaboração do modelo da ferramenta, sentiu-se a necessidade de uma linguagem que possibilitasse tanto a descrição do componente na biblioteca, sua interface de comunicação com os demais componentes do sistema além de possibilitar a interpretação de cada informação do componente de modo a possibilitar a geração dos códigos da aplicação e a análise de que componente melhor atende os requisitos de uma determinada aplicação. Assim, a necessidade de uma linguagem com tais características, levou ao desenvolvimento de uma nova linguagem específica para o modelo de ferramenta proposto nesta dissertação.

### **1.1. Motivação**

O mercado de Sistemas Embutidos está crescendo a uma taxa surpreendente. Suas principais áreas de aplicação são telecomunicações, automobilísticos, eletrodomésticos e automação de escritórios.

Diversos fabricantes ao redor do mundo concorrem para produzir os dispositivos mais econômicos e com mais funcionalidades, tornando este mercado cada vez mais competitivo.

A construção de sistemas para estes dispositivos requer a utilização de componentes leves, análise de componentes que melhor atendam os requisitos do projeto (custo, performance, consumo de potência, etc.) como também a possibilidade de inserir restrições de tempo real.

Segundo (Friedrich, 2000), existem dois problemas envolvidos neste contexto, o desenvolvimento de novos componentes e a análise destes componentes. Segundo o autor, muitos trabalhos já se obtiveram quanto ao desenvolvimento de novos componentes. Enquanto este é um passo necessário, é o mais fácil e ignora completamente como os componentes interagem um com os outros. O segundo problema é que enquanto um trabalho significativo foi efetuado no desenvolvimento de componentes CORBA, DCOM, Jini, muito pouco foi feito com componentes satisfatórios para sistemas embutidos.

Para sistemas embutidos nós temos alguns trabalhos em componente para OSs. Estes incluem: MMLITE, Seixo, Puro, eCOS, icWorkshop e 2K. Porém, estes poderiam ser considerados a primeira geração sistemas e têm focado a construção de componentes interligados com uma infra-estrutura significativa e fixa. Estes sistemas freqüentemente têm pouco ou quase nada no que diz respeito à ferramenta de análise e, na maior parte, um suporte mínimo de configuração.

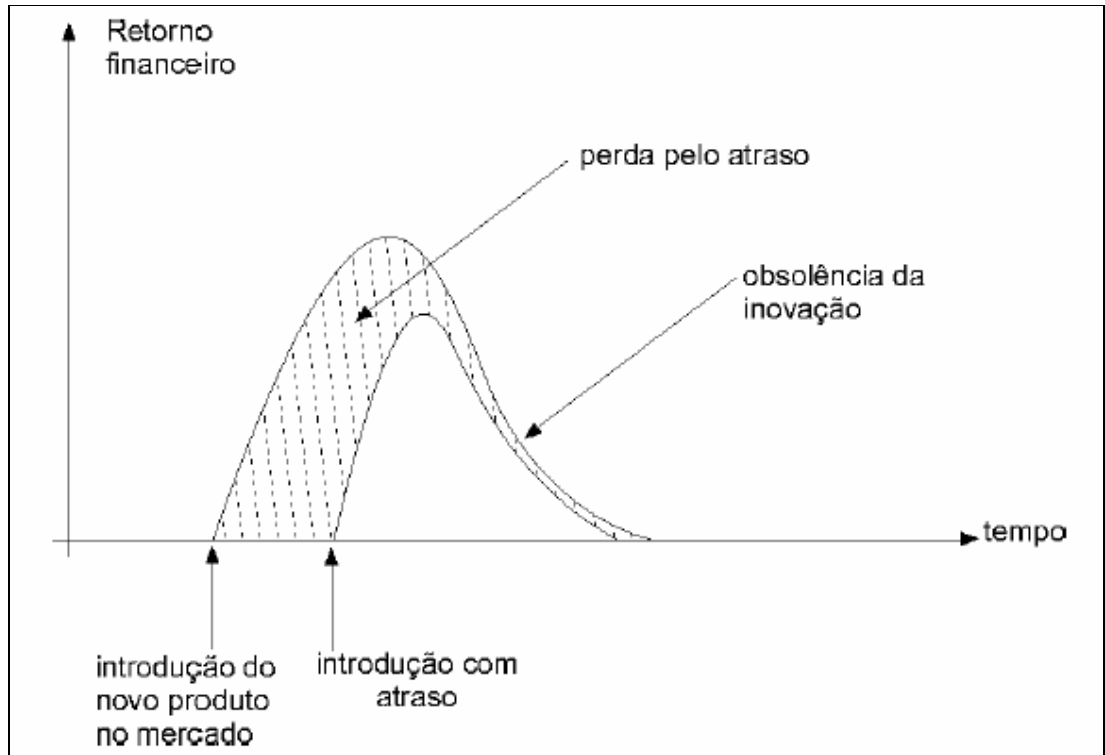
Faz-se necessária uma ferramenta que suporte a especificação de requerimentos de sistema embutidos seguidas por construção correta e útil do sistema embutido com

análise cuidadosa de dependências de componentes como também o tempo, memória, poder, e custo. Obviamente, temos muitas vantagens neste modelo de ferramenta. Por exemplo, minimiza o custo de novos códigos que devem ser escritos quando uma aplicação é desenvolvida e necessita de composição, que é essencial no desenvolvimento rápido de sistemas embutidos.

## **1.2. Justificativa**

Com a complexidade, tanto no nível de *hardware* quanto de *software*, no projeto de aplicações embutidas, percebe-se a necessidade de um ambiente que, possibilite o acesso, a configuração e a integração de componentes no âmbito de uma nova aplicação. Ou seja, um ambiente que permita explorar e avaliar a reutilização de componentes.

A reutilização de componentes associada a uma ferramenta que possibilite selecionar e analisar os componentes de maneira eficiente contribui para a redução significativa do tempo de projeto do sistema e conseqüentemente o tempo que um determinado produto leva desde sua concepção até chegar ao mercado consumidor. A redução do tempo de projeto “time-to-market”, está dentre os maiores desafios de um projeto. (Wagner, 2001).



**Figura 1 – Retorno Financeiro e Janelas de Tempo (Wagner, 2001)**

Segundo o autor, a pressão mercadológica num mercado mundial globalizado, somada à contínua evolução tecnológica, impõe às empresas a necessidade de projetarem novos sistemas embutidos dentro de janelas de tempo cada vez mais estreitas, de poucos meses. Conforme ilustrado na Figura 1, atrasos de poucas semanas no lançamento de um produto podem comprometer seriamente os ganhos esperados de um novo produto no mercado.

A proposta deste trabalho de dissertação é auxiliar o desenvolvimento de *software* para dispositivos embutidos que utilizam componentes genéricos em seu projeto. A proposta é exemplificada através de uma ferramenta de *software* para o desenvolvimento e geração de aplicações embutidas, isto é, uma ferramenta que

possibilita o suporte a uma biblioteca de componentes(repositório) e gera o código fonte em C para uma determinada plataforma alvo.

Escolheu-se a linguagem Visual C++ por fornecer facilidades no desenvolvimento de interfaces gráficas e a linguagem C pela facilidade na manipulação de dispositivos de *hardware* e, por ser amplamente utilizada no desenvolvimento de *softwares* para aplicações embutidas.

Através desta ferramenta proposta, um projetista pode facilmente desenvolver um *software* reutilizando os componentes existentes no repositório do ambiente ou inserindo novos componentes. Uma aplicação construída no ambiente da ferramenta pode ser inserida novamente no repositório, de modo a tornar-se um componente a ser incorporado em uma outra aplicação. A utilização da ferramenta tem as seguintes vantagens: redução do tempo de projeto de um *software* embutido, reutilização de *software* e conseqüentemente, uma redução do custo do desenvolvimento de uma aplicação embutida e seu respectivo tempo de incorporação no mercado. (“*time-to-market*”)

### ***1.3 Objetivos e contribuições***

Esta dissertação tem como principal objetivo a proposta de um ambiente com suporte computacional para o desenvolvimento e geração de *software* para aplicações embutidas usando o conceito de componentes. O ambiente proposto permite o armazenamento de componentes de *software* através de uma estrutura denominada *repositório*, permitindo a ligação dos componentes definidos para um determinado projeto e a geração da aplicação propriamente dita.

Para atingir tal objetivo, o trabalho de pesquisa descrito nesta dissertação, resultou nas seguintes contribuições:

- criação de uma linguagem de descrição de componentes de *software* específica para o ambiente;
- desenvolvimento de um interpretador voltado à utilização da linguagem na manipulação do repositório de componentes;
- criação de uma estrutura de armazenamento de componentes reutilizáveis;
- suporte à descrição da arquitetura de *Software* de modo a documentar tanto a aplicação quanto os componentes que fazem parte do projeto de uma aplicação;
- desenvolvimento de um gerador de código para as aplicações desenvolvidas no ambiente.

#### ***1.4. Organização do Texto***

Este trabalho conta com mais quatro capítulos, além deste capítulo introdutório. O Capítulo 2 apresenta os diversos aspectos e requisitos que envolvem um Projeto de Sistemas Embutidos, as características das principais linguagens que podem ser utilizadas para descrição de sistemas baseados em componentes e a descrição de algumas Ferramentas de Configuração e Desenvolvimento de Aplicações Embutidas. O Capítulo 3 apresenta a arquitetura da ferramenta e detalhes de descrição de cada módulo. Outro tópico também detalhado neste capítulo é referente às características da linguagem de descrição de componentes e a aspectos de implementação do projeto do

ambiente proposto nesta dissertação. Os estudos de caso feitos no ambiente desenvolvido são relatados e analisados no Capítulo 4. Finalmente, no Capítulo 5, são apresentadas as conclusões sobre o trabalho desenvolvido e sugestões de melhoramentos para trabalhos futuros.

## CAPÍTULO 2

### PROJETO DE SISTEMAS EMBUTIDOS

A essência de um projeto para sistemas embutidos está em implementar um conjunto específico de funções que satisfaçam as características de desempenho, custo, consumo de potência e tempo. A escolha de uma arquitetura é que determina que componentes de *hardware* e *software* poderão ser utilizados. (Sangiovanni-Vincentelli at al., 2001).

O autor traz uma abordagem de projeto baseado em plataforma (*platform-based design*), onde uma plataforma é uma abstração que possibilita refinamento em camadas de baixo nível.

Em geral, as plataformas são caracterizadas por componentes programáveis. Uma biblioteca de uma determinada plataforma pode conter componentes reconfiguráveis em tempo de projeto. A combinação de programação, processadores configuráveis e configuração em tempo de execução resulta em uma plataforma “*highly programmable*”<sup>4</sup>. (Sangiovanni-Vincentelli at al., 2001).

---

<sup>4</sup> plataforma altamente configurável



Ainda segundo o autor, para maximizar o reuso de *software*, a plataforma da arquitetura deve estar abstraída até o nível que uma aplicação possa ter um alto nível de interface com o *hardware* (API). Uma camada de *software* executa nesta abstração, conforme ilustrado na Figura 2. Esta camada concentra as partes essenciais da plataforma de uma arquitetura:

- os blocos programáveis e o subsistema de memória por um RTOS (*Real Time Operation System*);
- o subsistema de I/O pelos *drivers* de dispositivo e
- a conexão de rede pelo subsistema de comunicação de rede.

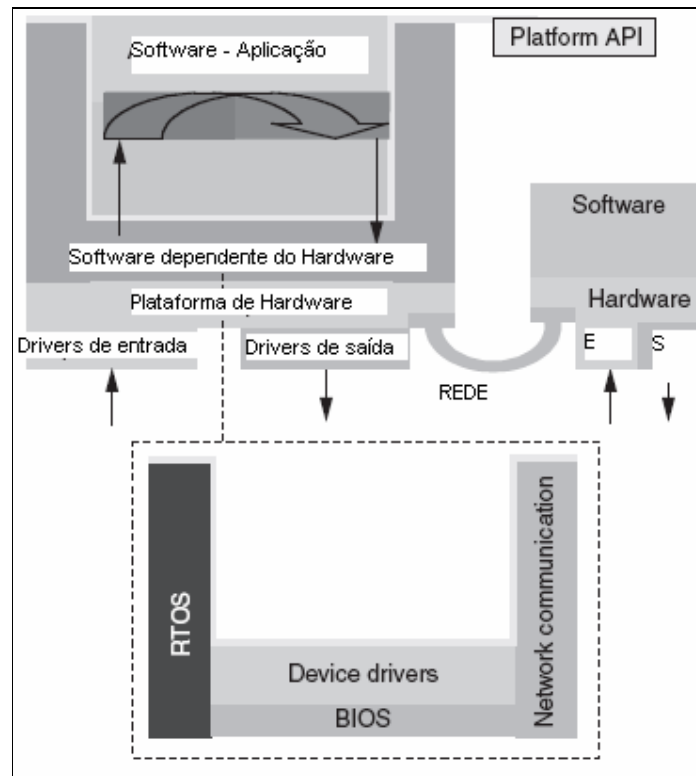


Figura 2 – Estrutura de software em camadas (Sangiovanni-Vincentelli et al., 2001)

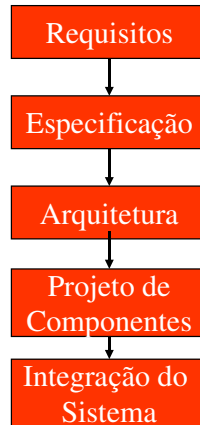
Desta forma, uma plataforma API provê uma interface entre o *software* dependente do *hardware* e o *middleware* e a camada de aplicação do *software*. A plataforma é geralmente parte de uma rede e disponibiliza uma interface para o ambiente de rede.

Por outro lado, em um projeto de sistema embutido, como em qualquer outro tipo de projeto de software, é importante que haja uma metodologia de projeto a ser seguida. Wolf (2001) explana três razões que fortificam a utilização de uma metodologia:

- facilita o acompanhamento das etapas de projeto;
- facilita uma manutenção das necessidades da aplicação;
- permite desenvolver ferramentas de projeto computacional e;
- facilita a comunicação entre os membros de um projeto.

A Figura 3 mostra um resumo dos principais níveis de abstração em um Projeto de Sistema Embutido. Em uma visão *top-down*, onde inicia-se pelos Requisitos do Sistema, seguindo temos o passo Especificação, onde criamos uma descrição detalhada das necessidades da aplicação. Os detalhes internos do Sistema são analisados quando desenvolvemos a Arquitetura, que fornece a estrutura dos Componentes do Sistema. Uma vez conhecendo os componentes necessários, podemos projetar estes componentes, incluindo os módulos de software e as especificações de hardware necessárias. Baseado nestes componentes, finalmente implementa-se um sistema completo.

## Níveis de Abstração



**FIGURA 3 - Principais Níveis de Abstração no Processo de Projeto (Wolf, 2000)**

O projeto da Arquitetura irá definir quais os componentes que satisfazem a Especificação do Sistema. A Arquitetura inclui os componentes de *hardware* e os componentes de *Software*.

Uma outra visão para metodologia de projetos de sistemas embutidos é apresentada em (Silva et al,1999). Os autores identificam quatro atividades essenciais durante o projeto de um sistema integrado de *hardware* e *software* seguindo especificação, particionamento, síntese e validação. A especificação e análise onde tem-se a utilização de linguagens visuais ou textuais que possibilitem identificar e descrever quais componentes de *software* e *hardware*<sup>5</sup> devem ser implementados; o particionamento, onde são determinados os dois tipos de componentes, *hardware* e *software*. Quando houver um fluxo de informações entre estes componentes, haverá

---

<sup>5</sup> *firmware* ou *device drives*

uma interface entre o *hardware* e o *software*; a síntese tendo-se implementadas as primitivas básicas dos componentes de *hardware* e *software* (já particionados) – portas lógicas para o *hardware* e linguagem de máquina para o *software* e uma interface é gerada para as diferentes interconexões entre componentes de *hardware* e *software*; e finalmente a validação onde é feito a verificação, através de simulação ou emulação, do modelo.

### 2.2.1. Componentes de *Hardware*

Uma Arquitetura de Hardware de um Sistema Embutido inclui os seguintes elementos:

CPU - um Sistema Embutido claramente deve conter um microprocessador. Existem várias arquiteturas de microprocessadores, e cada qual contendo modelos variando a velocidade de *clock*, largura do barramento de dados e assim por diante. A escolha da CPU é muito importante, mas não pode ser analisada separadamente sem considerar o software que será executado na máquina.

Barramento - a escolha do barramento está intimamente ligada à escolha da CPU, sendo o barramento uma parte integrante do microprocessador. Mas em aplicações que utilizam intensamente o barramento, esta escolha pode ser um fator mais limitante que a CPU.

Memória - determinar quais as características da memória depende do volume de dados e o tamanho das instruções do programa. A velocidade da memória irá determinar grande parte da desempenho do sistema.

Dispositivos de Entrada e Saída (E/S) - a escolha dos dispositivos de E/S pode variar desde os mais sofisticados aos mais simples e de baixo custo, dependendo dos requisitos de interface com o usuário.

### 2.2.2. Componentes de *Software*

Segundo Wolf (2001), um aspecto fundamental no projeto da Arquitetura do *Software* é o **particionamento** – quebrar a funcionalidade do sistema em partes de forma que seja mais facilmente implementado, testado e modificado.

Dividindo a funcionalidade do sistema em partes que correspondem a um modelo principal de operações e funções é frequentemente a melhor escolha. Isto porque, diferentes tipos de funcionalidades necessitam de códigos diferentes organizados em procedimentos separados. Esta característica possibilita que um componente de *software* possa ser desenvolvido independentemente e posteriormente formar uma composição direcionada as características de uma aplicação específica.

Szyperski *apud* Urting et al. (2004), define que “Um componente de *software* é uma unidade de composição com interfaces específicas através de contratos e dependências de contexto explícitas. Um componente pode ser distribuído independentemente e está sujeito a composição com outras partes”.

O desenvolvimento de uma aplicação baseada em componentes requer que estejam explícitas as especificações de cada componente e suas interfaces de forma que possibilitem a composição dos mesmos. Desta forma, um componente é uma unidade de desenvolvimento independente, tal qual uma “caixa preta”, que possibilite sua utilização e reutilização através de interfaces de acesso à sua estrutura interna. As interfaces podem fornecer serviços diferenciados atendendo a diferentes projetos de *software*, sendo que através das interfaces está a possibilidade de um componente interagir com outros componentes.

Segundo Villela (2001), a facilidade de reutilização de componentes de *software* pode ser realizada através da implementação de um repositório de componentes. Através da estrutura de um repositório, é possível construir aplicações ligando componentes existente no repositório, através das interfaces definidas em cada componente. Uma aplicação desenvolvida poderá tornar-se um novo componente do repositório de modo a compor um outro projeto de *software*.

Um componente de *software* é uma parte encapsulada do *software* com uma *interface* explícita em seu ambiente, projetado de forma que possa ser utilizado em diferentes configurações.

No contexto dos sistemas embutidos, conforme relatos deste capítulo, pode-se implementar um ambiente baseado em componentes. Os componentes de sistemas embutidos devem considerar requisitos de desempenho, *deadline*, segurança, custo, ligação com um *hardware* específico e uma análise global do sistema. Os componentes devem ter um domínio específico direcionado para uma necessidade particular e obedecendo as especificações exigidas para tempo real, segurança, potência, tamanho e custo. Devem podem criados hierarquicamente, gerando subcomponentes. (Wolf, 2001).

### ***2.3 Linguagens de Descrição de Componentes***

Segundo Wolf (2001), um *Sistema computacional* pode ser definido como um sistema digital composto por componentes de *hardware* e *software*. Para poder elevar o nível de abstração destes sistemas, utilizam-se ferramentas de apoio ao projeto que aceitam especificações em linguagens como SystemC, SpecC, Esterel, VHDL, SDL,

UML, Java, entre outras (Marcon, 2002). Não existe um consenso quanto qual linguagem seja a melhor para a especificação de sistemas computacionais em geral. Pode-se supor que as diferentes linguagens sejam adequadas aos domínios de aplicações originais para os quais foram concebidas.

Um *modelo* é uma abstração de um sistema real e utilizado para viabilizar o raciocínio sobre este sistema. Neste contexto, *abstração* é a seleção de características e propriedades essenciais e a correspondente exclusão de outras características que não são relevantes em um dado contexto. Um *modelo computacional* (em inglês, *model of computation* ou MOC), segundo Marcon (2002), é um modelo de um sistema composto por *hardware* e *software*. Estes modelos servem para formalizar as características de uma classe de sistemas, habilitando a manipulação elementos computacionais desta classe. Uma descrição de um sistema computacional é uma representação gráfica do seu modelo. Uma distinção importante entre os conceitos de modelo e descrição. É que enquanto a descrição designa representações concretas, um modelo designa uma representação abstrata.

Ainda segundo Marcon (2002), as propriedades mostram a capacidade de representação de um modelo computacional são:

- *Suficiência* – é a capacidade de um modelo computacional de ser autônomo para representar todo o comportamento de um sistema computacional modelado através dele. Assim os modelos computacionais em geral são capazes de representar qualquer algoritmo computável, por terem poder computacional equivalente a máquinas de Turing;



- *Expressividade* – é a capacidade de um modelo computacional para representar as estruturas básicas essenciais de um sistema computacional.

A modelagem de um sistema deve ser realizada através o emprego de primitivas suportadas pelo modelo computacional escolhido. Caso essas primitivas não sejam capazes de capturar as estruturas básicas do sistema computacional, o modelo computacional é inadequado, isoladamente, para a modelagem que se deseja fazer. A capacidade de representação de um modelo computacional é afetada pelo seu grau de especialização que, quanto maior, mais reduzida é a quantidade de sistemas que podem ser expressos por este. Em contrapartida, quanto menos abstrato for este, mais expressivo ele será, aumentando a sua capacidade de dar melhor suporte à representação de diferentes sistemas. Em um extremo poder-se-ia assumir que cada sistema computacional teria o seu próprio modelo computacional, em cujo caso os conceitos de modelo computacional e modelo do sistema computacional confundem-se. Mas isto não é prático, pois existe uma infinidade de sistemas com características comuns que podem ser capturadas por modelo computacional mais genérico, e tratadas uniformemente. Logo, a principal vantagem da generalidade é habilitar a automatização do processo de projeto com base nos modelos computacionais, produzindo métodos e ferramentas automatizadas para aplicar no projeto de uma grande classe de sistemas computacionais. Apenas para exemplificar, considere dois circuitos de transmissão de dados, cuja diferença é uma palavra de alinhamento (seqüência ordenada de bits que permite sincronizar a operação do sistema). Ambos circuitos têm pelo menos uma diferença, o subcircuito que calcula o alinhamento. Todavia, o modelo computacional subjacente não necessita em geral, ser alterado devido a este subcircuito. Em contrapartida, a arquitetura e o comportamento de um sistema não definem o modelo computacional.

Como exemplo, considere um circuito que implemente uma determinada funcionalidade, e que este mesmo circuito deva ter sua frequência de operação aumentada em 10 vezes. Neste caso, nem a arquitetura, nem o comportamento são alterados, mas devido às novas restrições de projeto, um circuito que poderia ser implementado por um modelo computacional essencialmente seqüencial, poderá ter que ser implementado por um circuito com um modelo computacional que demonstre melhor o paralelismo e com isto atenda os requisitos de desempenho. O que se pode deduzir é que o modelo computacional serve para refletir a organização necessária para implementar o comportamento do sistema computacional frente ao seu conjunto de requisitos.

Marcon (2002) afirma que a implementação de um projeto de sistemas embutidos envolve um grande número de processos. Quanto maior o sistema, maior o número de processos concorrentes e maior a dificuldade de especificar estes sistemas em linguagem formal.

Conforme já destacado no início deste capítulo, muitas linguagens existentes têm seu foco direcionado para a construção e especificação de componentes de *software*. No próximo item deste capítulo estão em destaque algumas destas linguagens, trazendo uma abordagem de suas principais características, vantagens e desvantagens no contexto de projeto de *software* para sistemas embutidos.

### **2.3.1. A Linguagem SDL (Specification and Description Language)**

(Duarte, 2003) apresenta no texto abaixo uma breve definição e descrição da Linguagem SDL:

A *Specification and Description Language (SDL)* é, como diz o próprio nome, uma linguagem de especificação e descrição de sistemas. Esta linguagem é padronizada pela *International Telecom Union (ITU)* e foi concebida para a especificação e descrição de sistemas de telecomunicações, sistemas distribuídos e sistemas embarcados. Por isso, SDL foi, originalmente, uma linguagem utilizada por empresas de telecomunicações, mas já é hoje utilizada em vários setores industriais para descrição de sistemas de tempo real. SDL apresenta uma notação gráfica denominada SDL/GR e uma sintaxe textual padrão equivalente chamada de SDL/PR. SDL apresenta vários tipos de construções para representar a descrição de um projeto, sendo que cada tipo de construção correspondente a uma *visão*. As diferentes visões são a visão arquitetural (*architecture view*), a visão da comunicação (*communication view*), a visão comportamental (*behavior view*) e a visão da informação (*information view*). Estas diferentes visões formam um conjunto coerente e consistente que permite a descrição completa de um projeto. (Duarte, 2003).

Originalmente a linguagem SDL foi desenvolvida para o projeto de sistemas de telecomunicações, mais especificamente dirigida à protocolos de comunicação e sistemas distribuídos. No entanto, a linguagem SDL também é bastante utilizada para especificação e prototipação de sistemas embutidos de tempo real. Padrão desenvolvido pelo ITU-TS. A SDL baseia-se em métodos formais, o que torna viável seu uso como ferramenta de especificação, verificação, testes e simulação.

A linguagem SDL possui três componentes principais:

- **Sistema:** é o nível mais externo, o seu objetivo é delimitar o ambiente da especificação. O sistema serve como um recipiente para o armazenamento de um conjunto de instâncias de blocos;
- **Blocos:** formam o nível intermediário da especificação, e têm por objetivo estruturar o sistema de forma clara, fornecendo um mecanismo de abstração gradativa da complexidade imposta pela especificação. Cada

instância de um bloco é um recipiente para um conjunto de instâncias de processos ou para um conjunto de instâncias de outros blocos;

- **Processos:** realizam a camada mais próxima da implementação de um sistema, onde os sinais trocados dentro do sistema são efetivamente tratados. A instância de um processo ou é a implementação de uma máquina de estados, ou é um conjunto de instâncias de serviços.

Na especificação SDL, a estrutura hierárquica é composta da seguinte forma: na raiz o objeto sistema . Os nós intermediários possuem objetos do tipo bloco , nomeados por B1, B2, etc., que são dispostos hierarquicamente. E nas folhas encontram-se os processos (P1, P2, etc.), constituídos por máquinas de estados finitos – conforme Figura 4.

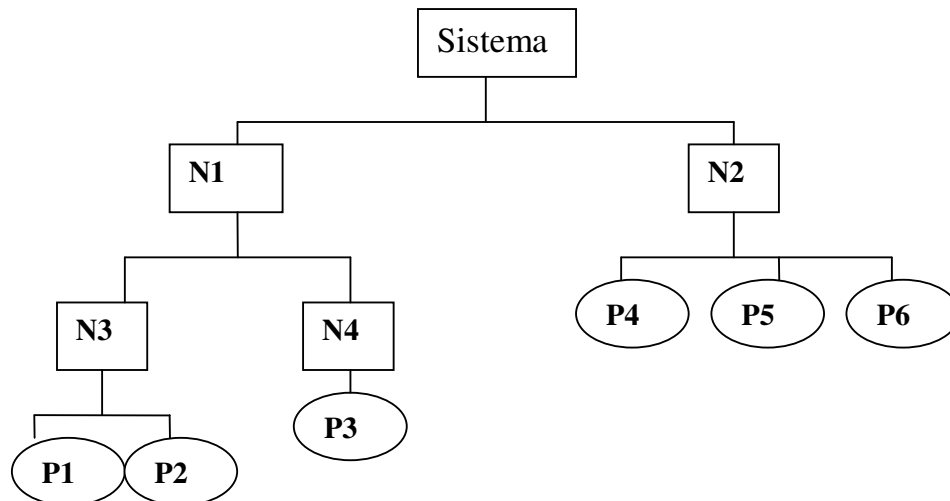


FIGURA 4 – Estrutura Hierárquica SDL

Existem diversos adjetivos que caracterizam os tipos de sistema que devem ser especificados em SDL. Estes adjetivos expressam propriedades típicas de sistemas de tempo real:

- cooperativos, coordenados, comunicantes, reativos e interativos;
- concorrentes, paralelos;
- distribuídos e não determinísticos.

Todos estes adjetivos são expressos por SDL através de um modelo que descreve o comportamento dinâmico do sistema e o comportamento da estrutura interna do mesmo tais como seus componentes e a comunicação entre eles.

A linguagem SDL apresenta dois padrões de representação: o gráfico SDL-GR e o textual SDL-PR, sendo que a representação textual é usualmente gerada a partir da representação gráfica e é utilizada para fins como geração de código, verificação, simulação, testes e otimizações. (Duarte, 2003).

### **2.3.2. A Linguagem VHDL (Very High Speed Integrated Circuit)**

A linguagem VHDL é uma linguagem padronizada pela IEEE, utilizada para descrever *hardware* em vários níveis de abstração (Perry, 1998). VHDL foi originalmente utilizada para simulação e documentação, sendo posteriormente adotada para a síntese automatizada de sistemas digitais. O fato de ser um padrão aberto permitiu o intercâmbio de descrições entre várias ferramentas e sistemas de apoio ao projeto, fazendo com que se seja hoje muito usada em projetos de *hardware*.

VHDL é uma linguagem imperativa com algumas características de orientação a objetos, de origem essencialmente paralela, mas que permite a programação sequencial dentro de processos. Diferentes processos são usados para implementar comportamentos concorrentes. VHDL também permite representar vários níveis de abstração, o que inclui desde o nível de chaves até o nível de sistemas (Perry, 1998).

A forma mais abstrata de descrever sistemas utilizando VHDL é através de construções comportamentais. Embora estas construções sejam de grande valia para especificações iniciais e simulação destas, as ferramentas de síntese dificilmente conseguem obter bons resultados a partir de descrições com construções comportamentais, o que força os projetistas de VHDL terem em mente o *hardware* alvo, sempre que forem projetar um circuito e produzir eventualmente mais de uma descrição do projeto em mais de um nível de abstração. Embora VHDL tenha um alto potencial para a descrição de *hardware*, a linguagem não foi direcionada para a descrição de *software*, o que é fácil de notar pela incapacidade da linguagem para descrever operações dinâmicas, tais como alocação e liberação de memória, criação e destruição de processos.

A linguagem VHDL foi projetada para permitir uma boa descrição hierárquica e modular dos circuitos. Para tanto, a construção principal é dada pelo par “entidade-arquitetura”.

Entidade é uma construção que define os sinais da interface de um bloco construído em VHDL com os demais blocos. Arquitetura é uma construção que descreve o comportamento do sistema, cuja interface está definida na entidade correspondente. Uma arquitetura é uma implementação de uma interface.

### 2.3.3. SystemC

É uma linguagem que utiliza a Biblioteca de classes para C++, com suporte ao co-design do *hardware-software* e a descrição da arquitetura de sistemas complexos que consistem de componentes de *software* e *hardware*. A linguagem SystemC possibilita criar a especificação de um sistema através de definições específicas. (Salmito, 2003).

A linguagem SystemC apresenta as seguintes definições:

- **Módulos:** São entidades hierárquicas onde podem conter outros módulos ou processos.
- **Processos:** São usados para descrever as funcionalidades dos módulos.
- **Portas:** Módulos possuem portas nas quais o conecta a outros módulos. Podem ser unidirecionais ou bidirecionais.
- **Sinais:** São usados para se conectar portas (fios). Podem ser controlados por um (*unresolved*) ou vários (*resolved*) *drivers*.
- **Clocks:** É um sinal especial usado para a sincronização do sistema durante a simulação.

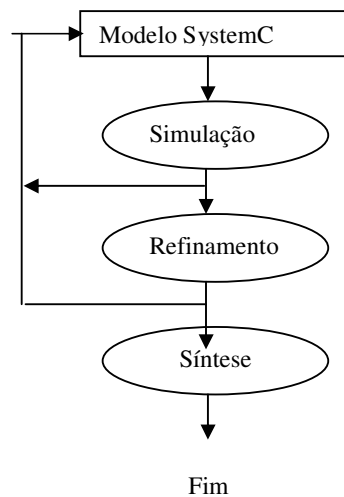


FIGURA 5 - Design Utilizando SystemC

São características da linguagem SystemC:

- C++ provê a abstração de controle e de dados necessários para descrição de sistemas.
- A orientação a objeto e as macros do C++ permite estender a linguagem com classes, sem mudar a sua sintaxe.
- Verificar a funcionalidade do sistema antes do início da sua implementação.
- Criação de modelos de performance do sistema, ou validação da performance do sistema.

#### **2.3.4. A Linguagem UML (Unified Modeling Language)**

O objetivo da UML é desenvolver uma linguagem padronizada para a especificação, visualização, documentação e construção de componentes de um sistema. (Booch, 1994).

A UML não é uma metodologia já que as metodologias consistem de linguagem de modelagem e procedimento de uso do mesmo.

A UML pode ser utilizada para:

- Mostrar as fronteiras de um sistema e suas principais funções usando atores e estudos de caso;
- Ilustrar a realização de estudos de casos com diagramas de interação;
- Representar uma estrutura estática de um sistema utilizando diagramas de classes;



- Modelar o comportamento de objetos com diagramas de transição de estados;
- Revelar a arquitetura de implementação física com diagramas de comportamento e implantação;
- Estender sua funcionalidade através de estereótipos.

A UML não é só uma simples padronização em busca de uma notação unificada, já que contém conceitos novos que são encontrados em outros métodos orientados a objetos. A UML, em seu estágio atual, define uma notação e um metamodelo. A notação é o material gráfico dos modelos, o metamodelo é a sintaxe da linguagem de modelagem.

Segundo Souza (2000), a linguagem UML é considerada atualmente um passo significativo em termos de notação para descrição de sistemas OO, pois reúne relevantes experiências neste campo. Outra vantagem da utilização desta linguagem diagramática é o fato de que suas notações são aplicáveis a quaisquer processos de desenvolvimento. Além disto, propõe mecanismos de extensão, tais como: *stereotype*, *tagged value* e comentários. Isto torna a notação da UML flexível e adaptável para a descrição de sistemas, em diferentes áreas de aplicação.

Lavazza em (Lavazza, 2003), fala como utilizar a UML para modelar sistemas de tempo real. Na UML para tempo real, UML-RT, são propostos mecanismos para modelar as particularidades dos sistemas de tempo real. Basicamente, é empregado o mecanismo de extensão, *stereotype*, para instanciar novas subclasses de conceitos que irão, por exemplo, modelar os objetos ativos, as mensagens e os eventos. Esta extensão

propõe uma notação gráfica para expressar marcas de tempo<sup>6</sup>, marcas de estado<sup>7</sup>, mensagem *broadcast* e mensagem concorrente no diagrama de seqüência. Propõe, também, que mensagens estereotipadas sejam empregadas no diagrama de colaboração. Sugere a utilização do diagrama de temporização, presente na metodologia de (Booch, 1994), para auxiliar nas descrições dos requisitos temporais. Outra sugestão relevante está no uso do diagrama de contexto da análise estruturada. O objetivo do uso do diagrama de contexto é mostrar os objetos do sistema, interagindo com os objetos externos do ambiente, de forma que possa ser visualizado a troca de mensagens e os eventos que ocorrem entre o 'sistema' e o 'meio'.

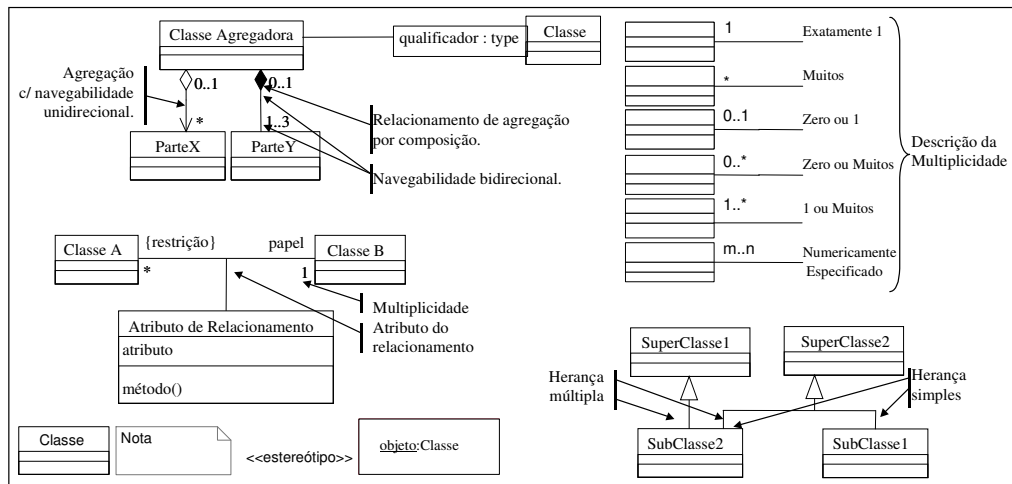


FIGURA 6 – Notação do Diagrama de Classes (Souza, 2000)

No diagrama de classe (Figura 6) é modelada a estrutura estática das entidades que compõem o sistema. Sendo que o diagrama de objetos mostra as instâncias das

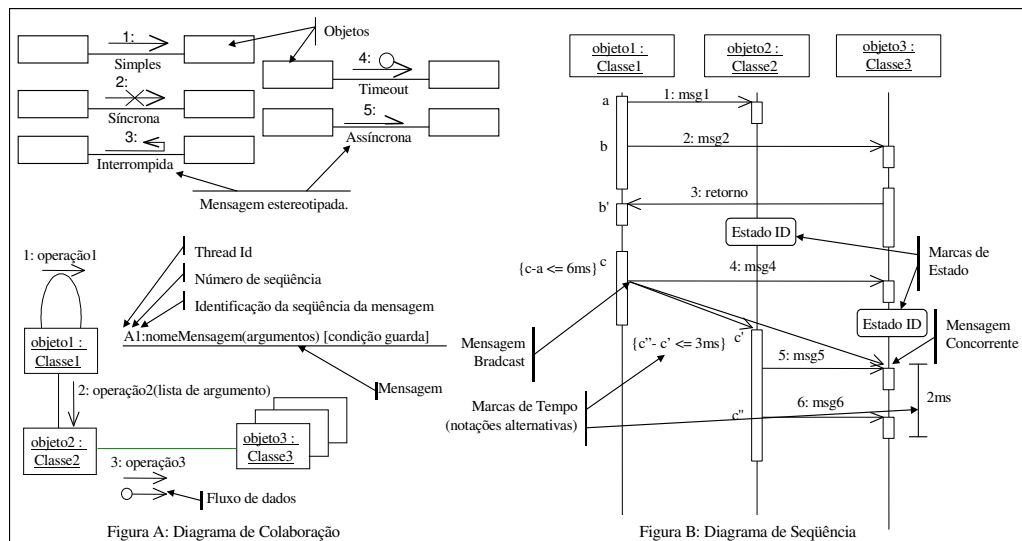
<sup>6</sup> Marcas de tempo são os adornos ou restrições expostas junto às mensagens, indicando um tempo máximo para a execução de um conjunto delas. Podendo, também, indicar um tempo máximo para que a funcionalidade, descrita através do diagrama de seqüência, seja executada.

<sup>7</sup> As marcas de estados, no diagrama de seqüência, indicam em que 'estado' o objeto se encontra no instante em que o método, disparado pela mensagem, está executando.

classes - útil nas descrições da simulação do modelo e na modelagem de características e comportamentos inerentes ao objeto.

O diagrama de estados descreve os estados possíveis que um objeto poderá assumir, num dado instante de tempo. Descreve, também, os eventos, as condições ou as ações que disparam as transições de um estado, do objeto, para outro. Este diagrama apresenta as ações que são executadas enquanto o objeto permanece no estado.

O diagrama de seqüência (Figura 7) pode ser visualizado como o sistema se comporta ao realizar uma atividade. O diagrama de colaboração e o diagrama de seqüência mostram informações semelhantes, porém com representações gráficas distintas. Desta maneira, o uso do diagrama de colaboração é importante quando há a necessidade de evidenciar os objetos e seus vínculos, facilitando a compreensão da interação, como é mostrado na Figura 7.A.



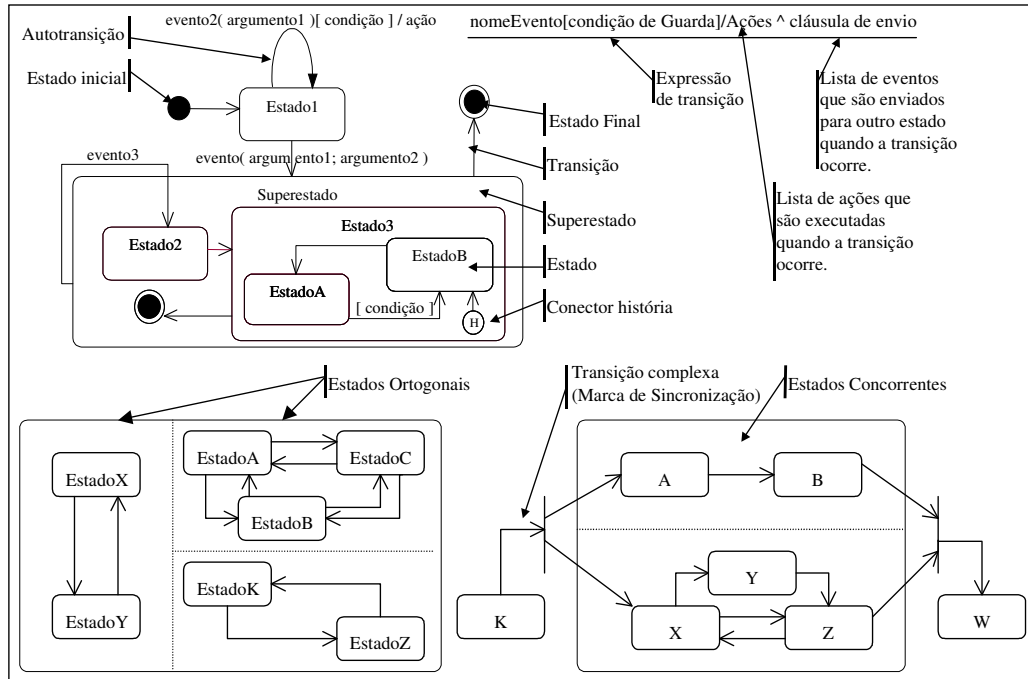
**Figura 7 – Notação dos Diagramas de Interação (Souza, 2000)**

Como mencionado anteriormente, foi proposto um padrão de estereótipos para representar as mensagens periódicas e aperiódicas e um padrão de sincronização das

mesmas, para a notação dos diagramas de interação da UML-RT. Mensagens periódicas são aquelas caracterizadas por um período, dentro do qual devem ocorrer, e pelo *jitter*, que é a variação em torno do período, o ocorrido *versus* o tempo em que deveriam ocorrer. Um mensagem é dita aperiódica quando o intervalo de chegada ao objeto não pode ser prognosticado. Estas mensagens possuem um intervalo mínimo de chegada. Caso este tempo seja ultrapassado, expressa que há possibilidade de chegada em grupo. São independentes e, de certa forma, randômicas. Isto significa que a chegada de uma mensagem não afeta a probabilidade de chegada da próxima, dentro do intervalo de tempo mínimo estabelecido.

O diagrama de estados é uma técnica bem conhecida dentro da orientação a objetos. É baseado nos *statechart* de David Harel (Harry, 1998). Foi adotada por Rumbaugh no método OMT e por Booch (Booch, 1994). Esta técnica também incorpora características das máquinas de *Moore* e máquinas de *Mealy*. A aplicabilidade deste diagrama é na descrição do comportamento dos objetos. Cada diagrama descreve os estados possíveis que um objeto pode assumir, num dado instante, dentro da delimitação do problema modelado. A notação para descrever os diagramas de estados é mostrada na Figura 8.

As visões estáticas e dinâmicas dos sistemas modelados com a notação (dos diagramas de classes, de objetos, de interação e de estados) da UML-RT utilizam conceitos que são descritos por mais de um diagrama. Isto dificulta a tarefa da equipe de projeto em manter a consistência destes conceitos sem o auxílio de mecanismos que propiciem, de forma automática, esta facilidade. Além disso, na modelagem de um sistema de tempo real, é imprescindível que cada diagrama apresente a informação descrita de forma consistente e complementar com os demais.



**FIGURA 8 – Notação dos Diagramas de Estados (Souza, 2000)**

## **2.4. Ferramentas de Configuração Existentes**

Com o intuito de auxiliar no desenvolvimento de aplicações embutidas e na reutilização de componentes de *software*, diversas ferramentas estão disponíveis tanto no mercado quanto em projetos em academias. Projetos acadêmicos, como por exemplos OS-Kit e Vest têm em comum a intenção de distribuir a construção do Sistema Operacional através de composição.

Entretanto, muitas destas ferramentas não dispõem de compatibilidade na configuração ou não atendem às necessidades específicas das aplicações. Desta forma, é necessário melhorar as ferramentas de configuração e análise do Sistema Operacional, bem como a construção de seus componentes. (Friedrich, 2000).

O OS-Kit (Ford, 1997) dispõe de um conjunto de componentes de SO que pode ser combinado para configurar um SO. Entretanto, não especifica qualquer regra de ajuda na construção do Sistema.

O Modelo de Componentes Koala (Ommering et al, 2000) disponibiliza uma excelente gama de recursos em sua ferramenta, no entanto, apresenta o foco específico ao desenvolvimento de famílias de aparelhos televisores.

Nos próximos tópicos serão abordadas, com maiores detalhes, as características e recursos disponíveis em algumas ferramentas estudadas neste trabalho de pesquisa.

### 2.4.1. Koala

O Modelo de Componentes Koala baseia-se no conceito de reutilização de um mesmo software para diversas aplicações. O componente Koala é uma unidade de projeto, desenvolvimento e reutilização. Para (Ommering et al, 2000), um componente embora pequeno, requer muito tempo e esforço de desenvolvimento.

Neste modelo, um componente disponibiliza suas funcionalidades através de interfaces, o acesso a funcionalidades externas também é feito da mesma forma.

A Figura 9 representa graficamente uma plataforma de software de televisão no modelo Koala. As interfaces são representadas por pinos de *chips* e os triângulos indicam a direção da chamada de uma função entre os componentes.

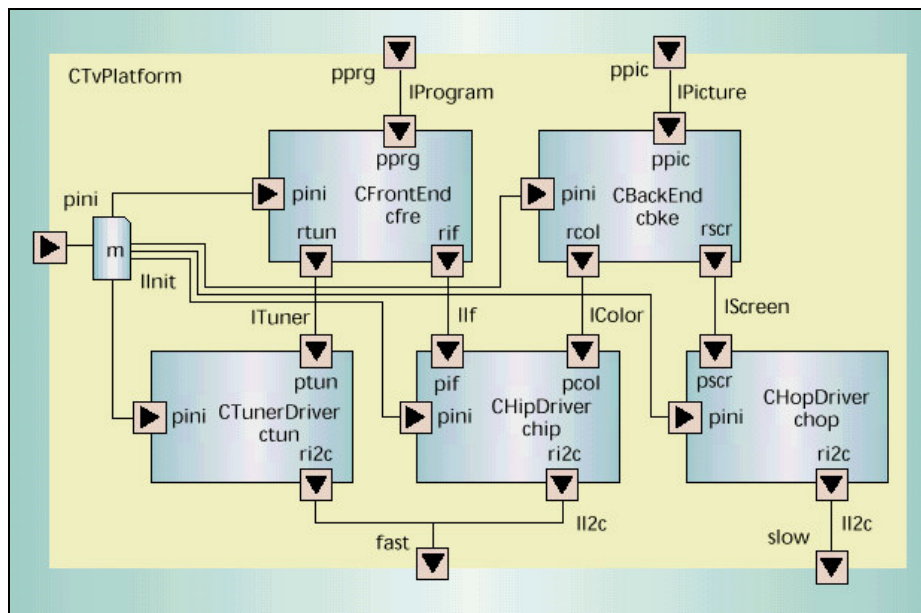


FIGURA 9 – Plataforma de Tv no Modelo Koala(Ommering et al, 2000)

A interface é definida usando uma linguagem simples de definição de interface (IDL), onde seus protótipos de funções são listas, conforme o exemplo de definição da interface Ituner :

```
Interface ITuner
{
    void SetFrequency (int f);
    int GetFrequency (void);
}
```

Segundo (Ommering et al, 2000) a descrição de um componente é feita através da linguagem CDL (*Component Description Language*), exemplificada na descrição do componente *CTunerDriver*.

O armazenamento dos componentes é feito através de um *repositório de componentes*, onde os desenvolvedores podem adicionar ou utilizar os componentes disponíveis. Este repositório fica submetido a um sistema gerenciador de configurações, onde ficam armazenados históricos de utilização dos componentes. A interface do repositório consiste em uma descrição IDL, informações sobre o componente e uma descrição da semântica de cada interface. O repositório é baseado na *web* e acessado mundialmente.

O motor afinador da plataforma de Tv ilustrado na Figura 9 fica descrita em CDL da seguinte forma:

```
componente CTunerDriver
{
    provides Tuner ptun;;
```



```

        IInit pini;
    requires I2c ri2c;
}

```

Cada interface está rotulada com dois nomes. Um nome longo, o nome do tipo de interface, no exemplo ITuner, sendo este o nome de identificação do componente no repositório. O outro nome, no exemplo ptun, é um nome local utilizado para se referir para o exemplo de interface particular. Esta convenção permite ter duas interfaces na borda de um componente com o mesmo tipo de interface.

A configuração dos componentes é feita de forma que cada interface de um componente pode ser ligada ou requerer zero ou muitas interfaces providas por outros componentes, demonstrada na composição de componentes para a plataforma de Tv:

```

component CTvPlatform
{
    provides IProgram pprg;
    requires I2c slow, fast;
    contains
    component CFrontEnd cfre;
    component CTunerDriver ctun;
    connects
    pprg = cfre.pprg;
    cfre.rtun = ctun.ptun;
    ctun.ri2c = fast;
}

```

Na implementação do Modelo Koala, os componentes são projetados independentemente uns dos outros. Segundo o autor, a ferramenta faz a leitura de todos os componentes e descrições de interfaces no sentido *top-down*. Todos os subcomponentes são recursivamente instanciados até que a ferramenta Koala gere um gráfico direcionado dos módulos, interfaces e ligações.

Para cada módulo, Koala gera um arquivo de cabeçalho com as macros, conforme mostrado na Figura 10, onde o *módulo m* implementa a função *p\_f*, enquanto que o *módulo n* referencia a função como *r\_f*.

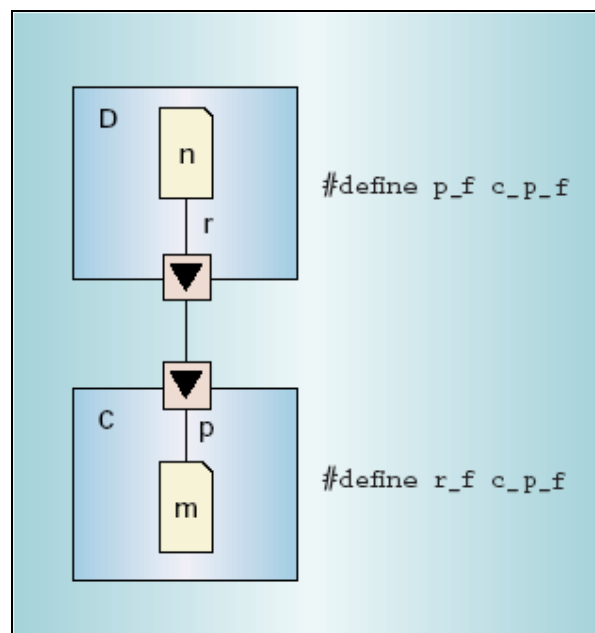


FIGURA 10 – Ligação Koala(Ommering at al, 2000)

Segundo (Ommering at al., 2000) o Koala é considerada uma ferramenta bastante eficaz no desenvolvimento de famílias de produtos eletrônicos, especificamente

aparelhos de Tv, apresentando uma estatística de mais de cem desenvolvedores de *software* utilizando o Koala.

Na próxima seção serão abordadas algumas ferramentas de configuração para aplicações embutidas, parte do estudo deste trabalho.

### 2.4.3. eCos - Embedded Configurable Operating System

O eCos<sup>8</sup>, é um sistema operacional configurável para aplicações embutidas. O ambiente é baseado no conceito de *framework*, onde cada componente do sistema (framework) pode ser reutilizado na construção de aplicações.

Através da ferramenta eCos, o desenvolvedor pode selecionar componentes que satisfazem uma determinada aplicação e configurá-los especificamente conforme os requisitos exigidos pela mesma.

A possibilidade de configurar e reutilizar seus componentes reduz significativamente o tempo de desenvolvimento de um determinado projeto. O controle sobre a configuração dos componentes (Massa ,2003) utilizado pelo eCos é o método de controle em tempo de compilação, utilizando o *linker* GNU.

Para Massa (2003), um sistema operacional embutido de tempo real carece de algumas funcionalidades básicas, incluindo controlador de interrupções e exceções, sincronização de *threads*, escalonador, temporizador e *drives* de dispositivos. eCos

---

<sup>8</sup> <http://sources.redhat.com/ecos>

disponibiliza estes componentes básicos com um kernel de tempo real como controlador central.

Seus principais componentes são:

*Hardware Abstraction Layer*(HAL) – software que provê o acesso ao *hardware*.

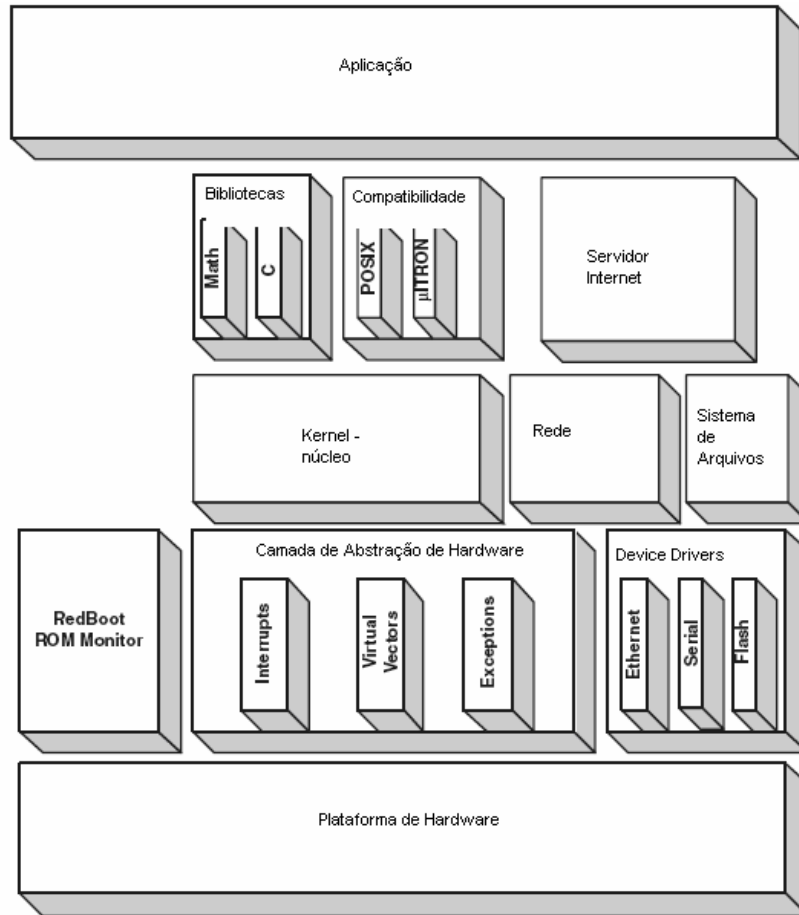
*Kernel* – inclui um tratador de interrupções e exceções, tread e sincronização, implementação do escalonador, *timers*, contadores e alarmes.

ISO C and bibliotecas – compatibilidade para chamadas de funções.

*Device drivers* – incluindo interface serial, Ethernet, Flash ROM e outros.

*Suporte GNU debugger*(GDB) – provê *software* para comunicação com *host*.

A arquitetura eCos é projetada como uma arquitetura de componentes configuráveis, consistindo em componentes de *software* tais como o Kernel e a Camada de Abstração do *Hardware*, conforme ilustrada em um exemplo na Figura 11.



**FIGURA 11 - Exemplo de Camadas para um Sistema Embutido no eCos (Massa, 2003)**

Considerando que os sistemas embutidos podem ser pequenos, rápidos ou mais sofisticados, é necessário um controle sobre todo sistema do *software* em construção. Neste caso uma forma de controlar componentes de *software* é em tempo de execução. Neste método, o código linkado na aplicação provê suporte para todos componentes requeridos ou não, como consequência, tem-se um código bastante grande. Como exemplo de controle em tempo de execução, temos uma aplicação rodando em *desktop*. Massa (2003).

Outra forma de controlar o componente é em tempo de ligação. Neste caso, o código pode ser utilizado somente para funções específicas conforme a necessidade do componente, o código que não é utilizado pela aplicação não é inserido. Uma desvantagem é a entrada das funções a serem removidas.

Ainda uma outra forma, é o controle em tempo de compilação, que fornece ao desenvolvedor o controle do comportamento dos componentes em uma fase anterior a execução, permitindo construir a implementação do próprio componente para a aplicação específica para a qual é planejado.(Massa, 2003).

Quanto à interface ao usuário, a ferramenta eCos dispõe de dois ambientes, o de configuração e o de administração de pacotes. Ver Figura 12.

O ambiente de configuração pode ser dividido em cinco partes de acordo com as preferências do usuário (eCos, 2003): *pacotes e componentes, propriedades, esquema da memória, conflitos, e saída:*

*Pacotes e Componentes:* estão organizados em forma de árvore hierárquica onde um pacote tem um ou mais componentes e outros elementos. Um pacote pode conter outro pacote, assim como um ou mais componentes.

Conforme dito anteriormente, os componentes do eCos estão codificados em linguagem C (eCos, 2003) e são portados para várias arquiteturas como ARM, Intel StrongARM, Fujitsu, SPARClite, Hitachi SH-3, Matsushita AM31/AM33, Motorola PowerPC, NEC VR4300 e Toshiba TX39. A comunicação dos componentes é feita através do *Kernel* que é responsável pela ligação dos componentes. Nesse caso o *Kernel* é de tamanho variável.

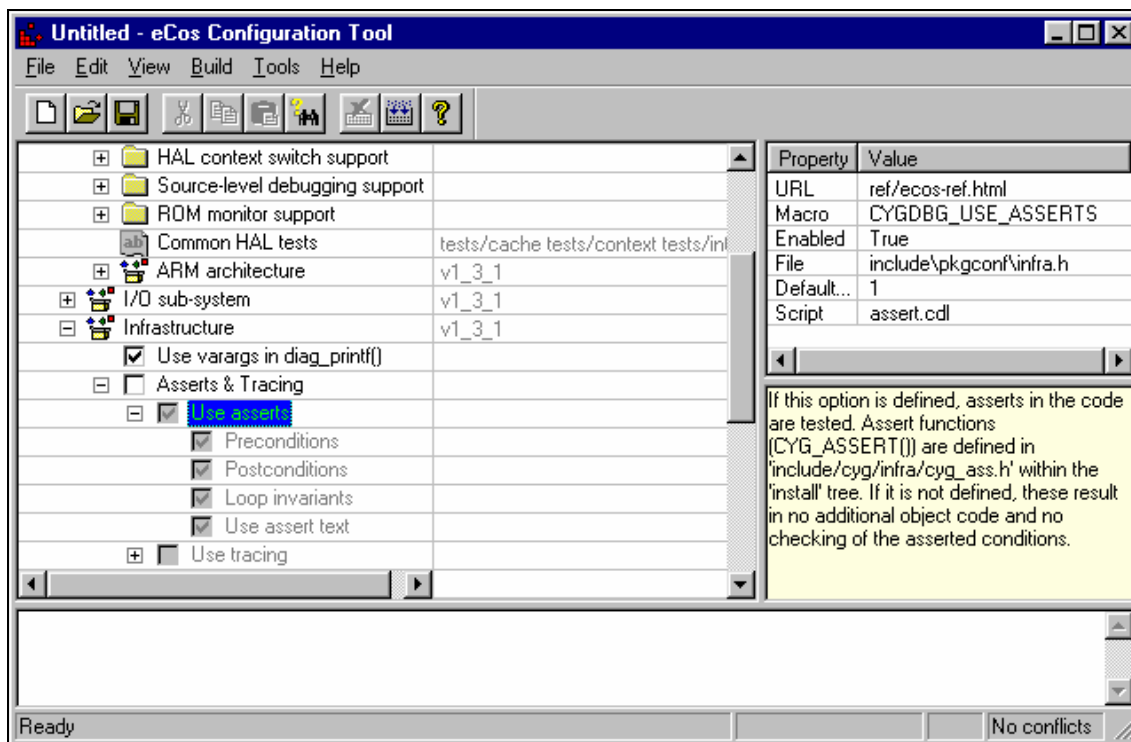


FIGURA 62 – Interface da ferramenta de configuração (eCos, 2003)

*Propriedades:* aqui são mostradas as propriedades dos pacotes e componentes como localização da sua *documentação* (tanto no próprio computador como na Internet), nome da sua *macro*, *caminho* e *nome* do arquivo onde sua *macro* se encontra (eCos, 2003).

No ambiente de administração de pacotes (eCos, 2003), o desenvolvedor pode adicionar e excluir os pacotes da ferramenta de configuração. Os pacotes estão disponíveis na Internet para *download*.

## **2.5. Conclusão**

Conforme descrito no início deste capítulo, os sistemas embutidos apresentam características singulares e ao mesmo tempo complexas a tal ponto que fica complicado iniciar cada projeto de aplicação a partir do zero. Dentro deste contexto, o conceito reutilização de componentes de *software* vem sendo bastante utilizado em diversas ferramentas com o propósito de auxiliar na construção de aplicações embutidas. Neste sentido, este capítulo iniciou realizando um levantamento sobre as características básicas dos sistemas embutidos, seguido pelo levantamento de metodologias abordadas pelos autores como Wolf(2001), trazendo uma abordagem *top-down* em cinco níveis de abstração (requisitos, especificação, arquitetura, projeto de componentes e integração do sistema).

O uso de componentes de *software* na modelagem e projeto de sistemas embutidos possibilita o reaproveitamento destes componentes na construção de diferentes aplicações. Neste sentido, fez-se um estudo sobre linguagens que auxiliem na modelagem e descrição destes componentes dentro de uma ferramenta de projeto, bem como uma análise de algumas ferramentas já existentes com o propósito de auxiliar na construção do modelo proposto neste trabalho. Dentre estas ferramentas, deu-se maior ênfase às ferramentas eCos, abordada por Massa(2003) e o Modelo Koala por (Ommering at al, 2000), já que ambas foram o referencial básico para a construção da ferramenta proposta neste trabalho.



## **CAPÍTULO 3**

### **A FERRAMENTA SIGAE**

A ferramenta Sigae, proposta nesta dissertação, foi desenvolvida com o objetivo de auxiliar um projetista de aplicações embutidas a construir sistemas em um ambiente onde seja possível otimizar os critérios de escolha dos componentes.

Conforme já exposto na seção 2.2, Wolf (2001) propõe um projeto baseado em uma metodologia *top-down*, seguindo os seguintes níveis de abstração: requisitos da aplicação, especificação, arquitetura, componentes e integração do sistema. Abordagem esta presente no modelo Sigae para configuração dos componentes.

Ao pensar em desenvolver uma aplicação baseada em componentes, é importante que, neste ambiente, esteja explícito o que é o componente (subrotina, objeto, processo); a forma como os componentes estão interligados, o mecanismo de reutilização do componente e se possível auxiliar na geração do código do mesmo. (Vilela, 2001).

Neste capítulo serão apresentados Modelo proposto em maiores detalhes, a constituição dos módulos, a interação dos componentes e as principais características do

ambiente que possibilitam sua utilização como suporte para desenvolvimento e geração de aplicações embutidas.

É suposto que os componentes de uma determinada aplicação serão inseridos no ambiente e incluídos em um repositório específico, excluindo do escopo deste trabalho a implementação de componentes propriamente ditos.

A ferramenta Sigae, resultado deste trabalho, foi desenvolvida no laboratório de projetos do curso de Ciência da Computação na Universidade do Sul de Santa Catarina.

### **3.1. Visão Geral**

A ferramenta desenvolvida neste trabalho foi idealizada de forma a tratar os problemas decorrentes da grande variedade dos componentes computacionais: o alto custo e o tempo gasto no desenvolvimento de projetos a partir do zero. A partir do momento em que você dispõe um repositório de componentes em um ambiente que possibilite a reutilização de componentes e aplicações, fica evidente a redução do tempo e custo no desenvolvimento de um projeto direcionado à uma aplicação embutida. (La Rocha, 2003).

Neste modelo, o desenvolvedor pode utilizar a ferramenta Sigae com dois objetivos bem específicos: inserir novos componentes no repositório e/ou abrir um projeto de *software* embutido baseado nos componentes existentes no repositório da ferramenta.

A ferramenta é composta por uma interface gráfica, de onde tem-se acesso aos principais módulos do sistema: descrição, ligação e compilação; ilustrados na Figura 13 deste trabalho.

Para inclusão de novos componentes no repositório, uma interface gráfica viabiliza a descrição dos componentes do sistema. Esta descrição consiste em informações detalhadas sobre o componente que esteja sendo inserido, tais como suas propriedades, operações e dependências.

Conforme já descrito no Capítulo 2, ferramentas como o Koala e o eCos utilizam uma linguagem para descrição dos componentes inseridos no repositório do ambiente,

ambas utilizam a linguagem CDL (*Component Definition Language Overview*) para a descrição, sendo que no Modelo Koala utiliza-se a linguagem IDL (*Interface Definition Language*) para a definição das interfaces dos componentes.

A descrição do componente na Ferramenta *Sigae* utiliza a linguagem de descrição de componentes LDEC<sup>9</sup> (Linguagem de descrição de Componente), que possibilita tanto a descrição do componente quanto de suas interfaces. Os componentes de *software* também podem ser incluídos na ferramenta através da importação de componentes externos, que neste caso, deverá estar descrito de acordo com a LDEC, sendo este um arquivo texto com extensão “.dec”.

As informações inseridas neste arquivo serão utilizadas para descrever posteriormente a aplicação durante o processo de desenvolvimento. Os módulos de descrição e ligação estão intimamente interligados, possibilitando editar a descrição em qualquer fase do projeto. Na fase de descrição são contemplados os níveis de requisitos, especificação e arquitetura proposta por Wolf (2001). Considerando que a grande maioria dos componentes destinados às aplicações embutidas são desenvolvidos em linguagem C ou C++, uma exigência da ferramenta é que, para cada componente inserido sejam incluídos dois tipos de arquivos, os arquivos de cabeçalho e o arquivo de código, referentes aos mesmos nesta linguagem. Sendo que os arquivos de código podem vir em código aberto ou fechado.

Os *requisitos da aplicação*, são especificações para a montagem da aplicação, onde nelas estão contidas informações direcionadas ao projeto que esteja sendo desenvolvido. Os requisitos da aplicação incluem a inserção de novos componentes e suas configurações específicas para o projeto de uma determinada aplicação.

---

<sup>9</sup> Linguagem própria da Ferramenta *Sigae*

Finalmente, a ferramenta Sigae gera o arquivo do kernel (núcleo), um arquivo *makefile*, que será cadastrado pelo compilador conforme a arquitetura de *hardware* e o arquivo executável da aplicação.

Os arquivos fontes do núcleo contêm as informações sobre os componentes e sua inicialização. O arquivo *makefile* contém as informações necessárias para gerar o executável da aplicação, tais como, os nomes dos arquivos que serão compilados, detalhes sobre o processador, os comandos de chamada dos programas relacionados ao compilador do *hardware* e opções para geração do código executável. Depois de gerar o arquivo *makefile*, o sistema chama o compilador do *hardware*, gerando o arquivo executável da aplicação. Este código executável da aplicação serão as instruções de máquina referente ao microcontrolador ou processador destinado ao projeto deste *software*.

### 3.2. Arquitetura da Ferramenta

Segue abaixo a descrição dos módulos principais da ferramenta, conforme ilustrado na Fig.13.

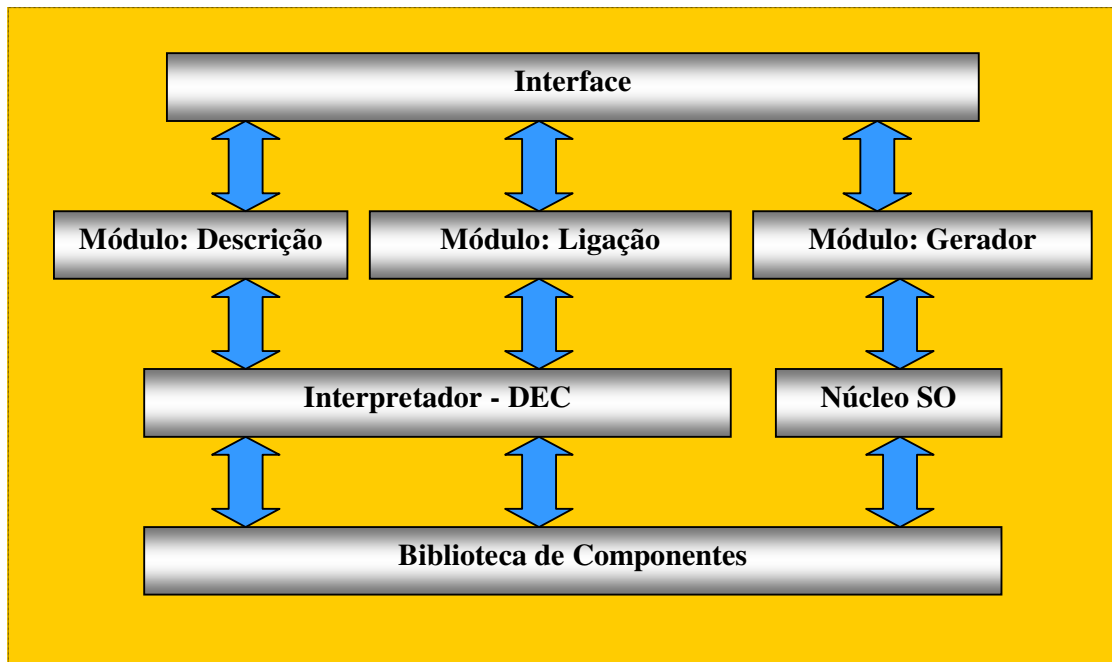


FIGURA 13 – Arquitetura da Ferramenta Sigae

#### 3.2.1. Módulo Descrição

Este módulo tem a função de possibilitar a descrição do componente no repositório da ferramenta. Através da descrição do componente na linguagem LDEC é possível inserir e manipular as informações de um determinado componente no repositório. O interpretador, que possibilita a manipulação das informações do

componente, foi desenvolvido com base na estrutura da linguagem LDEC, sendo que o mesmo foi gerado através da ferramenta Gals<sup>10</sup>.

Na inclusão de um componente no repositório, o interpretador analisa o arquivo de descrição de componentes (LDEC), verificando se o componente está descrito de forma correta, sem erros léxicos, sintáticos e semânticos<sup>11</sup> e retorna estas informações ao módulo de descrição. A partir destas informações, são resgatados os nomes dos arquivos, os identificadores do componente, a arquitetura a qual pertence o componente e sua categoria, criando em seguida a estrutura de pastas no repositório específicas deste componente. O repositório fica organizado em uma estrutura que segue o seguinte esquema: *arquitetura -> categoria*.

O módulo de descrição atua na inserção dos dados da arquitetura do componente inserido no arquivo *arquit.bdc*, sendo que este contém a lista de arquiteturas disponíveis no repositório e; inserir na lista de categorias - arquivo *categ.bdc*, a categoria do componente.

As informações do nome do arquivo de descrição do componente e o identificador do componente na lista de componentes da categoria ficam no arquivo "*Comp.bdc*". Qualquer informação relativa a um determinado componente é automaticamente removida caso ocorra algum erro de inclusão do componente.

Na remoção de componentes do repositório, o módulo de descrição abre o arquivo de descrição de componentes (LDEC), localizado no repositório, na pasta de sua respectiva arquitetura e categoria do componente. O procedimento aciona o interpretador de modo que o mesmo obtenha as informações necessárias para a remoção

---

<sup>10</sup> Ferramenta desenvolvida por Carlos Gesser – Disponível em: <http://www.inf.ufsc.br/~gesser/>

<sup>11</sup> A análise léxica está relacionada aos caracteres válidos (tokens), a análise sintática relacionada à seqüência lógica das informações e a análise semântica verifica se as informações estão consistentes.

do componente do repositório. Basicamente o interpretador resgata o identificador do componente e os arquivos que o compõe. A partir destas informações, o procedimento remove no repositório os arquivos do componente, retirando do arquivo de lista de componentes, arquivo “*Comp.bdc*”, seu registro.

### **3.2.2. Módulo Ligação**

(Ommering et al., 2000) define que uma configuração é um conjunto de componentes conectados de forma a compor um determinado produto. Seguindo a linha do modelo Koala, este módulo atua na manipulação dos componentes para configurar uma determinada aplicação.

Neste módulo é efetuada a escolha dos componentes (de sistema operacional, aplicações e dispositivos) para serem ligados e configurados, juntamente com a inserção do código principal da aplicação. O interpretador também atua neste módulo extraindo as informações de descrição do componente.

Na ferramenta fica disponível uma lista de componentes que podem ser incluídos para o projeto de uma determinada arquitetura. Após a inclusão dos componentes, é feita a configuração de suas propriedades, sendo que o acesso de edição de cada propriedade vai depender de como o componente foi construído e se ela consta no arquivo de especificação do mesmo.

A ligação propriamente dita dos componentes é feita através do componente “aplicação”, onde ficam disponibilizadas automaticamente todas as operações dos



componentes selecionados. Através deste componente o usuário projetista escreve o núcleo do código da aplicação, que deve ser escrito em linguagem C.

### **3.2.3. Módulo Gerador**

O Módulo Gerador verifica, através de uma chamada ao Módulo de Ligação, se foram preenchidos todos os requisitos dos componentes inseridos na compilação e verifica se o arquivo do projeto foi salvo em disco. Após, este procedimento, chama no Módulo de Ligação a Geração do Código fonte da Aplicação, sendo estes, os arquivos Nucleo.c e Nucleo.h., as informações contidas nestes arquivos serão utilizadas pelo Kernel do sistema operacional da ferramenta, na etapa de compilação.

Seqüencialmente, na compilação, a partir da arquitetura do projeto, o módulo localiza na lista de compiladores, o que compila para a arquitetura configurada inicialmente no início do projeto.

Na geração propriamente dita, o sistema chama o programa *make*, que contém através do arquivo *makefile*, os endereços onde estão os arquivos do sistema operacional e da aplicação, a forma como deve proceder a compilação e a chamada para os programas que compilam os códigos e geram o executável.

### 3.2.4. Núcleo de Componentes do Sistema Operacional

Módulo composto por um núcleo que faz o gerenciamento dos componentes do sistema e a interface com o controlador ( módulo que faz a abstração do *hardware* alvo da aplicação ).

Segundo Massa(2003), uma interrupção é um evento externo assíncrono que ocorre durante a execução de um programa, causando uma parada de execução normal. Existem diversas formas de tratar estas interrupções, algumas arquiteturas de processadores suportam vetores individuais para determinados tipos de interrupções, outras apresentam um vetor único para todas interrupções. No primeiro caso, o Tratador de Interrupções pode estar ligado diretamente ao vetor e processar quando ocorrer uma determinada interrupção. No caso de suportar vetor único, o *software* deve determinar que interrupção ocorreu antes de processar o Tratador de Interrupções.

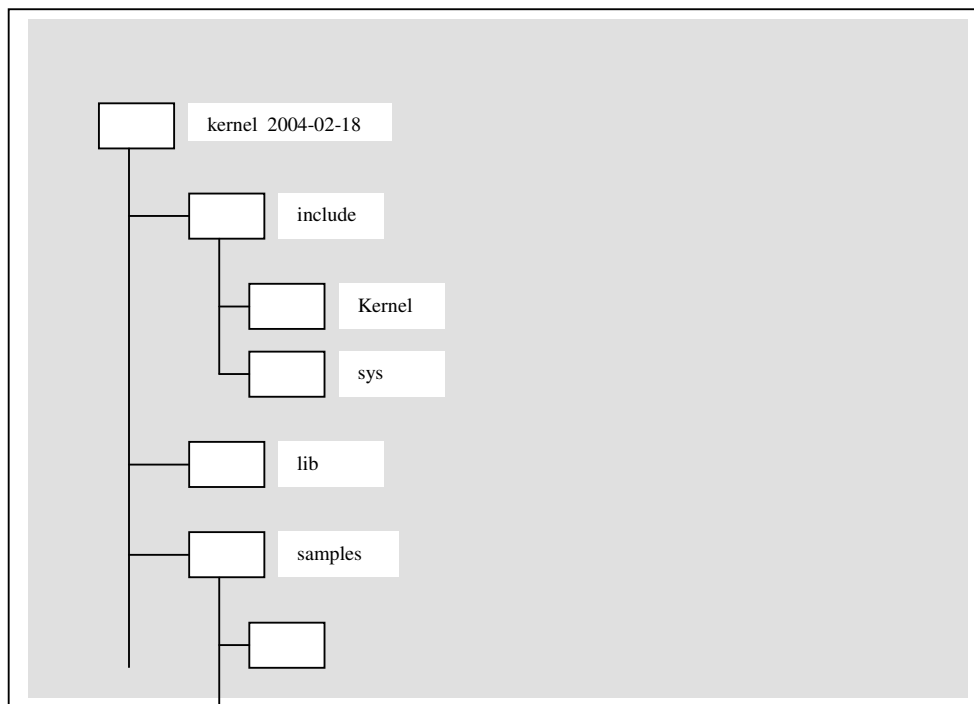


FIGURA 14 – Estrutura de Diretório do Componente SO Sigae

Ainda segundo Massa(2003), uma das preocupações fundamentais em sistemas embutidos com respeito a interrupção é latência. Latência é o intervalo de tempo entre a ocorrência da interrupção até a execução da Rotina de Tratamento.

A acesso ao Módulo Kernel é feito através de chamadas de sistema (*system calls*), conforme ilustrado na Quadro 1, e através do Tratador de Interrupções onde Núcleo efetua o gerenciamento de todo sistema embutido. Através do componente Tratador de Interrupções, cada requisição efetuada por um componente ligado ao controlador ou da aplicação, uma rotina de Tratamento de Interrupção é executada tendo-se uma interrupção a ser tratada pelo Kernel.

```

....
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type __##name(type1 arg1,type2 arg2,type3 arg3) \
{ \
int __res; \
__asm__ __volatile__ ( \
    "ldd r20,%A4\n\t" \
    "ldd r21,%B4\n\t" \
    "ldd r22,%A3\n\t" \
    "ldd r23,%B3\n\t" \
    "ldd r24,%A2\n\t" \
    "ldd r25,%B2\n\t" \
    "rcall system_call\n\t" \
    : "=r" (__res) \
    : "x" (__NR_##name), "m" ((int)arg1), "m" ((int)arg2), "m" ((int)arg3)); \
if (__res < 0) { \
    errno = -__res; \
    __res = -1; \
} \
return __res; \
} \
#ifdef __KERNEL_SYSCALLS__

static inline _syscall1(int,close,int,fd)
static inline _syscall3(int,ioc1,unsigned int,fd,unsigned int,cmd,unsigned long,arg)
static inline _syscall3(off_t,lseek,int,fd,off_t,offset,int,count)
static inline _syscall3(int,open,const char *,file,int,flag,int,mode)
static inline _syscall3(int,read,int,fd,char *,buf,off_t,count)
static inline _syscall3(int,write,int,fd,const char *,buf,off_t,count)

#endif
#endif /* _KERNEL_UNISTD_H */

```

**QUADRO 1 – Trecho Código System Calls**

O Mecanismo de Tratamento de Interrupções utilizado pelo modelo Sigae foi implementado pela linguagem C., utilizando-se um vetor de interrupções padrão.

A inicialização do sistema é feita a partir de um arquivo de código em linguagem *assembly* que só pode ser compilado para um tipo de microcontrolador ou processador. Para fazer a inicialização deste sistema em outros microcontroladores será necessário incluir outros arquivos de código que contenham a inicialização para cada microcontrolador que esteja cadastrado na ferramenta.

O Módulo de Componentes do Sistema Operacional está inserido em uma estrutura de diretórios específica do ambiente. Os componentes referentes ao *hardware* especificamente encontram-se localizados em `\Sigae\kernel_2004-02-18\lib`. Os componentes específicos do Sistema Operacional, localizam-se em `\Sigae\kernel_2004-02-18\include\kernel` e ilustrados na Figura 14.

```

#ifndef _KERNEL_DEVICE_H
#define _KERNEL_DEVICE_H

#include <kernel/inode.h>

#define MAX_DEVICE      5

#define DUMMY_MAJOR    0
#define UART_MAJOR     1
#define LED_MAJOR      2
#define KEY_MAJOR      3
#define LCD_MAJOR      4

struct device_t {
    int (*ioctl)(struct inode_t *inode, unsigned int cmd, unsigned long arg);
    int (*open)(struct inode_t *inode);
    int (*close)(struct inode_t *inode);
    int (*read)(struct inode_t *inode, char *buf, off_t count);
    int (*write)(struct inode_t *inode, const char *buf, off_t count);
    off_t (*lseek)(struct inode_t *inode, off_t offset, int count);
};

extern struct device_t *device_table[];

extern void device_init(void);
extern void register_device(int major, struct device_t *device);

#ifdef CONFIG_DUMMY
extern void dummy_init(void);
#endif
#ifdef CONFIG_UART
extern void uart_init(void);
#endif
#ifdef CONFIG_LED
extern void led_init(void);
#endif
#ifdef CONFIG_KEY
extern void key_init(void);
#endif
#ifdef CONFIG_LCD
extern void lcd_init(void);
#endif

#endif /* _KERNEL_DEVICE_H */

```

**QUADRO 2 – Trecho Código Kernel**

### 3.2.5. Biblioteca de Componentes (Repositório)

A Biblioteca foi implementada com um conjunto de componentes básicos para estudos de casos específicos a uma determinada estrutura de *hardware* em estudo.

Villela (2001) destaca como componentes essenciais a uma estrutura inicial elementos como sensores, atuadores, controladores voltados a aplicações de automação industrial. Neste sentido, a biblioteca dispõe de componentes básicos que possam auxiliar no desenvolvimento de aplicações embutidas, além de possibilitar a inclusão e configuração de componentes de acordo com as necessidades de uma determinada aplicação. A construção da biblioteca deve iniciar a partir destes três elementos básicos: o algoritmo (controlador), o sistema de captação de informações e o sistema de atuação no ambiente ou aplicação.

Partindo destes pressupostos, os componentes de cada arquitetura são guardados em diretórios que são formados por dois níveis. O primeiro é o nome da arquitetura e o segundo é a categoria que podem ser: atuador, interface com o usuário projetista, sensor e ainda podem ser definidas outras pelo usuário projetista. Estas pastas são armazenadas na pasta raiz do repositório.

As informações sobre as arquiteturas, categorias, componentes e compiladores de cada arquitetura são armazenados em arquivos que foram denominados de *arquivos de controle* ou *banco de dados de componentes*, cujo a extensão é “.dbc”. Nos arquivos de controle ficam as listas de pastas de arquitetura, categorias de componentes, identificadores de componentes e o nome de seus arquivos. Foi adotado usar esses arquivos para evitar que usuários sem muito conhecimento do funcionamento do

repositório, adicionem componentes manualmente. Cada um desses arquivos é colocado em um ponto diferente do repositório. São Os arquivos de controle:

- “*Arquit.bdc*”: neste arquivo fica uma lista que guarda os nomes de cada arquitetura inserida na ferramenta. Este arquivo se encontra dentro da pasta do repositório.
- “*Categ.bdc*”: neste arquivo fica uma lista que guarda os nomes de cada categoria que existem em uma determinada arquitetura inserida na ferramenta. Em cada pasta de arquitetura há um arquivo com este nome.
- “*Comp.bdc*”: neste arquivo ficam listas que guardam os nomes, os identificadores e os nomes dos arquivos de cada componente que existe em uma determinada categoria de uma determinada arquitetura inserida na ferramenta. Em cada pasta de categoria de cada arquitetura há um arquivo com este nome.
- “*Cmpldrs.bdc*”: neste arquivo fica uma lista que guarda os dados sobre os compiladores para cada arquitetura que está inserida na ferramenta. Este arquivo se encontra dentro da pasta do repositório.

### **3.3. Descrição dos Componentes na Ferramenta**

As linguagens abordadas no Capítulo 2 não foram utilizadas como parte integrante da ferramenta, mas contribuíram para a construção de uma linguagem que melhor atendesse as necessidades da ferramenta, descrição dos componentes, descrição da

interface e que possibilite a fácil ligação destes na aplicação. Em Massa (2003) e Koala (2000), utiliza-se para descrição a linguagem CDL. Entretanto esta linguagem não possibilita a descrição das interfaces e conseqüentemente dificulta o processo de ligação. Em Koala (2000), a descrição das interfaces é efetuada através da linguagem IDL (*Interface Definition Language*).

Tendo em vista a necessidade de utilizar-se neste projeto uma linguagem que possibilite tanto a descrição do componente e suas interfaces, bem como o resgate dos dados para análise nos requisitos da aplicação, desenvolveu-se uma linguagem específica para a ferramenta, que possibilita descrever o componente, suas propriedades, operações, tipos, enumerações, requisitos e interface gráfica de ligação com demais componentes e seu respectivo interpretador descritos no próximo tópico.

### **3.3.1. A Linguagem de Descrição de Componentes**

A linguagem denominada LDEC (linguagem de descrição de componente) possui as seguintes características:

- é validada a partir de um arquivo de descrição com extensão “.dec”;
- diferencia maiúsculo de minúsculo;
- Letras: caracteres de “A” até “Z”, “a” até “z” e o caractere “\_”;
- Dígitos: caracteres de “0” até “9”
- Operadores: = >> << \$ :



- Símbolos Especiais: { }
- Delimitadores – Os caracteres espaço, quebra de linha, ponto e vírgula “;” ou os comentários podem ser usados como separadores
- Comentários: Um comentário pode ser inserido em qualquer lugar do programa onde um delimitador é válido. A duas formas de comentário: de apenas uma linha que inicia com “//” e termina no final da linha; e de múltiplas linhas que é delimitado por “/\*” e “\*/”.

A descrição do componente é efetuada através da inclusão de comandos da linguagem específicos para a descrição de um componente. Estes comandos<sup>12</sup> são solicitados pelo módulo de descrição para a inclusão deste na biblioteca de componentes da ferramenta. elementos semânticos da linguagem LDEC. São comandos da linguagem:

- *Componente*: é a palavra de identificação do componente.
- *Nome*: é o nome do componente que aparece na ferramenta.
- *Descricao*: texto descritivo sobre as características do componente.
- *Codigo e Cabecalho*: São as indicações de arquivos de código e cabeçalho. Estes são os arquivos que contém os códigos fonte do componente. Pode ser indicado um ou mais arquivos de código e cabeçalho conforme a quantidade de arquivos que o compõe.
- *Local*: é a indicação dos locais extras que são usados para a busca de arquivos que não estão na mesma pasta dos códigos fonte do componente.

---

<sup>12</sup> Comandos semânticos da linguagem

- *Arquitetura*: indica a arquitetura a que o componente pertence.
- *Imagem*: a imagem representativa do componente a ser utilizada graficamente pela ferramenta.
- *Tipo*: Serve para indicar o tipo do componente. Um componente pode ser definido como: aplicação, dispositivo ou sistema (componente do SO).
- *Categoria*: Serve para indicar a categoria a que o componente pertence.
- *Requisitos*: são dependências que precisam ser atendidas pela ferramenta para que o componente possa ser ligado à aplicação.
- *Propriedades*: Uma propriedade é um valor usado pelo componente. Estes valores podem ser variáveis, constantes e macros. Por padrão, uma propriedade é uma macro, caso não seja é necessário especificar o seu tipo. Variáveis, como o nome já diz, são valores que podem ser alterados durante a execução, constantes, por sua vez são valores que não se alteram e macros são usados para se definir um conjunto de comandos, na compilação a macro é substituída pelo seu conteúdo. As informações da propriedade são:
  - Nome da propriedade que é exibido na ferramenta.
  - Descrição da propriedade.
  - Tipo de dados da propriedade.
  - Valor inicial da propriedade e que é usado como valor padrão.
  - Valores válidos que é usado da mesma forma que uma enumeração de constantes, mas ao invés de ser uma lista de valores, é uma indicação de um valor mínimo e máximo.

- Requisito da propriedade que indica o que é necessário para que possa usar esta propriedade.
- Condição de ativação que verifica se a condição para se ativar a propriedade foi preenchida.
- *Operações*: Uma operação é uma função que pode ser executado por um componente, ou seja, são as interfaces de entrada e saída que possibilitam a ligação com os demais componentes da aplicação. Cada operação ou interface necessita das seguintes informações:
  - Nome da operação que é exibido na ferramenta.
  - Descrição da operação.
  - Nome da operação que é exibido na ferramenta.
  - Tipo de dados que a operação retorna caso a propriedade retorne algum valor.
  - Parâmetro de entrada/saída que a operação tem.
  - Requisito da operação que indica o que é necessário para que possa usar esta operação.
  - Condição de ativação que verifica se a condição para se ativar a operação foi preenchida.

Um modelo de sintaxe de descrição do componente pode ser visualizado no Quadro 3.

```

////////////////////////////////////
//          Definição de Componente          //
////////////////////////////////////

*****
*          Definição da linguagem          *
*****

O cabeçalho do arquivo é "Componente"

Sintaxe:
Componente <Nome do Componente>

- Arquivo para ser compilado
Nome = <Nome humanamente legível>
Descricao = <Descrição>
Codigo = <Nome do arquivo de código>[, <Descrição>]
Cabecalho = <Nome do arquivo de cabeçalho>[, <Descrição>]
Local = <Local para ser incluído na lista de pastas>[, <Descrição>]
Arquitetura = <Arquitetura que o componente foi feito>[, <Descrição>]
Imagem = <Nome do arquivo de imagem>[, <Descrição>]
Tipo = <Tipo de componente>[, <Descrição>]
Categoria = <Categoria do componente>[, <Descrição>]

- Propriedades

Propriedade <Nome da propriedade>
Nome = <Nome humanamente legível>
Descricao = <Descrição da Propriedade>
Tipo = <Tipo de dado>[, <Descrição>]
Valor = <Valor Inicial>[, <Descrição>]
Valores Validos = <Início>[ Ate <Final> ][, <Descrição>]
Requer = <Valor requerido>[, <Descrição>]
Ativar Se = <Macro que depende>[, <Descrição>]
Definicao = [Constante/Variavel/Macro][, <Descrição>]
Fim Propriedade [<Nome da propriedade>]

- Operações de Entrada/Saída

Operacao <Nome da Operação>
Nome = <Nome humanamente legível>
Descricao = <Descrição>
Tipo = <Tipo de dado de saída>
Parametro = <Tipo> : <Nome>[, <Descrição>]
Ativar Se = <Macro que depende>[, <Descrição>]
Requer = <Valor requerido>[, <Descrição>]
Fim Operacao [<Nome da Operação>]

- Definição de tipos externos

TipoDef <Identificador>
Definicao = <Arquivo onde implementa>
Tipo = <Tipo real do identificador>
Formato = <Formato mostrado na caixa de texto>
Descricao = <Descrição>
Fim TipoDef [<Identificador>]

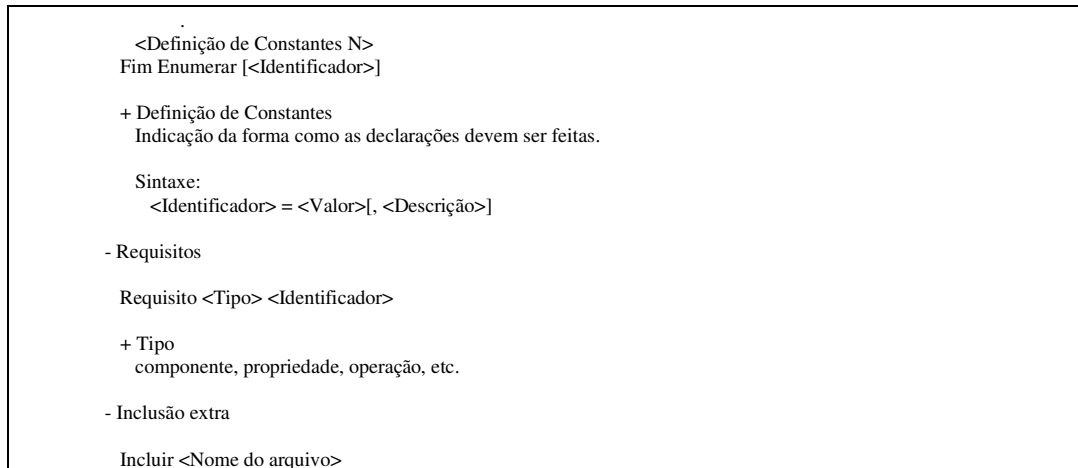
+ Tipos de estruturas
{ <Variavel 1> ; ... ; <Variavel N> }

+ Definição Variável
<Identificador> Como <Tipo>
<Tipo> <Identificador>

- Enumeração

Enumerar <Identificador> Como <Tipo>
Descricao = <Descrição>
<Definição de Constantes 1>
<Definição de Constantes 2>
.
.

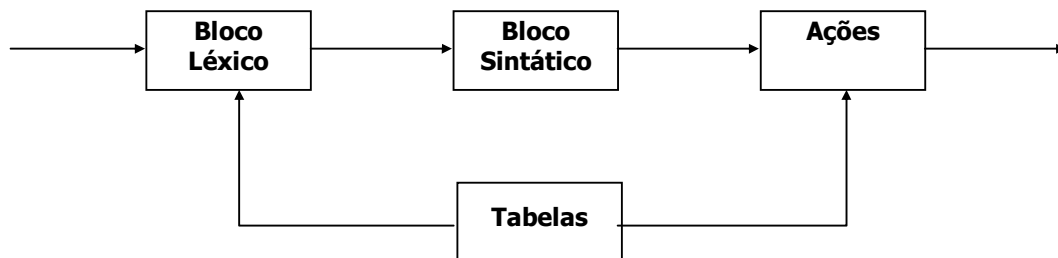
```



**QUADRO 3 – Modelo de Descrição do Componente**

### 3.3.2. O Interpretador da Linguagem de Descrição de Componentes

Na ferramenta Sigae, todo resgate e interpretação das informações são realizados pelo interpretador. O Módulo de Análise obtém os dados do arquivo do componente existente no repositório. O resgate e a interpretação destas informações, são realizados por um interpretador desenvolvido para o sistema.



**FIGURA 15 – Estrutura do Interpretador LDEC**

Conforme ilustrado na Figura 15, em um modelo simples de interpretador o trabalho pode ser feito por uma interconexão serial entre três blocos chamados: *bloco léxico*, *bloco sintático* e *bloco semântico* ou *ações do interpretador*. Os três blocos têm acesso a tabelas de informações globais sobre o programa fonte. Desta forma, o objetivo de um interpretador é traduzir as seqüências de caracteres que representam o programa fonte em ações que incorporam a intenção do projetista. (Setzer, 1986).

O Interpretador da linguagem LDEC é formado por três módulos: o analisador léxico, analisador sintático e analisador semântico.

O Analisador Léxico obtém os tokens do texto descrito na linguagem LDEC, relativo ao componente que esteja sendo analisado, e os envia ao analisador sintático. O analisador sintático constitui no coração do processo de tradução (tradução dirigida por sintaxe). A análise semântica é dirigida pela sintaxe, ou seja, durante o processo de análise sintática são disparadas ações de verificação semântica e de inserção de dados dos componentes selecionados para a aplicação. (Aho, 1995). Os diagramas sintáticos estão disponíveis no Anexo 1 deste trabalho.

Os elementos léxicos serão definidos através do conjunto de caracteres e símbolos válidos para a linguagem “LDEC”. Com isto os elementos léxicos ou tokens, subdividem-se em classes.

A classe *identificadores* é definida por um conjunto de letras e números, sendo o primeiro caractere uma letra seguido de letras e/ou números.

Na classe *números*, pode-se descrever por seis modos: números inteiros na base decimal, binário, octadecimal e hexadecimal e números de ponto flutuante com ou sem exponencial.

A classe *literais* é definida por um conjunto de letras dígitos e símbolos. O literal pode ser utilizado de duas formas, como um texto (vetor de caracteres) ou como um valor. Quando for usado como texto é usado aspas. Quando for usado como valor é usado o apóstrofe e, neste caso, o literal é usado como valor numérico.

Os operadores podem ser :

- *Operador de atribuição* “=”: este operador serve para atribuir um valor ao comando indicado.
- *Operadores de ponteiro* “\$” “>>” e “<<”: servem para definir um tipo de dado como ponteiro.

As palavras reservadas foram classificadas conforme abaixo:

- *Informações*: são as palavras reservadas que indicam as informações sobre o componente. As palavras reservadas são: Componente, Nome, Descrição, Código, Cabeçalho, Local, Arquitetura, Imagem, Tipo, Categoria, Valor, Valores, Validos, Requisito, Ativar, Se, Definição, Até, Formato, Propriedade, Fim, Constante, Variável, Macro, Operação, Parâmetro, TipoDef, Enumerar, Como, Incluir.
- *Tipos de dados*: são as palavras reservadas que indicam o tipo de uma propriedade, o retorno de uma operação bem como o tipo dos parâmetros que são passados. As palavras reservadas são: IntMnc, IntCrt, IntPqn, IntMed, Inteiro, NatMnc, NatCrt, NatPqn, NatMed, Natural, DecPqn, DecMed, DecGrd, Decimal, BoolMnc, BoolCrt, BoolPqn, BoolMed, Booleano, PntMnc, PntCrt, PntPqn, PntMed, Ponteiro, CarNrm, CarEst, Caracter, LtrNrm, LtrEst, Literal, Estrutura, Classe, Enumeração;

A análise semântica é dirigida pela sintaxe, ou seja, durante o processo de análise sintática são disparadas ações de verificação semântica e de inserção de dados dos componentes selecionados para a aplicação.

As ações semânticas<sup>13</sup> estão divididas em cinco grupos:

- o grupo 1 é formado pelas ações de conversão e obtenção dos dados (1 à 17);
- o grupo 2 é formado pelas ações de obtenção das informações gerais sobre o componente(100 à 117);
- o grupo 3 é formado pelas ações que obtêm as propriedades do componente(200 à 215);
- o grupo 4 é formado pelas ações que obtêm as informações sobre uma operação do componente (300 à 312);
- o grupo 5 é formado pelas ações que obtêm o tipo de dado do usuário (400 à 408 e 500 à 506).

---

<sup>13</sup> A descrição completa das ações semânticas utilizadas pelo interpretador está descrita no Anexo 2.



### **3.4. Geração do Código**

Ao incluir os componentes necessários da aplicação, em tempo de seleção de componentes, pode-se visualizar a geração do código fonte da aplicação no próprio componente denominado “aplicação”.

Um componente de aplicação pode ser formado pela composição de outros componentes e/ou aplicações, que por sua consequência tornam-se componentes de uma aplicação maior. Neste caso, o relacionamento entre os componentes é feito diretamente na caixa de edição de texto da aplicação, onde ficam disponibilizadas as operações de cada componente inserido no projeto e suas interfaces de entrada e saída.

Concluídas as etapas de desenvolvimento da aplicação é feita a geração do código executável, efetuado pelo compilador inserido na ferramenta para a arquitetura do projeto.

O resultado final de todo o processo de geração de código é definido por dois tipos de conjuntos de arquivos, os que definem um componente e os que definem o código: o arquivo de cabeçalho (\*.h); o arquivo do componente (\*.c) e o arquivo do projeto (*makefile*), ilustrado no Quadro 4.

```
#####
#                               #
#####

# Makefile gerado pelo SIGAE

# ***** Informações de compilação ***** #

# Arquitetura do projeto
MCU = atmega103

# Pasta do repositório
REP = C:/SIGAE/Executavel/Repositorio/

# Pasta dos compiladores
CMPPST =

# Compiladores
CMP1 = $(CMPPST)avr-gcc.exe
LNG1 = $(CMPPST)avr-gcc.exe
CNVCP1 = $(CMPPST)avr-objcopy.exe
CNVCP2 = $(CMPPST)avr-objcopy.exe
CNVDB3 = $(CMPPST)avr-objdump.exe

# Informações do compiladores
CFLAGS = -g -Os -Wall -mmcu=$(MCU) -D_APLICACAO_TESTE_=1 -I "$(REP)Nucleo/include" -I
"$(REP)Nucleo/ATMega103" -I "$(REP)ATMega103/Interface" -I "$(REP)ATMega103/Rotinas" -I
"C:/Documentos/Aula/Projeto/Prototipo/Executavel/Exemplos/Teste"

# Informações dos ligadores
LDFLAGS = -mmcu=$(MCU) -lm

# ***** Compilação dos arquivos ***** #
all:
"$(CMP1)" $(CFLAGS) -o "Nucleo.o" -c "Nucleo.c"
"$(CMP1)" $(CFLAGS) -o "led.o" -c "$(REP)ATMega103/Interface/led.c"
"$(CMP1)" $(CFLAGS) -o "teste.o" -c "teste.c"
"$(CMP1)" $(CFLAGS) -o "lcd.o" -c "$(REP)ATMega103/Interface/lcd.c"
"$(CMP1)" $(CFLAGS) -o "key.o" -c "$(REP)ATMega103/Interface/key.c"
"$(CMP1)" $(CFLAGS) -o "__libc.o" -c "$(REP)Nucleo/lib/__libc.c"
"$(CMP1)" $(CFLAGS) -o "device.o" -c "$(REP)Nucleo/lib/device.c"
"$(CMP1)" $(CFLAGS) -o "errno.o" -c "$(REP)Nucleo/lib/errno.c"
"$(CMP1)" $(CFLAGS) -o "fs.o" -c "$(REP)Nucleo/lib/fs.c"
"$(CMP1)" $(CFLAGS) -o "init.o" -c "$(REP)Nucleo/lib/init.c"
"$(CMP1)" $(CFLAGS) -o "inode.o" -c "$(REP)Nucleo/lib/inode.c"
"$(CMP1)" $(CFLAGS) -o "func.o" -c "$(REP)Nucleo/lib/func.c"
"$(CMP1)" $(CFLAGS) -o "entry.o" -c "$(REP)Nucleo/ATMega103/entry.S"
"$(CMP1)" $(CFLAGS) -o "setup.o" -c "$(REP)Nucleo/ATMega103/setup.S"
"$(LNG1)" $(LDFLAGS) -o "teste.elf" "Nucleo.o" "led.o" "teste.o" "lcd.o" "key.o" "__libc.o" "device.o" "errno.o" "fs.o"
"init.o" "inode.o" "func.o" "entry.o" "setup.o"
"$(CNVCP1)" -O ihex -R .eeprom "teste.elf" "teste.hex"
"$(CNVCP2)" -O binary -R .eeprom "teste.elf" "teste.bin"
"$(CNVDB3)" -D "teste.elf" > "teste.lst"

# ***** Limpeza dos arquivos ***** #
clean:
rm -f *.o *.elf *.hex *.bin *.lst
```

**QUADRO 4 – Exemplo arquivo *makefile***

### 3.5. O projeto da ferramenta

Nesta seção serão mostrados alguns diagramas que explicam o projeto da ferramenta de desenvolvimento e geração de softwares para aplicações embutidas descrita nas seções anteriores deste capítulo.

Inicialmente será mostrado o diagrama de atividades que especificam as atividades envolvidas na construção de um *software* utilizando a ferramenta Sigae ( Figura 16).

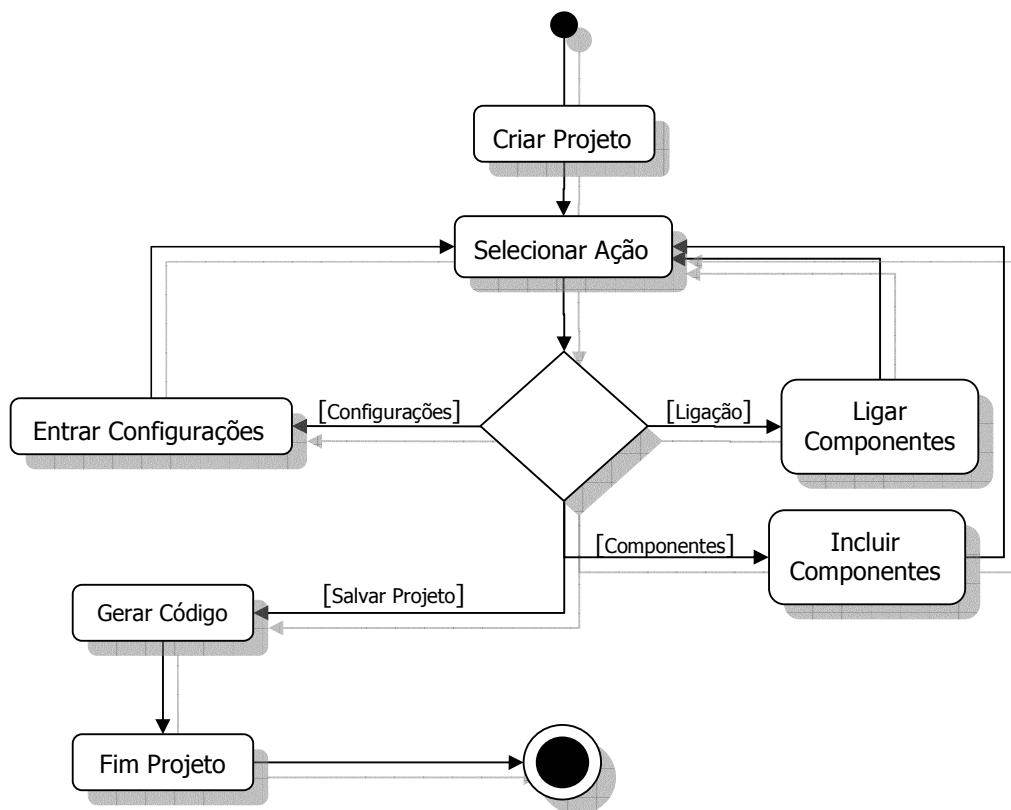
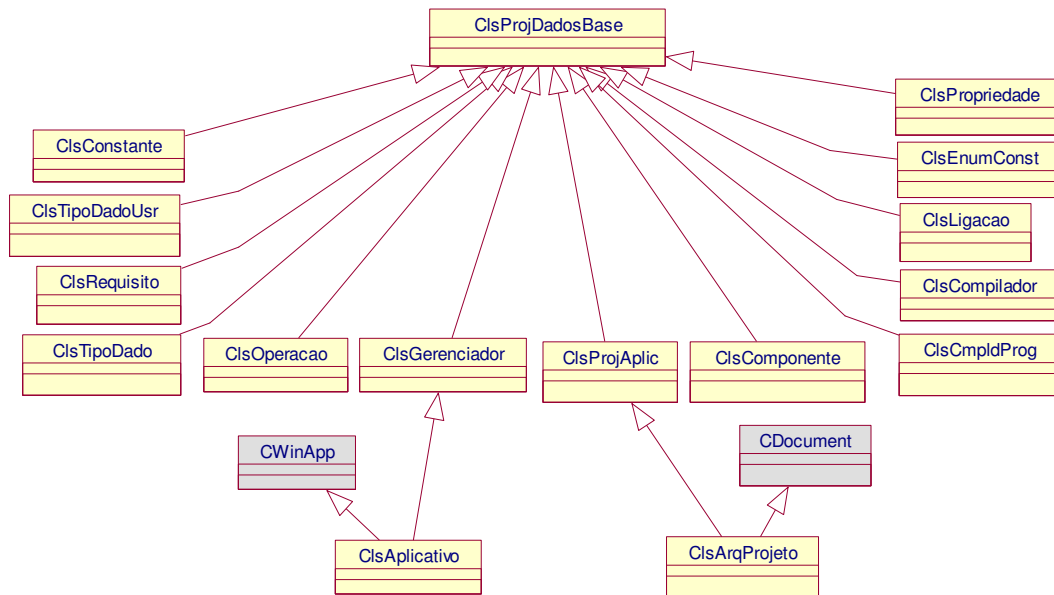


FIGURA 16 - Diagrama de Atividades – desenvolvimento do *software* usando o Sigae

Este diagrama ilustra basicamente os tipos de atividades possíveis: incluir novos componentes, configurar e/ou construir um novo *software*. Na inclusão de novos componentes, o usuário insere todas as informações referente a um determinado componente, seus arquivos objeto e de cabeçalho, através de uma interface de edição, na edição também são fornecidas informações relativas as interfaces de entrada e saída e as respectivas operações disponibilizadas pelo componente que esteja sendo inserido.

Na construção de um *software* para uma aplicação específica, o usuário seleciona os componentes disponíveis no repositório através de uma interface gráfica, onde podem ser feitas as ligações. Um componente denominado aplicação deve ser inserido de modo que possibilite a configuração dos componentes e a visualização de suas operações e interfaces.

Na Figura 17 tem-se o diagrama das principais classes envolvidas no desenvolvimento de um *software* para um sistema embutido.



**FIGURA 17 - Diagrama de Classes da Ferramenta**

A Classe Projeto(ClsProjAplic) mostrada no Quadro 5 contém os métodos e informações para a construção do *software* pelo projetista. pa aplicação do usuário projetista. Esta classe manipula informações de identificação do *software*, arquitetura do *hardware* e a lista de componentes e respectivas ligações entre os mesmos. A classe projeto possui quatro métodos:

- *Abrir* -> abrir um projeto armazenado em memória secundária;
- *Salvar* -> armazenar as informações e dados de um projeto de *software* em memória secundária (disco rígido);
- *VerificarRequisitos* -> avaliar a consistência da composição de componentes efetuada em um determinado *software* verificando se os requisitos especificados por cada componente foram atendidos;
- *Gerar* -> gerar os códigos fonte utilizados na geração do *software* embutido;

| ClsProjAplic   |  |
|--|--|
| ClsProjAplic()   |  |
| ClsProjAplic(sAp : CtLiteral)  |  |
| ClsProjAplic(cObj : RfCtClsProjAplic) : explicit                           |  |
| ~ClsProjAplic()  |  |
| ArqProjeto() : CtLiteral   |  |
| ArqProjeto(vVal : CtLiteral) : void  |  |
| PstDestino() : CtLiteral   |  |
| PstDestino(vVal : CtLiteral) : void  |  |
| NomeArquivo() : CtLiteral  |  |
| NomeArquivo(vVal : CtLiteral) : void                                       |  |
| Arquitetura() : CtLiteral  |  |
| Arquitetura(vVal : CtLiteral) : void                                       |  |
| Compilador() : CtLiteral   |  |
| Compilador(vVal : CtLiteral) : void  |  |
| LstComponentes() : RfListaComponentes                                      |  |
| LstComponentes() : RfCtListaComponentes                                    |  |
| LstComponentes(vVal : RfCtListaComponentes) : void                         |  |
| LstLigacoes() : RfListaLigacoes  |  |
| LstLigacoes() : RfCtListaLigacoes  |  |
| LstLigacoes(vVal : RfCtListaLigacoes) : void                               |  |
| Alterado() : Booleano  |  |
| Alterado(bAlt : Booleano) : void   |  |
| PstTemp() : CtLiteral  |  |
| Limpar() : void  |  |
| VerificarRequisitos() : Booleano   |  |
| Gerar(sListaArquivos : PtListaString, sMsg : PtString = Vazio) : Booleano  |  |
| Abrir(sNm : CtLiteral = Vazio, cGr : PtClsGerenciador = Vazio) : Booleano  |  |
| Salvar(sNm : CtLiteral = Vazio, cGr : PtClsGerenciador = Vazio) : Booleano |  |
| Pasta() : CtString   |  |
| LimparTemporario() : void  |  |
| operator=(cObj : RfCtClsProjAplic) : RfCtClsProjAplic                      |  |
| operator==(cObj : RfCtClsProjAplic) : Booleano                             |  |
| operator!=(cObj : RfCtClsProjAplic) : Booleano                             |  |
| operator>(cObj : RfCtClsProjAplic) : Booleano                              |  |
| operator>=(cObj : RfCtClsProjAplic) : Booleano                             |  |
| operator<(cObj : RfCtClsProjAplic) : Booleano                              |  |
| operator<=(cObj : RfCtClsProjAplic) : Booleano                             |  |
| Instancia() : IntPqn   |  |
| InstanciaDe(lVal : IntPqn) : Booleano                                      |  |

**QUADRO 5 – Classe Projeto**

A Classe Componente(ClsComponente), apresentada em maiores detalhes no Quadro 6, disponibiliza as informações necessárias para inserir um componente no repositório e/ou em um projeto de *software*. De sua totalidade, destaca-se:

- *nome do componente* -> nome que identifica o componente no ambiente da ferramenta;
- *identificador* -> nome utilizado para identificação no código fonte do projeto;
- *categoria* -> classificação do componente por categoria, sendo esta utilizada na lista de categorias do repositório;
- *tipo* -> identifica se o componente é específico do sistema operacional, uma aplicação ou dispositivo de *hardware*;
- *arquivos* -> especificação dos arquivos de código e de cabeçalho;
- *propriedades* -> propriedades do componente;
- *operações* -> especificação das operações (interfaces) de entrada e saída do componente;
- *requisitos* -> especificação dos requisitos necessários para a ligação do componente em um projeto.

| ClsComponente |  |
|---------------|--|
| ✚             | ClsComponente(sAD : CtLiteral = Ltr_Tp(LTR_VAZIO), ITP : ComponenteTipo = Comp_Sistema)  |
| ✚             | ClsComponente(cObj : RfCtClsComponente) : explicit                                       |
| ✚             | Identificador() : CtLiteral  |
| ✚             | Identificador(vVal : CtLiteral) : void   |
| ✚             | Manuseador() : CtLiteral   |
| ✚             | Manuseador(vVal : CtLiteral) : void  |
| ✚             | Categoria() : CtLiteral  |
| ✚             | Categoria(vVal : CtLiteral) : void   |
| ✚             | Tipo() : ComponenteTipo  |
| ✚             | Tipo(vVal : ComponenteTipo) : void   |
| ✚             | ArqDados() : CtLiteral   |
| ✚             | ArqDados(vVal : CtLiteral) : void  |
| ✚             | ArqCodigo() : RfListaString  |
| ✚             | ArqCodigo() : RfCtListaString  |
| ✚             | ArqCodigo(vVal : RfCtListaString) : void   |
| ✚             | ArqCabecalho() : RfListaString   |
| ✚             | ArqCabecalho() : RfCtListaString   |
| ✚             | ArqCabecalho(vVal : RfCtListaString) : void  |
| ✚             | ArqLocal() : RfListaString   |
| ✚             | ArqLocal() : RfCtListaString   |
| ✚             | ArqLocal(vVal : RfCtListaString) : void  |
| ✚             | Imagem() : CtLiteral   |
| ✚             | Imagem(vVal : CtLiteral) : void  |
| ✚             | Imagem(IQual : ComponenteImagem) : RfImagem  |
| ✚             | Imagem(IQual : ComponenteImagem) : RfCtImagem  |
| ✚             | Imagem(IQual : ComponenteImagem, sImg : RfCtImagem) : void                               |
| ✚             | Imagem(sImgG : RfCtImagem, sImgP : RfCtImagem) : void                                    |
| ✚             | LstPropriedades() : RfListaPropriedades  |
| ✚             | LstPropriedades() : RfCtListaPropriedades  |
| ✚             | LstPropriedades(vVal : RfCtListaPropriedades) : void                                     |
| ✚             | LstOperacoes() : RfListaOperacoes  |
| ✚             | LstOperacoes() : RfCtListaOperacoes  |
| ✚             | LstOperacoes(vVal : RfCtListaOperacoes) : void   |
| ✚             | LstTiposDadosUsr() : RfListaTiposDadosUsr  |
| ✚             | LstTiposDadosUsr() : RfCtListaTiposDadosUsr  |
| ✚             | LstTiposDadosUsr(vVal : RfCtListaTiposDadosUsr) : void                                   |
| ✚             | LstEnum Const() : RfListaEnum Const  |
| ✚             | LstEnum Const() : RfCtListaEnum Const  |
| ✚             | LstEnum Const(vVal : RfCtListaEnum Const) : void   |
| ✚             | LstRequisitos() : RfListaRequisitos  |
| ✚             | LstRequisitos() : RfCtListaRequisitos  |
| ✚             | LstRequisitos(vVal : RfCtListaRequisitos) : void   |
| ✚             | Dimencoes() : RfRetangulo  |
| ✚             | Dimencoes() : RfCtRetangulo  |
| ✚             | Dimencoes(vVal : RfCtRetangulo) : void   |
| ✚             | Dimencoes(IEsq : Inteiro, ITopo : Inteiro, ILarg : Inteiro, IAlt : Inteiro) : void       |
| ✚             | Limpar() : void  |
| ✚             | Pasta() : CtString   |
| ✚             | PontoDentro(sPonto : PtCtPonto) : Booleano   |
| ✚             | PontoDentro(IX : IntMed, IY : IntMed) : Booleano   |
| ✚             | PontoDentro(sPonto : PtCtPonto, IDsvX : IntMed, IDsvY : IntMed) : Booleano               |
| ✚             | PontoDentro(IX : IntMed, IY : IntMed, IDsvX : IntMed, IDsvY : IntMed) : Booleano         |
| ✚             | CalcTamanho(cComp : PtClsComponente, IFonte : HFONT) : void                              |
| ✚             | CalcTamanho(cLista : RfCtListaComponentes, IFonte : HFONT, bVer : Booleano = Nao) : void |
| ✚             | operator=(cObj : RfCtClsComponente) : RfCtClsComponente                                  |
| ✚             | operator==(cObj : RfCtClsComponente) : Booleano  |
| ✚             | operator!=(cObj : RfCtClsComponente) : Booleano  |
| ✚             | operator>(cObj : RfCtClsComponente) : Booleano   |
| ✚             | operator>=(cObj : RfCtClsComponente) : Booleano  |
| ✚             | operator<(cObj : RfCtClsComponente) : Booleano   |
| ✚             | operator<=(cObj : RfCtClsComponente) : Booleano  |
| ✚             | Instancia() : IntPqn   |
| ✚             | InstanciaDe(IVal : IntPqn) : Booleano  |

**QUADRO 6 – Classe Componente**



A Classe Gerenciador, ilustrada no Quadro 7, é a implementação do módulo de análise na Arquitetura da Ferramenta apresentada na seção 3.2 deste trabalho. Contém os métodos para manipulação do repositório, a lista de compiladores e os métodos de manipulação do mesmo (Quadro 7). Na manipulação do repositório de componentes destaca-se os métodos:

- *CompiladorCarregar* -> é chamada quando o programa é inicializado para carregar a lista de compiladores cadastrados;
- *CompiladorGravar* -> é chamado quando é efetuada uma alteração na lista de compiladores cadastrados;
- *RepositórioInserir* -> inclusão de componentes no repositório;
- *RepositórioRemover* -> remoção de componentes;
- *RepositórioProcurar* -> busca de componentes;
- *RepositórioCarregar* -> obtenção da lista de arquiteturas e categorias registradas no repositório e carregar um determinado componente em um determinado projeto de *software*.

| CsGerenciador |  |
|---------------|--|
| ✓             | CsGerenciador(sPstRep: CLiteral = Ltr_Tp(LTR_VAZIO), sLstComp: FQListaCompiladores = Vazio)  |
| ✓             | CsGerenciador(cObj: FQCsGerenciador): explicit   |
| ✓             | Repositorio(): CLiteral  |
| ✓             | Repositorio(VAl: CLiteral): void   |
| ✓             | LstCompiladores(): FQListaCompiladores   |
| ✓             | LstCompiladores(): FQListaCompiladores   |
| ✓             | LstCompiladores(VAl: FQListaCompiladores): void  |
| ✓             | Mensagem(): CLiteral   |
| ✓             | PstTemp(): QString   |
| ✓             | Limpar(): void   |
| ✓             | RepositorioInserir(cComp: FQCsComponente, sArq: CLiteral, sArq: CLiteral, d.stArqEx: FQListaString = Vazio): Booleano  |
| ✓             | RepositorioRemover(sArq: CLiteral): Booleano   |
| ✓             | RepositorioProcurar(sNm: CLiteral, sArq: CLiteral, sCat: CLiteral, sArq: PQString = Vazio): Booleano   |
| ✓             | RepositorioCarregar(cComp: FQCsComponente, sArq: CLiteral, sCat: CLiteral, sArq: CLiteral = Vazio, sNm: CLiteral = Vazio, d.stEx: FQListaString = Vazio): Booleano |
| ✓             | Repositorio.sArquiteturas(d.st: FQListaString): Booleano   |
| ✓             | Repositorio.sCategorias(sArq: CLiteral, d.st: FQListaString): Booleano   |
| ✓             | Repositorio.sComponentes(sArq: CLiteral, sCat: CLiteral, d.st: FQListaString): Booleano  |
| ✓             | Repositorio.local(sPst: Literal, sArq: CLiteral, sCat: CLiteral): Literal  |
| ✓             | Repositorio.local(sPst: PQString, sArq: CLiteral, sCat: CLiteral): Booleano  |
| ✓             | CompiladorCarregar(): Booleano   |
| ✓             | CompiladorClavar(): Booleano   |
| ✓             | operator=(cObj: FQCsGerenciador): FQCsGerenciador  |
| ✓             | operator==(cObj: FQCsGerenciador): Booleano  |
| ✓             | operator!=(cObj: FQCsGerenciador): Booleano  |
| ✓             | operator>(cObj: FQCsGerenciador): Booleano   |
| ✓             | operator>=(cObj: FQCsGerenciador): Booleano  |
| ✓             | operator<(cObj: FQCsGerenciador): Booleano   |
| ✓             | operator<=(cObj: FQCsGerenciador): Booleano  |
| ✓             | Instancia(): IntPtr  |
| ✓             | InstanciaDe(VAl: IntPtr): Booleano   |

QUADRO 7 – Classe Gerenciador

A *Classe Compilador (ClsCompilador)*, detalhada no Quadro 8, contém as informações sobre um compilador de uma determinada arquitetura e métodos para chamar o compilador e obter o resultado da compilação. A classe compilador disponibiliza a lista de arquivos necessários à compilação e geração do código executável do *software* em projeto.

| CsCompilador |   |
|--------------|---|
| ✓            | CsCompilador(sNm : CLiteral = Ltr_Tp(LTR_VAZIO), sArqt : CLiteral = Ltr_Tp(LTR_VAZIO), sDsc : CLiteral = Ltr_Tp(LTR_VAZIO), dLstPr : FICListaOmpldProg = Vazio) |
| ✓            | CsCompilador(cObj : FICCsCompilador) : explicit   |
| ✓            | LstProgramas() : FICListaOmpldProg  |
| ✓            | LstProgramas() : FICListaOmpldProg  |
| ✓            | LstProgramas(Val : FICListaOmpldProg) : void  |
| ✓            | Arquitetura() : CLiteral  |
| ✓            | Arquitetura(Val : CLiteral) : void  |
| ✓            | Arquitetura(D) : CLiteral   |
| ✓            | Arquitetura(D)(Val : CLiteral) : void   |
| ✓            | LstBiblioteca() : FICListaString  |
| ✓            | LstBiblioteca() : FICListaString  |
| ✓            | LstBiblioteca(Val : FICListaString) : void  |
| ✓            | LstIndusoes() : FICListaString  |
| ✓            | LstIndusoes() : FICListaString  |
| ✓            | LstIndusoes(Val : FICListaString) : void  |
| ✓            | Mensagem() : CLiteral   |
| ✓            | Limpar() : void   |
| ✓            | Gerar(cProjeto : FICsProjAplic) : Booleano  |
| ✓            | operator=(cObj : FICCsCompilador) : FICCsCompilador   |
| ✓            | operator==(cObj : FICCsCompilador) : Booleano   |
| ✓            | operator!=(cObj : FICCsCompilador) : Booleano   |
| ✓            | operator>(cObj : FICCsCompilador) : Booleano  |
| ✓            | operator<=(cObj : FICCsCompilador) : Booleano   |
| ✓            | operator<(cObj : FICCsCompilador) : Booleano  |
| ✓            | operator>=(cObj : FICCsCompilador) : Booleano   |
| ✓            | Instancia() : IntPqn  |
| ✓            | InstanciaDe(Val : IntPqn) : Booleano  |

QUADRO 8 – Classe Compilador

### 3.6. Conclusão

O presente capítulo apresentou detalhadamente a ferramenta desenvolvida neste trabalho. O capítulo apresentou inicialmente uma visão geral da arquitetura, abordando seguidamente os módulos que compõe a mesma e finalmente algoritmos e diagramas de classes utilizados no projeto.

A ferramenta foi desenvolvida com o objetivo de facilitar o reaproveitamento de componentes de *software* e o desenvolvimento e geração de códigos especificamente para *software* de sistemas embutidos.

A construção deste modelo envolveu a elaboração de uma linguagem para descrever os componentes inseridos no ambiente, de modo a facilitar ao projetista o acesso às operações e requisitos de cada componente. Como consequência o desenvolvimento de um interpretador para a manipulação das informações especificadas conforme a sintaxe da linguagem.

Assim, a ferramenta tem o propósito de minimizar o tempo de desenvolvimento do *software*, aumento da produtividade e facilitar o reaproveitamento de aplicações e códigos de componentes já desenvolvidos.

## **CAPÍTULO 4**

### **ESTUDOS DE CASO**

A ferramenta, conforme já descrita no Capítulo 4, possibilita a configuração direta dos componentes da aplicação e a configuração indireta dos componentes do Sistema Operacional através da implementação de chamadas de sistema. O módulo da ferramenta possibilita ao projetista escolher ou inserir os componentes de *software* configurá-los e, por fim, gerar a aplicação a ser inserida no dispositivo de *hardware*. O processo de desenvolvimento de um projeto de aplicação está ilustrado no diagrama presente na Figura 16.

Para validação da proposta apresentada, bem como das características e versatilidades do ambiente proposto, foram realizados dois estudos de caso, sob a plataforma de *hardware* que está descrita na próxima seção.

O primeiro estudo é um relógio digital com teclas para atualização de hora, minuto e segundo. O segundo é um forno de microondas. No segundo caso incluiu-se componentes específicos para a função de contagem de tempo, sendo dentre estes a aplicação do relógio digital onde pode-se testar a ligação de um componente de

aplicação desenvolvido incluído no repositório e reutilizado em uma aplicação com maior abrangência.

#### 4.1. *Plataforma de hardware*

Para fins de análise da ferramenta, utilizou-se um projeto de *hardware* voltado para aplicações embutidas, foram incluídos neste projeto um Display LCD , um Microcontrolador ATmega103, Memória Interna de 8 Kbytes, Memória RAM de 128 Kbytes, um conjunto de oito lâmpadas e um composto de oito teclas. A estrutura do Projeto do *Hardware* é ilustrada visualizada na Figura 18.

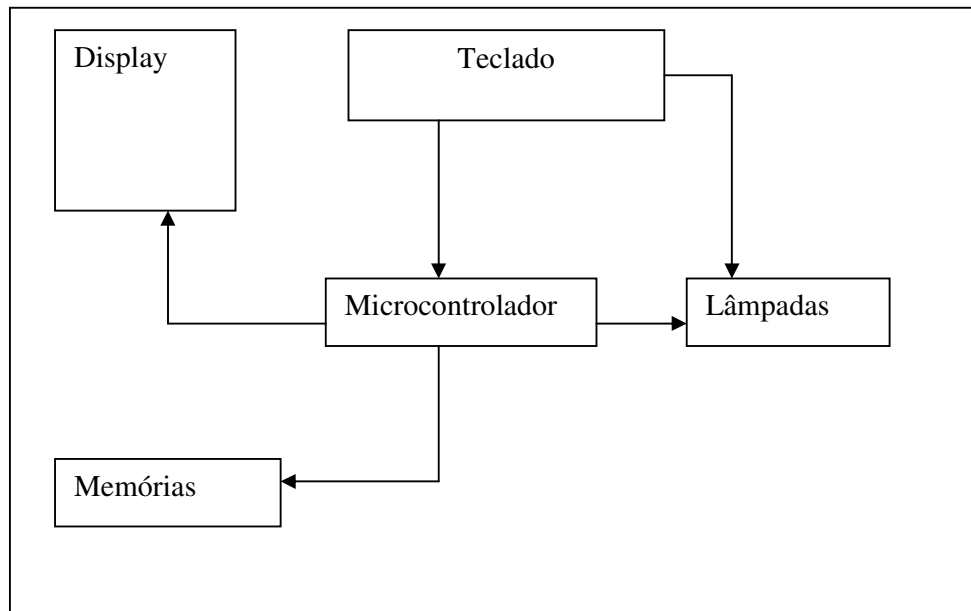


FIGURA 18 – Plataforma de *Hardware*

A implementação dos componentes de *software* referentes a plataforma de hardware descrita neste capítulo não constituem parte integrante do escopo deste trabalho. Desta forma os componentes utilizados nos estudo de casos foram concedidos pelo projetista de aplicações embutidas Alex Sandro Moretti, contribuindo com sua avaliação sob os aspectos práticos de utilização da ferramenta.

#### **4.1.1. Microcontrolador ATmega103**

O microcontrolador ATmega103 apresenta CPU otimizada para aplicações de controle e automação, contendo uma ROM interna (4 Kbytes), RAM interna (128 bytes), comunicação serial full duplex, oscilador interno, 2 temporizadores, 11 instruções do *software*, dentre outras características.

#### **4.1.2. Memórias**

As memórias utilizam a técnica matricial, onde tem-se uma matriz de células, as quais são acessadas por meio de linhas e colunas. Cada célula da matriz pode armazenar um bit. Para construir a matriz é preciso ligar os componentes de forma que possibilite o acesso aos mesmos. Desta forma são necessários dois condutores, um condutor é denominado linha e o outro coluna.

### **4.1.3. Teclado**

Utilizado no projeto de forma a permitir a inserção de dados na aplicação a ser gerada. A coleta de informações através do teclado é um recurso bastante utilizado em diversos equipamentos de sistemas embutidos, como por exemplo, controle remoto, forno de microondas, dentre outros.

### **4.1.4. Display de Cristal Líquido – LCD**

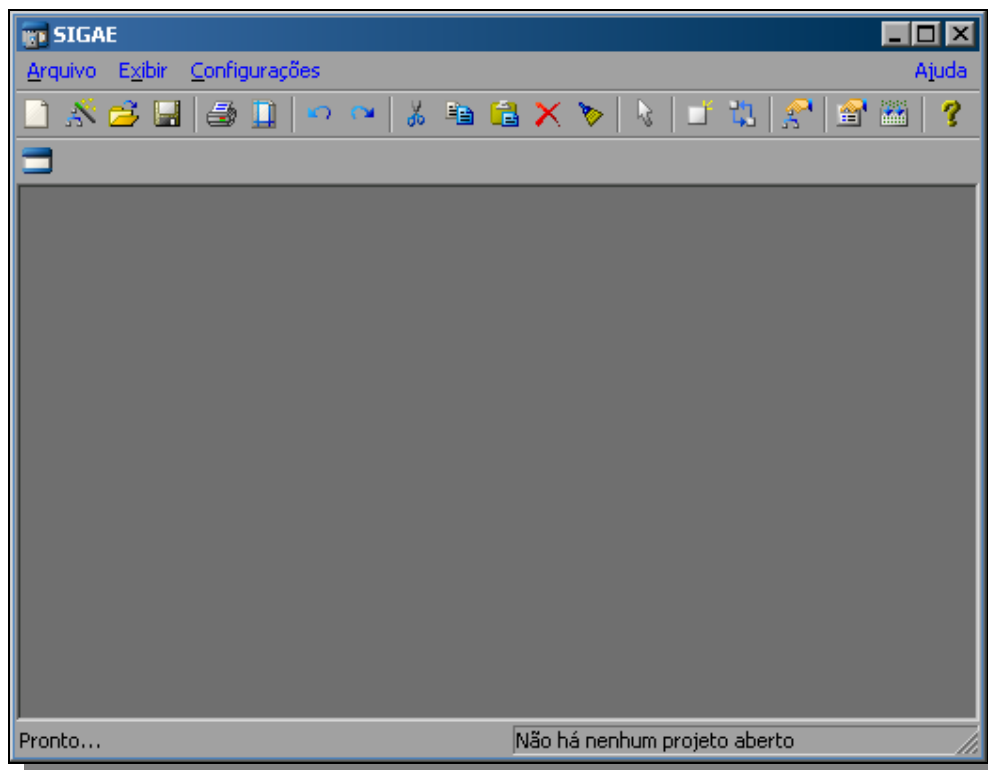
Inserido no projeto do *hardware* embutido de forma a permitir a visualização dos dados enviados ou processados no sistema. O display LCD proporciona uma interface prática entre o usuário da aplicação e o sistema embutido.

## **4.2. Estudo de Caso 1 : Relógio Digital**

O relógio digital desenvolvido, utilizou-se o microcontrolador ATMega103 para fazer a leitura dos botões e atualizar a hora apresentada em um display. Seguindo a metodologia de projeto proposta por Wolf(2001), no que diz respeito ao desenvolvimento do *software* foram incorporados como passos a serem seguidos na utilização da ferramenta de desenvolvimento deste estudo de caso.



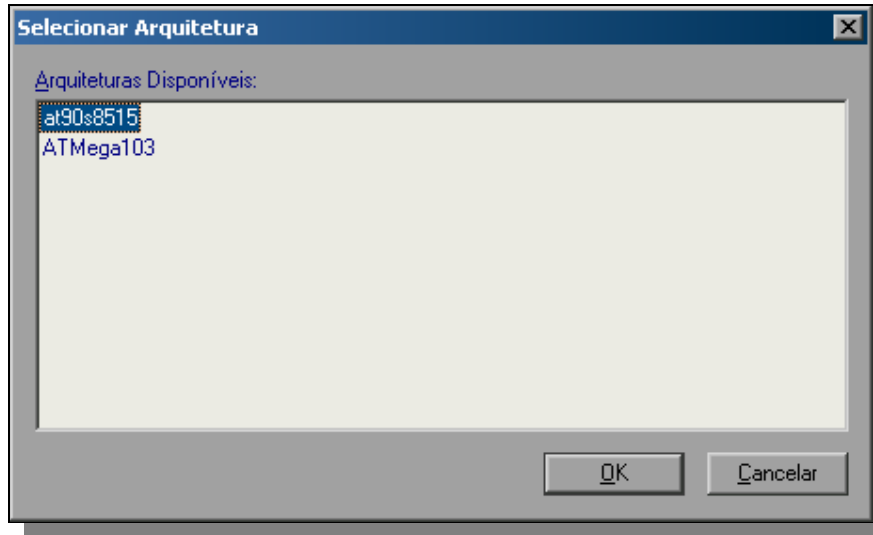
A ferramenta é inicializada com a tela principal mostrada na Figura 19. Esta tela inicial é dividida basicamente em quatro barras: menu, ferramentas e componentes. A barra de menu disponibiliza as funcionalidades que a ferramenta oferece. A barra de ferramentas apresenta as principais e de rápido acesso ao projetista e a barra de componentes fornece acesso aos últimos componentes acessados do repositório e finalmente a barra de status apresenta uma rápida ajuda no processo de desenvolvimento do *software*.



**FIGURA 19 – Janela Inicial da ferramenta**

Inicialmente deve-se criar um novo projeto, para isso usa-se o menu **Arquivo -> Novo**, onde aparecerá uma janela de seleção da arquitetura do processador, ilustrada na

Figura 20. Neste caso optou-se pela arquitetura ATmega103. Pressionado o botão OK, é aberta uma janela para este novo projeto.



**FIGURA 20 – Janela de Arquitetura**

Os requerimentos necessários para o relógio são definidos na documentação da aplicação e definição dos componentes necessários para atender as funcionalidades desejadas pelo relógio.

As propriedades referentes ao projeto são inseridas pelo menu **Projeto -> Propriedades do Projeto**, onde devem ser inseridos o nome do projeto, a pasta (diretório) de destino e o nome do arquivo executável que será gerado pela aplicação. O projeto recebeu a identificação de *clock*.

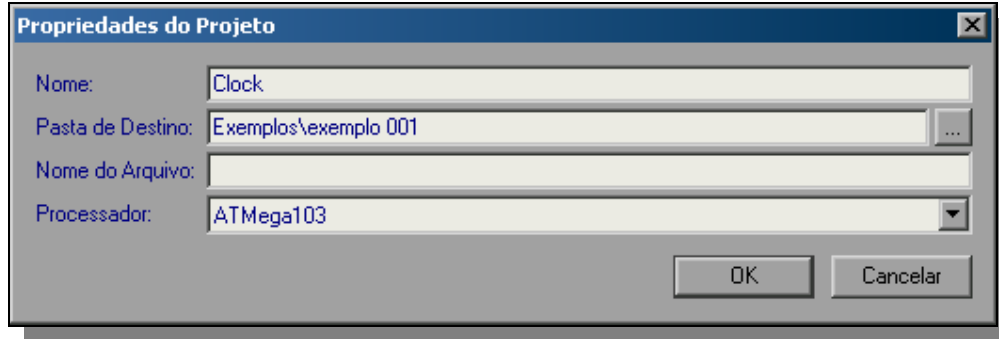


FIGURA 21 – Propriedades do Projeto

Os componentes inseridos no Projeto Relógio foram selecionados através do menu **Projeto -> Inserir Componente**. No qual foram escolhidos os componentes *display*, teclado e *leds*. Ver Fig.22. Os componentes também ficaram disponíveis para seleção na barra de componentes. Assim, nesta etapa de requerimentos da aplicação foi feita a seleção dos componentes existentes no repositório e que foram incorporados ao projeto *clock*.

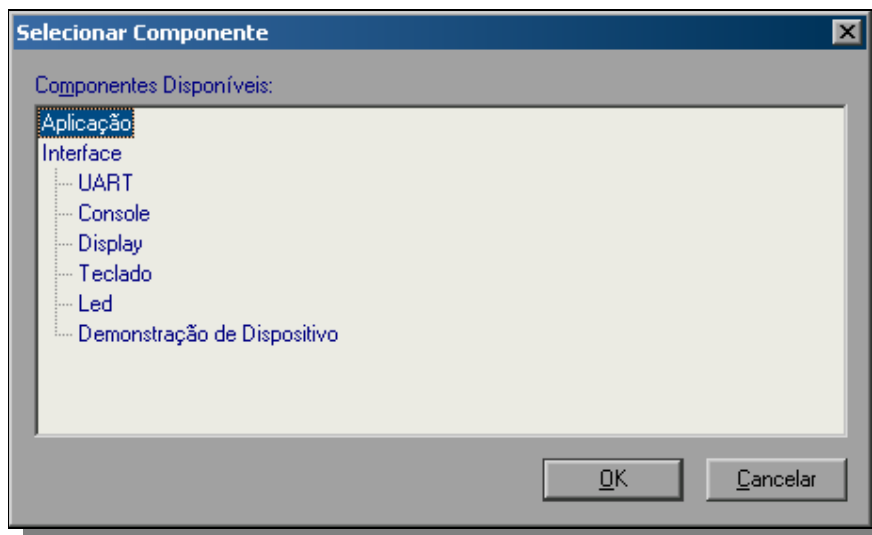


FIGURA 22 – Componentes do Repositório

A próxima etapa de especificação da aplicação é realizada pela ligação e configuração dos componentes para o *software* do *clock*. A ferramenta dispõe de uma interface gráfica de edição das ligações entre os componentes, bastante similar ao modelo de interface gráfica Koala descrito na seção 2.4.1 e Figura 10. As setas indicam o sentido de ligação dos componentes em termos de entrada e saída de dados. (Figura 23)

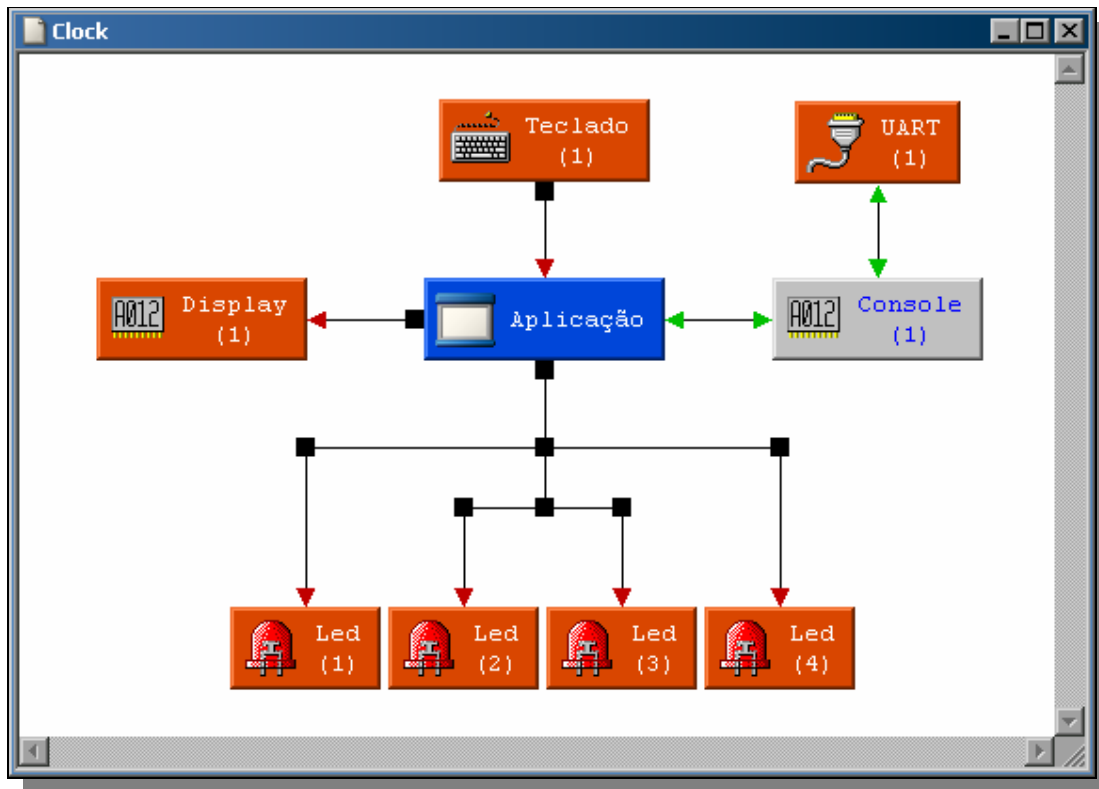


FIGURA 73 – Janela de Edição das ligações dos componentes da aplicação

A ligação dos componentes também é feita em uma segunda etapa através do desenvolvimento da aplicação na janela de edição de aplicação. Nesta janela são inseridas a descrição do projeto e os códigos fontes, conforme ilustrado na Figura 24.

Esta janela disponibiliza a lista de componentes inseridos na etapa anterior com suas interfaces e operações de forma que possam ser utilizadas na ligação propriamente dita dos mesmos. Ainda são inseridas as seguintes informações: nome da aplicação, identificador, imagem, categoria do *software*<sup>14</sup>, imagem representativa da aplicação (opcional) e um texto de descrição do *software*.

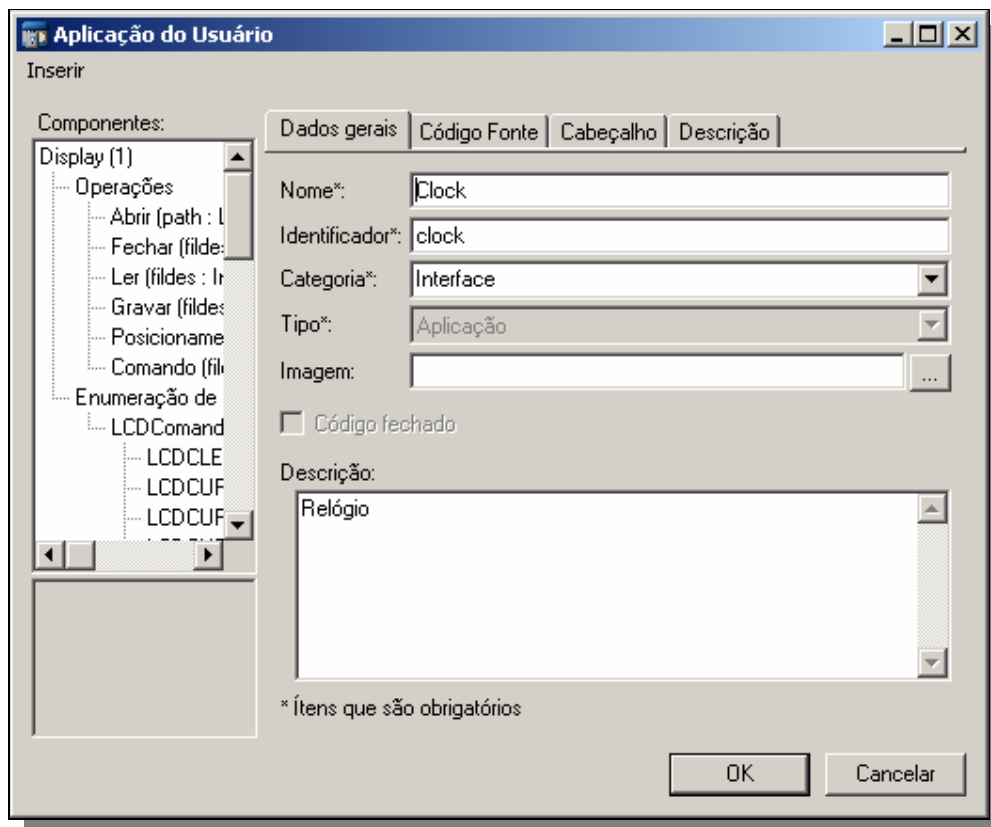


FIGURA 24 – Janela de descrição do projeto de *software*

Na medida em que são selecionadas operações da lista de componentes, automaticamente o código fica inserido na caixa de edição de texto da aplicação.

<sup>14</sup> Categoria utilizada para classificação pelo repositório.

Ao inserir as informações relacionadas ao estudo de caso do *clock*, a descrição do componente *clock* foi gerada automaticamente a partir dos dados indicados. Esta descrição é referente ao arquivo de extensão *dec*, sob a sintaxe da linguagem LDEC elaborada para esta ferramenta. Como trata-se de uma linguagem, a descrição poderá ser feita manualmente obedecendo sua sintaxe e seus comandos. Neste caso a ferramenta faz uma chamada ao interpretador de modo a verificar a descrição está descrita de acordo com a linguagem e desta forma garantir a possibilidade de reutilização desta aplicação.

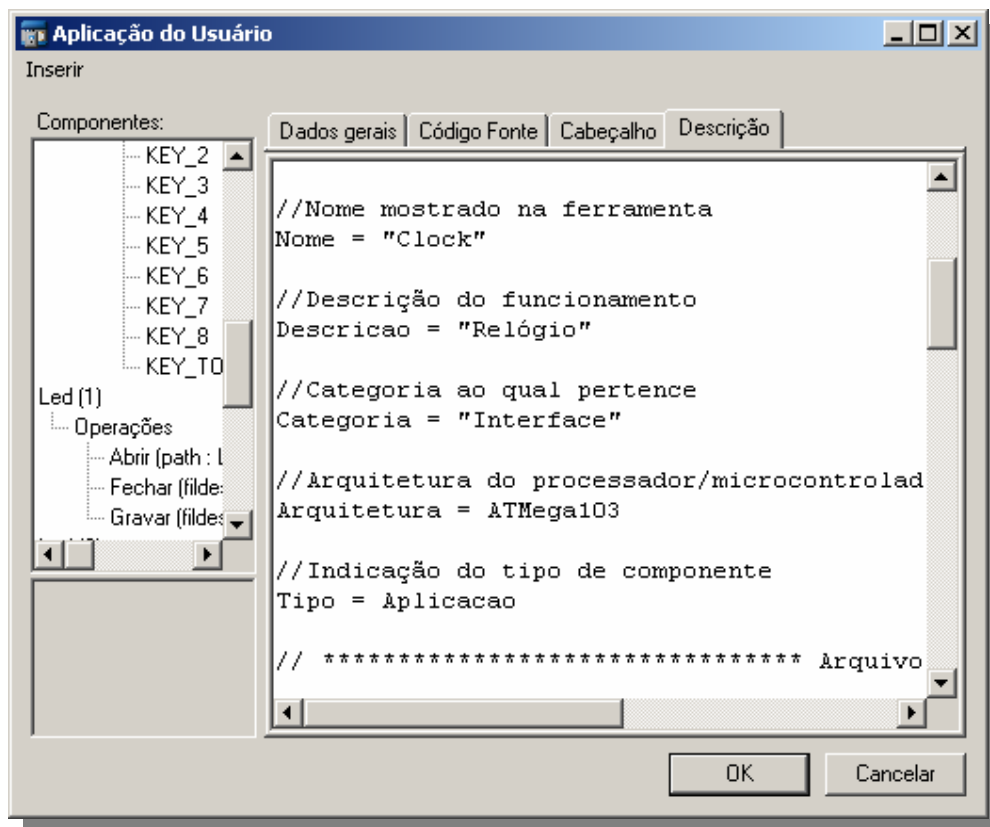


FIGURA 25 – Janela da Linguagem de Descrição de Componente

Visualizando a Figura 26, percebe-se a aba “Código Fonte”, onde ficam inseridos automaticamente os arquivos de cabeçalho de cada componente de software. Uma listagem descritiva das interfaces de cada componente é visualizada na caixa de componentes localizada à esquerda da janela, facilitando a edição do código de ligação dos mesmos. O Quadro 9 mostra integralmente o código gerado pela ferramenta a partir das ligações dos componentes para este estudo de caso, a seção “rotinas” presente no código é inserida manualmente pelo projetista, sendo este o centro de acesso às interfaces de cada componente selecionado para este projeto.

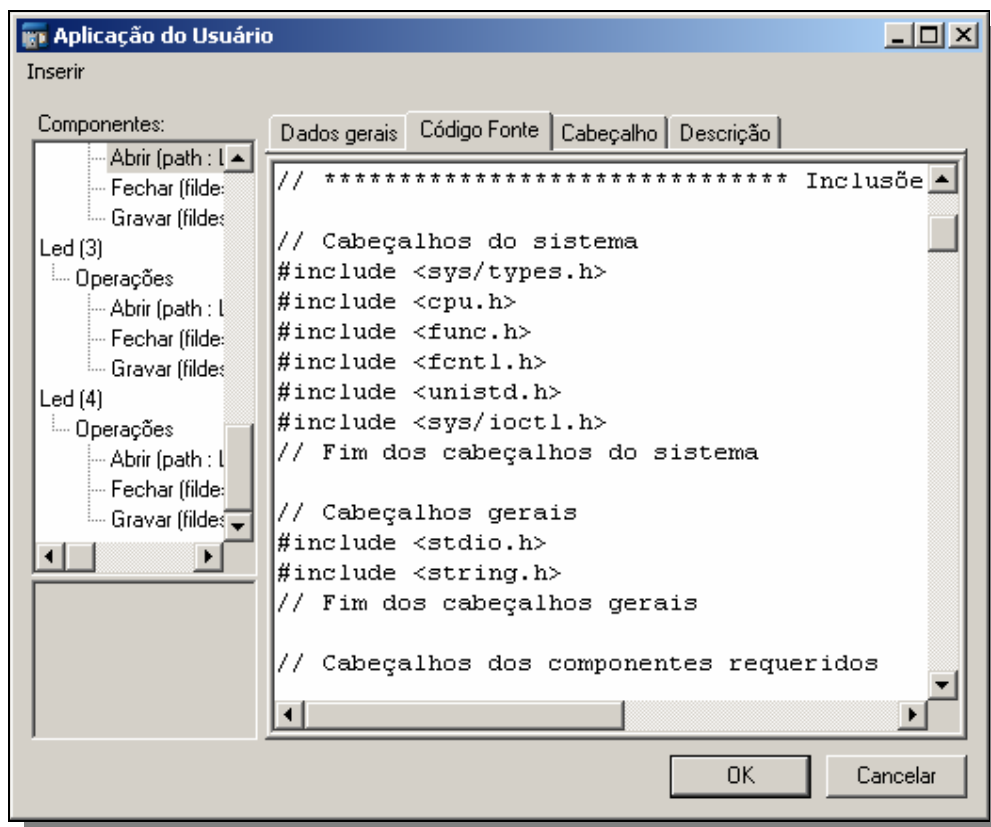
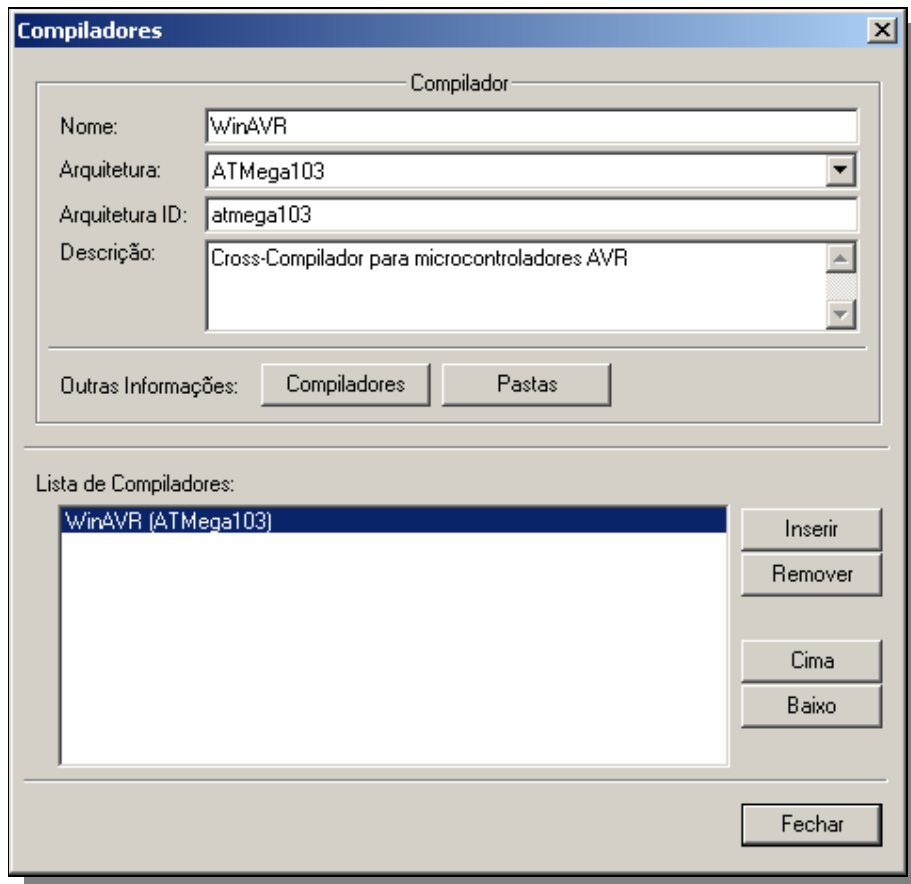


FIGURA 26 – Janela de edição do *software*

Antes de gerar o código do *software*, o compilador referente à arquitetura selecionada para o projeto é especificado e/ou cadastrado na ferramenta. A Figura 27 apresenta a janela onde são requeridas informações de descrição e cadastro do compilador.

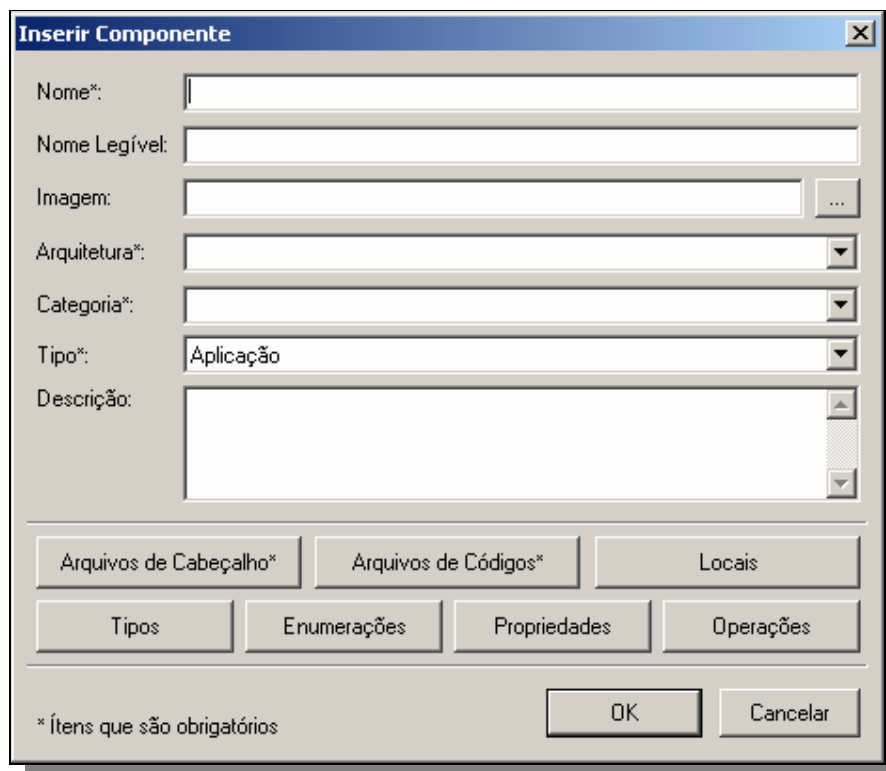


**FIGURA 27 – Janela de especificação do compilador**

Quanto à manipulação do repositório de componentes, esta é acessada através do menu **Configurações -> Repositório**. A partir desta opção pode-se descrever um novo componente e inseri-lo no repositório, importar um componente já descrito ou remover um componente do repositório.



Para a descrição de um componente no repositório, é apresentada uma janela conforme ilustrada na Figura 28. Nesta janela existem campos a serem obrigatoriamente preenchidos referentes a identificação do componente, arquivos de cabeçalho e códigos e diretórios onde a documentação está disponível, os demais campos são referentes as definições já descritas no Capítulo 4.



A janela intitulada "Inserir Componente" possui os seguintes campos e controles:

- Nome\*: Campo de texto obrigatório.
- Nome Legível: Campo de texto.
- Imagem: Campo de texto com botão de navegação de arquivos (...).
- Arquitetura\*: Menu suspenso obrigatório.
- Categoria\*: Menu suspenso obrigatório.
- Tipo\*: Menu suspenso obrigatório, atualmente com "Aplicação" selecionado.
- Descrição: Área de texto grande com barras de rolagem.

Na parte inferior, há uma barra de ferramentas com os seguintes botões:

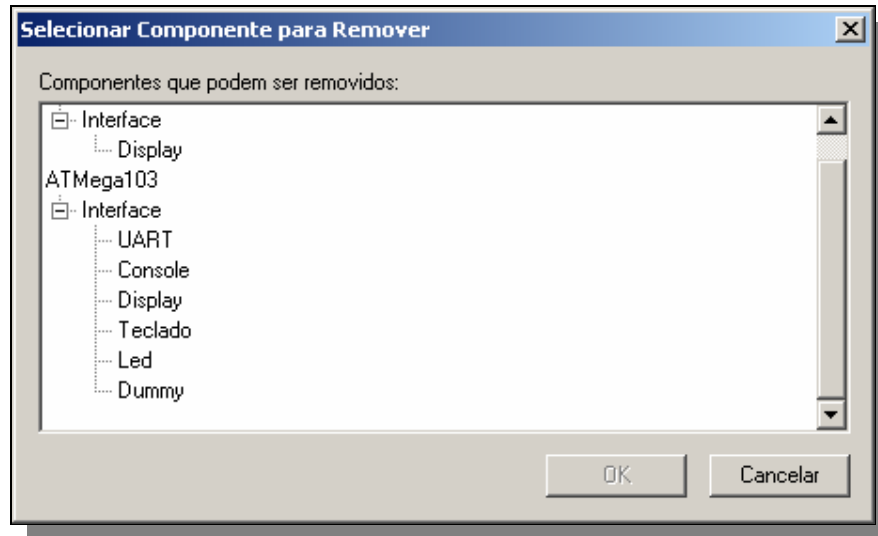
- Arquivos de Cabeçalho\*
- Arquivos de Códigos\*
- Locais
- Tipos
- Enumerações
- Propriedades
- Operações

Na base da janela, há o texto "\* Ítems que são obrigatórios" e os botões "OK" e "Cancelar".

FIGURA 28 – Janela de inserção e descrição do componente para o repositório

Qualquer componente inserido no repositório pode ser removido através do menu **Configurações -> Repositório -> Remover Componentes**. Após será exibida uma janela com a lista de componentes existentes no repositório para seleção. (Figura 29). Neste estudo de caso não foi removido nenhum componente do repositório, entretanto

este ponto foi abordado com o objetivo de descrever a utilização da ferramenta integralmente.



**FIGURA 29 – Removendo um componente do repositório**

Finalmente, o projeto foi gravado em memória secundária acessando o menu **Arquivo -> Salvar** e gerou-se o código do *software* do projeto *clock* através do menu **Projeto->Gerar Código**. Os códigos fontes, de cabeçalho e executáveis ficam disponíveis no diretório especificado na descrição inicial do projeto.

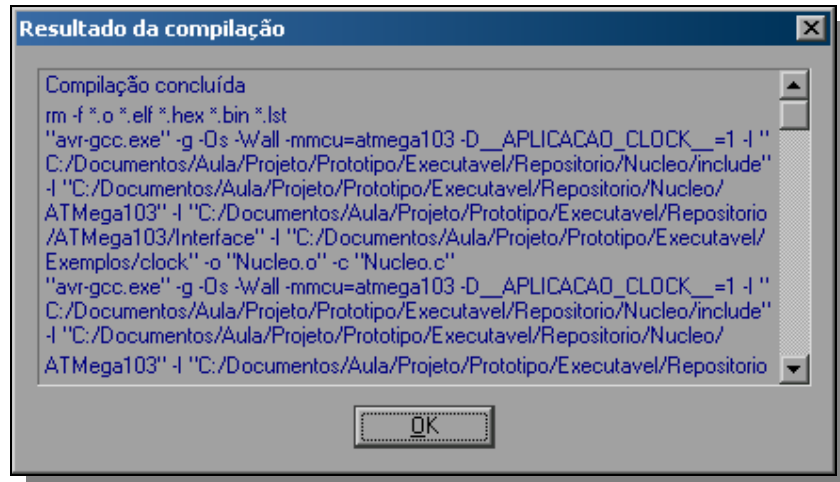


FIGURA 30 – Janela de Compilação

```

////////////////////////////////////
//                               //
//                               //
////////////////////////////////////

/* AVISO IMPORTANTE:
   Não modifique a estrutura básica indicada
   pois a ferramenta a utiliza para ajustar
   os detalhes conforme for editando o projeto
*/

// ***** Inclusões ***** //

// Cabeçalhos do sistema
#include <sys/types.h>
#include <cpu.h>
#include <func.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
// Fim dos cabeçalhos do sistema

// Cabeçalhos gerais
#include <stdio.h>
#include <string.h>
// Fim dos cabeçalhos gerais

// Cabeçalhos dos componentes requeridos
#include <lcd.h>
#include <key.h>
#include <led.h>
// Fim dos cabeçalhos dos componentes requeridos

// Cabeçalhos da aplicação
#include "clock.h"
// Fim dos cabeçalhos da aplicação

// ***** Definições ***** //

// ***** Tipos ***** //

// ***** Estruturas ***** //

// ***** Constantes ***** //

```

```

// ***** Variáveis ***** //

// Manuseadores dos dispositivos
int clock_lcd1;
int clock_key1;
int clock_led1;
int clock_led2;
int clock_led3;
int clock_led4;
// Fim dos manuseadores dos dispositivos

// Variáveis de Tempo
int hora, minuto, segundo;

// Variáveis de controle
int hv, mv, sv;

// ***** Rotinas ***** //

// Mostra a hora no display
void MostrarHora(){
    char buff[12]; int tmn;

    // Verifica se houve alguma alteração
    if ((hora == hv) && (minuto == mv) && (segundo == sv)) return;

    // Mostra a hora
    tmn = sprintf(buff, "%02u:%02u:%02u", hora, minuto, segundo);
    ioctl(clock_lcd1, LCDCLEAR);
    write(clock_lcd1, buff, tmn);

    // Indica a última hora
    hv = hora;
    mv = minuto;
    sv = segundo;
}/* MostrarHora */;

// Muda para a próxima hora
void ProximaHora(){
    hora++;

    if (hora > 24) hora = 0;
}/* ProximaHora */;

// Muda para o próximo minuto
void ProximoMinuto(int bMin){
    minuto++;

    if (minuto > 59){
        minuto = 0;
        if (bMin) ProximaHora();
    };
}/* ProximoMinuto */;

// Muda para o próximo segundo
void ProximoSegundo(int bMin){
    segundo++;

    if (segundo > 59){
        segundo = 0;
        if (bMin) ProximoMinuto(bMin);
    };
}/* ProximoSegundo */;

// Muda para a hora anterior
void HoraAnterior(){
    hora--;
}

```

```

    if (hora < 0) hora = 24;

    /* HoraAnterior */;

    // Muda para o minuto anterior
    void MinutoAnterior(int bMin){
        minuto--;

        if (minuto < 0){
            minuto = 59;
            if (bMin) HoraAnterior();
        };

    }/* MinutoAnterior */;

    // Muda para o segundo anterior
    void SegundoAnterior(int bMin){
        segundo--;

        if (segundo < 0){
            segundo = 59;
            if (bMin) MinutoAnterior(bMin);
        };

    }/* SegundoAnterior */;

    // Pausa o programa por um segundo
    void PausaSegundoComLed(){
        int d, l;

        // Inicia as variáveis
        l = 1;
        d = 0;

        // Pausa e pisca os leds
        for (d = 0x2; d; d--){

            // Pisca os leds
            write(clock_led1, &l, 1);
            write(clock_led2, &l, 1);
            write(clock_led3, &l, 1);
            write(clock_led4, &l, 1);

            // Indica se deve ligar ou desligar os leds
            l = !l;

            // Pausa por meio segundo
            Delay10ms(50);

        };

    }/* PausaSegundoComLed */;

    // Pausa o programa por um segundo
    void PausaSegundo(){
        Delay10ms(100);
    }/* PausaSegundo */;

    // Reinicia o relógio
    void Reiniciar(){
        hora = 0;
        minuto = 0;
        segundo = 0;
        hv = -1;
        mv = -1;
        sv = -1;
    }/* Reiniciar */;

    // Verifica se a hora está zerada
    int Vazio(){
        return (!hora && !minuto && !segundo);
    }/* Vazio */;

```

```

// Inicializa a aplicação
#ifdef __APLICACAO_CLOCK__
int main(){
    int ch;

    // Manuseadores dos dispositivos
    clock_lcd1 = open("lcd", O_WRONLY);
    clock_key1 = open("key", O_RDONLY);
    clock_led1 = open("led", O_WRONLY);
    clock_led2 = open("led", O_WRONLY);
    clock_led3 = open("led", O_WRONLY);
    clock_led4 = open("led", O_WRONLY);
    // Fim dos manuseadores dos dispositivos

    // Inicia a hora
    Reiniciar();

    // Executa a hora
    for (;;) {

        // Mostra a hora na tela
        MostrarHora();

        // Obtém a tecla precionada
        ch = 0;
        read(clock_key1, &ch, 1);

        // Muda para o próximo segundo
        if ((ch & KEY_1) == KEY_1){
            ProximoSegundo(0);
            Delay10ms(10);

        // Muda para o próximo minuto
        } else if ((ch & KEY_2) == KEY_2){
            ProximoMinuto(0);
            Delay10ms(10);

        // Muda para a próxima hora
        } else if ((ch & KEY_3) == KEY_3){
            ProximaHora();
            Delay10ms(10);

        // Muda para o segundo anterior
        } else if ((ch & KEY_4) == KEY_4){
            SegundoAnterior(0);
            Delay10ms(10);

        // Muda para o minuto anterior
        } else if ((ch & KEY_5) == KEY_5){
            MinutoAnterior(0);
            Delay10ms(10);

        // Muda para a hora anterior
        } else if ((ch & KEY_6) == KEY_6){
            HoraAnterior();
            Delay10ms(10);

        // Incrementa a hora
        } else {
            ProximoSegundo(1);
            PausaSegundoComLed();
        };

    };

} // main
#else // __APLICACAO_CLOCK__
void clock_init(){

    // Manuseadores dos dispositivos
    clock_lcd1 = open("lcd", O_WRONLY);

```

```

clock_key1 = open("key", O_RDONLY);
clock_led1 = open("led", O_WRONLY);
clock_led2 = open("led", O_WRONLY);
clock_led3 = open("led", O_WRONLY);
clock_led4 = open("led", O_WRONLY);
// Fim dos manuseadores dos dispositivos

// Inicia a hora
Reiniciar();

} // clock_init
#endif // __APLICACAO_CLOCK__

```

#### QUADRO 9 – Código fonte gerado pela Ferramenta - Estudo de Caso Relógio

O Quadro 10 apresenta o arquivo de descrição de componentes gerado pela ferramenta segundo a linguagem LDEC abordada neste trabalho.

```

////////////////////////////////////
//                               //
////////////////////////////////////
Componente clock
/* AVISO IMPORTANTE:
   Não modifique a estrutura básica indicada
   pois a ferramenta a utiliza para ajustar
   os detalhes conforme for editando o projeto
*/
// ***** Dados Básicos ***** //

//Nome mostrado na ferramenta
Nome = "Clock"
//Descrição do funcionamento
Descricao = "Relógio"
//Categoria ao qual pertence
Categoria = "Interface"
//Arquitetura do processador/microcontrolador
Arquitetura = ATMega103
//Indicação do tipo de componente
Tipo = Aplicacao
// ***** Arquivos ***** //
//Arquivos de código fonte
Codigo = "clock.c"
//Arquivos de cabeçalho
Cabecalho = "clock.h"
// ***** Tipos de Dados do Usuário ***** //
// ***** Enumeração de Constantes ***** //
// ***** Propriedades ***** //
// ***** Operações ***** //
Operacao MostrarHora
Fim Operacao MostrarHora
Operacao ProximaHora
Fim Operacao ProximaHora
Operacao ProximoMinuto
Parametro = Booleano : bMin
Fim Operacao ProximoMinuto
Operacao ProximoSegundo
Parametro = Booleano : bMin

```

```

Fim Operacao ProximoSegundo
Operacao HoraAnterior
Fim Operacao HoraAnterior
Operacao MinutoAnterior
  Parametro = Booleano : bMin
Fim Operacao MinutoAnterior
Operacao SegundoAnterior
  Parametro = Booleano : bMin
Fim Operacao SegundoAnterior
Operacao PausaSegundoComLed
Fim Operacao PausaSegundoComLed
Operacao PausaSegundo
Fim Operacao PausaSegundo
Operacao Reiniciar
Fim Operacao Reiniciar
// ***** Requisitos ***** //
Requisito Componente lcd
Requisito Componente key
Requisito Componente led
Requisito Componente led
Requisito Componente led
Requisito Componente led

```

**QUADRO 10 - Arquivo DEC do projeto Clock**

A fase de Teste de Integração do sistema pode ser feita através do *software* de simulação de *hardware* Proteus, cedido para testes pelo projetista Alex Sandro Moretti. A ferramenta gera um arquivo *makefile* e o respectivo código executável da aplicação que neste estudo de caso, gerando os arquivos *clock.elf*, *clock.hex*, *clock.bin* e arquivos de cabeçalho. Nos testes no ambiente Proteus, inseriu-se o código da aplicação gerada, de modo que pode-se manipular as teclas atualizando a hora visível no display em tempo real. (Figura 31). As legendas da Figura 30 identificam em 01 os leds que piscam conforme o passar dos segundos, 02 nas teclas que são utilizadas para atualizar o relógio e 03 onde exibe-se a hora no display LCD.



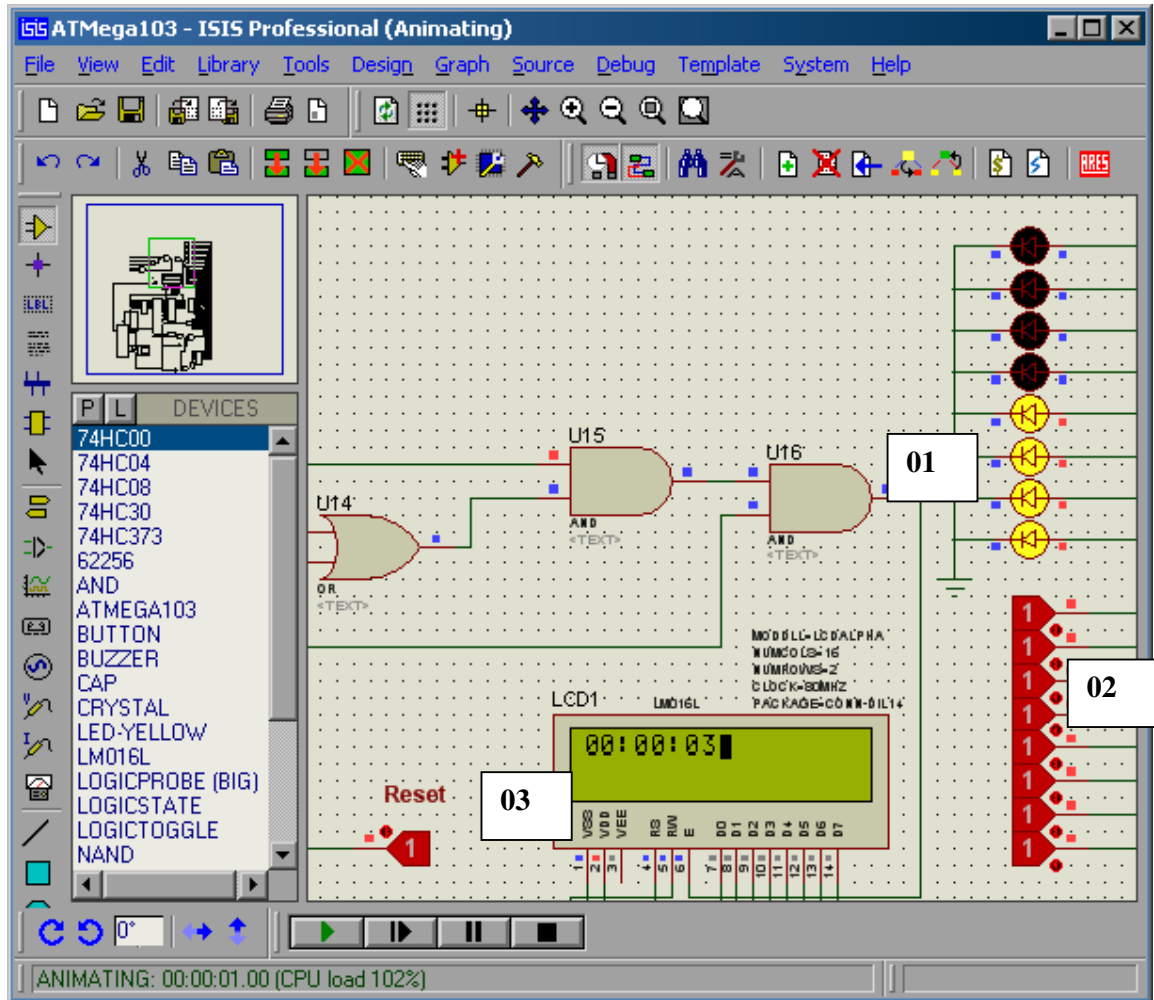


FIGURA 31 – Teste do código - Estudo de Caso Relógio

### 4.3. Estudo de Caso 2 : Forno de Microondas

Para a implementação do Forno de Microondas foi utilizada a mesma seqüência metodológica aplicada no estudo de caso do *clock*. Nos requisitos desta aplicação determinou-se que o forno apresentaria teclas de liga e desliga, acionamento do *timer* para contagem do tempo de descongelamento do alimento e lâmpadas para indicar o término do programa.

Na especificação do modelo, os componentes necessários para a aplicação já estavam inseridos no repositório – *leds*, teclado, *display*, *lcd*. Também foi inserido o componente *clock* desenvolvido no primeiro estudo de caso. (Figura 33)

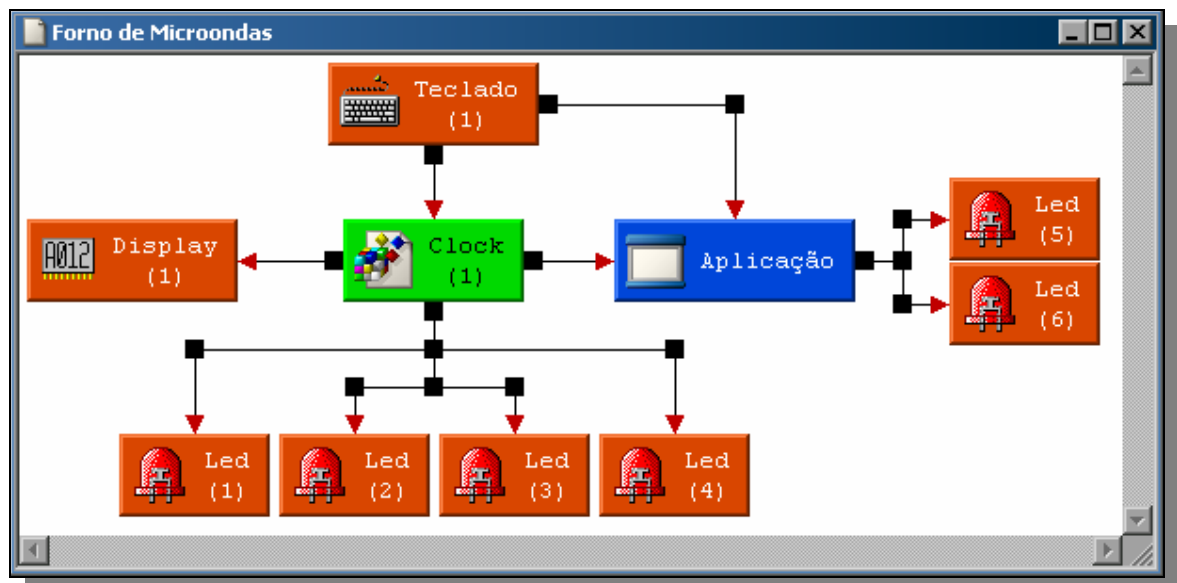


FIGURA 32 – Janela gráfica de visualização dos componentes – estudo de caso 2

Ao configurar a aplicação, todas as interfaces do componente clock estavam disponível de forma que puderam ser utilizadas na ligação com os demais componentes. O clock foi utilizado especificamente no controle do *timer* e exibição do relógio quando não utilizado para fins de aquecimento. A Figura 32 é um fragmento da Janela de configuração da aplicação, onde pode-se visualizar as operações desenvolvidas no *software* clock.

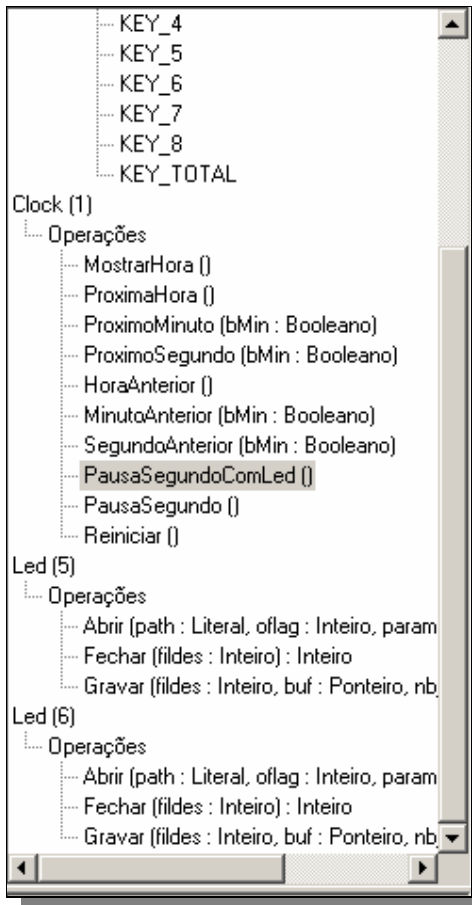


FIGURA 33 – Caixa de Listagem de Operações dos componentes – estudo de caso 2

Da mesma forma ocorrida no primeiro estudo de caso, a ferramenta gerou os arquivos *makefile* para o projeto microondas, onde fez-se os testes da aplicação em

tempo real. O Quadro 11 apresenta a listagem completa gerada pela ferramenta para o projeto microondas.

Em testes no simulador de *hardware* (Figura 34), os componentes de *hardware* funcionaram perfeitamente, onde ficaram disponíveis duas teclas liga e desliga (legendas 1 e 2) e as demais para ajustes de hora, minuto e segundo (3) para descongelamento. Os *leds* foram utilizados em substituição ao apito e na legenda 4 exibe-se a hora e o timer no *display* LCD.

```
#####
#           Forno de Microondas           #
#####

# Makefile gerado pelo SIGAE

# ***** Informações de compilação ***** #

# Arquitetura do projeto
MCU = atmega103

# Pasta do repositório
REP = C:/SIGAE/Executavel/Repositorio/microondas

# Pasta dos compiladores
CMPPST =

# Compiladores
CMP1 = $(CMPPST)avr-gcc.exe
LNG1 = $(CMPPST)avr-gcc.exe
CNVCP1 = $(CMPPST)avr-objcopy.exe
CNVCP2 = $(CMPPST)avr-objcopy.exe
CNVDB3 = $(CMPPST)avr-objdump.exe

# Informações do compiladores
CFLAGS = -g -Os -Wall -mmcu=$(MCU) -D__APLICACAO_MICRO_=1 -I "$(REP)Nucleo/include" -I
"$(REP)Nucleo/ATMega103" -I "$(REP)ATMega103/Interface" -I " C:/SIGAE/Executavel/Repositorio/microondas"

# Informações dos ligadores
LDFLAGS = -mmcu=$(MCU) -lm
```

```

# ***** Compilação dos arquivos ***** #

all:
    "$(CMP1)" $(CFLAGS) -o "Nucleo.o" -c "Nucleo.c"
    "$(CMP1)" $(CFLAGS) -o "lcd.o" -c "$(REP)ATMega103/Interface/lcd.c"
    "$(CMP1)" $(CFLAGS) -o "key.o" -c "$(REP)ATMega103/Interface/key.c"
    "$(CMP1)" $(CFLAGS) -o "led.o" -c "$(REP)ATMega103/Interface/led.c"
    "$(CMP1)" $(CFLAGS) -o "clock.o" -c "$(REP)ATMega103/Interface/clock.c"
    "$(CMP1)" $(CFLAGS) -o "micro.o" -c "micro.c"
    "$(CMP1)" $(CFLAGS) -o "__libc.o" -c "$(REP)Nucleo/lib/__libc.c"
    "$(CMP1)" $(CFLAGS) -o "device.o" -c "$(REP)Nucleo/lib/device.c"
    "$(CMP1)" $(CFLAGS) -o "errno.o" -c "$(REP)Nucleo/lib/errno.c"
    "$(CMP1)" $(CFLAGS) -o "fs.o" -c "$(REP)Nucleo/lib/fs.c"
    "$(CMP1)" $(CFLAGS) -o "init.o" -c "$(REP)Nucleo/lib/init.c"
    "$(CMP1)" $(CFLAGS) -o "inode.o" -c "$(REP)Nucleo/lib/inode.c"
    "$(CMP1)" $(CFLAGS) -o "func.o" -c "$(REP)Nucleo/lib/func.c"
    "$(CMP1)" $(CFLAGS) -o "entry.o" -c "$(REP)Nucleo/ATMega103/entry.S"
    "$(CMP1)" $(CFLAGS) -o "setup.o" -c "$(REP)Nucleo/ATMega103/setup.S"
    "$(LNG1)" $(LD_FLAGS) -o "micro.elf" "Nucleo.o" "lcd.o" "key.o" "led.o" "clock.o" "micro.o" "__libc.o" "device.o"
    "errno.o" "fs.o" "init.o" "inode.o" "func.o" "entry.o" "setup.o"
    "$(CNVCP1)" -O ihex -R .eeprom "micro.elf" "micro.hex"
    "$(CNVCP2)" -O binary -R .eeprom "micro.elf" "micro.bin"
    "$(CNVDB3)" -D "micro.elf" > "micro.lst"

# ***** Limpeza dos arquivos ***** #

clean:
    rm -f *.o *.elf *.hex *.bin *.lst

```

**QUADRO 11 – Arquivo *makefile* – estudo de caso 2**

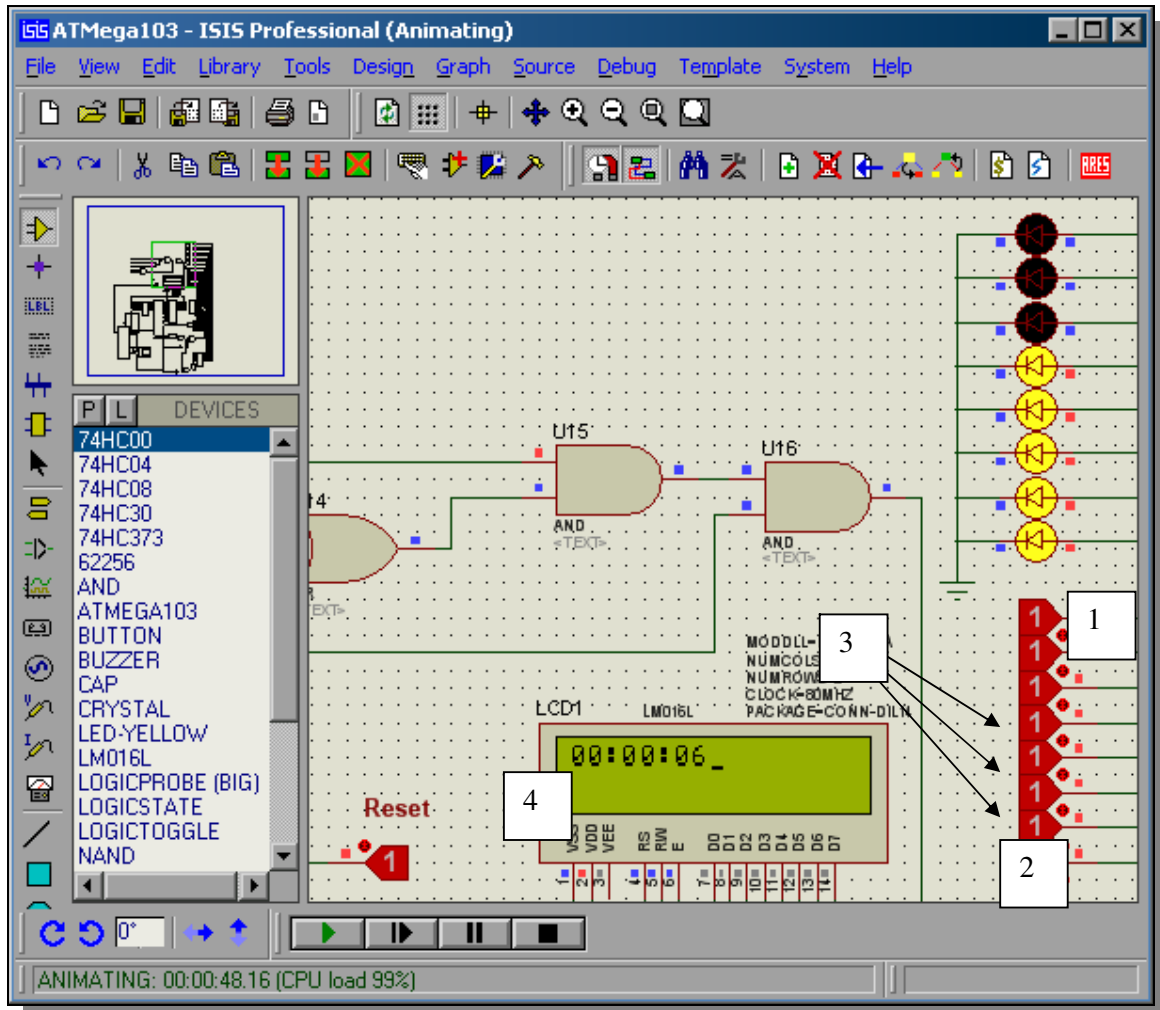


Figura 84 – Teste do código - estudo de caso forno microondas

#### **4.4. Análise dos resultados**

Através deste estudo teórico e desenvolvimento prático, além de entrevistas efetuadas com projetistas de aplicações embutidas, conclui-se que a ferramenta desenvolvida neste trabalho vem a simplificar, desde a documentação destes tipos de projetos, que muitas vezes ocorrem de forma bastante informal quanto na facilidade de reutilização de componentes e aplicações já desenvolvidas.

Nos estudos de caso deste capítulo, escolheu-se justamente dois exemplos onde pudéssemos reutilizar uma aplicação já desenvolvida. Neste caso a aplicação **clock** foi inserida no repositório de componentes e novamente reutilizada em uma nova aplicação, como um *microondas*.

Para análise dos resultados dos estudos de caso apresentados neste capítulo foram utilizadas duas referências básicas: a facilidade no desenvolvimento do *software* e a capacidade de reutilização de um *software* já desenvolvido e armazenado no ambiente repositório. No desenvolvimento dos *softwares* pôde ser constatado que em ambos estudos de caso, a ferramenta forneceu facilidades de acesso aos componentes disponíveis no repositório, sua ligação na interface gráfica e inclusão no código de ligação da aplicação. Outro ponto a considerar é a simplicidade vinculada à utilização da ferramenta, além da flexibilidade de se utilizar uma linguagem que pode ser gerada pelo próprio ambiente.

Quanto à reutilização de componentes e aplicações, na prática, ficou bastante restrito o número de componentes disponibilizados na biblioteca (repositório) do ambiente. Um exemplo, no estudo de caso *microondas*, não havia disponível no

repositório um componente próprio de acionamento das funções como aquecimento ou descongelamento dos alimentos.

Os componentes classificados como dispositivos, que atuam no acionamento físico dos dispositivos de *hardware* ficam necessariamente vinculados a sua respectiva arquitetura. Isto porque fica praticamente impossível construir um componente da categoria “dispositivo” que seja adaptável a qualquer plataforma de *hardware/software*. Entretanto, os componentes classificados “aplicação” e “sistema” ficam abstraídos da camada física e podem ser reutilizados em outras arquiteturas.

Analisando o desempenho, a ferramenta além de possibilitar a visualização gráfica da especificação do modelo de um determinado *software*, possibilita a descrição (documentação) rápida através de interface gráfica gerando todos os arquivos necessários para o perfeito funcionamento do *software* e sua inclusão no repositório. Em testes realizados pelo citado projetista, o código de ligação dos componentes em sua versão executável foi simulado em tempo real na ferramenta Proteus, fornecendo os resultados esperados em cada estudo de caso.

A ferramenta, mesmo possibilitando a descrição dos componentes através da linguagem implementada no modelo proposto, ainda são necessárias algumas melhorias neste aspecto. Estas melhorias referem-se à mudanças na forma de descrição gráfica dos componentes, melhorando a forma de descrição dos diagramas. O ideal seria que houvesse uma ferramenta auxiliar interligada a modelagem do componente, no diagrama gráfico da ferramenta. No estágio atual, para descrever uma interface é necessário relacionar os componentes através de setas gráficas indicando a direção de ligação das interfaces e através da inclusão dos métodos no código fonte da aplicação. Por outro lado, a ligação no código fonte é facilitada pela visualização das interfaces de



cada componente na janela de edição do *software*, não exigindo do projetista o conhecimento prévio dos nomes corretos dos eventos e operações disponíveis em cada componente inserido em sua especificação. Além disso, esta ferramenta possibilita que as operações e as interfaces dos componentes sejam incluídas automaticamente no código de ligação e todas estas informações ficam armazenadas no modelo no arquivo de descrição da aplicação.

## **CAPÍTULO 5**

### **CONCLUSÕES E TRABALHOS FUTUROS**

O desenvolvimento de *softwares* para aplicações embutidas exige do projetista metodologias de desenvolvimento compatíveis com a complexidade e características intrínsecas nos sistemas embutidos. A crescente evolução dos microprocessadores e microcontroladores, juntamente com a necessidade de aplicações mais complexas e com muitos recursos, torna inviável iniciar o desenvolvimento destas aplicações a partir do zero.

Traçando uma comparação entre os computadores de propósito geral e as plataformas de *hardware* para sistemas embutidos, os primeiros estão cada vez mais poderosos e com aplicações complexas em diversas modalidades. Para desenvolver essas aplicações com clareza, de forma coesa e funcional, existe um conjunto de ferramentas de apoio e de desenvolvimento, tais como: sistemas operacionais, compiladores, editores, simuladores entre outras. E não poderia ser diferente para os *softwares* destinados aos sistemas embutidos, já que atuam no mesmo nível de abstração. Em exemplos de aplicações embutidas, cita-se : celulares, GPS, televisões

com Internet, videogames, calculadoras e até mesmo pequenos dispositivos de dentro de computadores como, por exemplo, controlador de discos e placas de vídeo.

Na corrida dos avanços tecnológicos, fica evidente a necessidade de uma ferramenta flexível e que proporcione ao projetista a facilidade em expandir, modificar ou atualizar componentes e/ou aplicações já desenvolvidas.

Para desenvolver a arquitetura da ferramenta proposta nesta dissertação, foram realizados estudos sobre algumas já existentes, tais como Koala (2000) e eCos (Massa, 2003), onde percebeu-se tanto características comuns quanto pontos que deixavam a desejar umas em relação às outras.

Quanto às características comuns, nota-se a utilização de uma linguagem de descrição dos componentes e a organização de uma biblioteca ou repositório. Em Koala (Ommering et al, 2000), a interface gráfica de ligação dos componentes facilita a visualização do modelo de *software* e as interfaces entre seus componentes.

Um outro ponto de estudo deste trabalho foi direcionado às características das linguagens que poderiam ser utilizadas como descrição dos componentes na ferramenta. A opção pela originalidade e flexibilidade quanto à linguagem e sua utilização tanto para a descrição do componente quanto de suas interfaces, levou a construção de uma linguagem própria da ferramenta.

Como resultado, construiu-se uma ferramenta, onde o núcleo do sistema operacional fica inserido no repositório como um componente de *software*, possibilitando assim a inclusão de novas funcionalidades. Um ambiente de alto desempenho, onde através de um interpretador tem-se acesso às interfaces dos

componentes em tempo de projeto, concomitantemente à geração de seu respectivo arquivo de descrição.

O último aspecto de relevância deste trabalho refere-se às características de reutilização tanto de componentes simples quanto de componentes de aplicações. Neste sentido, a definição da forma como um componente é inserido no repositório e a estrutura básica de ligação entre os componentes, possibilitou o fácil desenvolvimento dos estudos de caso apresentados no Capítulo 4.

Como vantagens da utilização da ferramenta verificou-se que:

- facilita o rápido desenvolvimento de um *software* para aplicações embutidas através da reutilização de componentes;
- permite a geração automática do arquivo de descrição de componentes dentro da sintaxe especificada pela linguagem LDEC do ambiente;
- proporciona uma visão geral do sistema, facilitando as futuras alterações;
- proporciona a geração do código executável do *software*, com tamanho de código variando conforme a necessidade da aplicação.

Além das vantagens acima citadas, a ferramenta representa ao projetista de *software* para sistemas embutidos, uma redução no tempo de desenvolvimento de projetos, conseqüentemente, redução no tempo que o produto leva para chegar ao mercado e o custo final deste *software*.

Para finalizar, sugere-se como extensão deste trabalho um estudo sobre formas de incorporação de ferramentas de modelagem de componentes, fortalecimento da

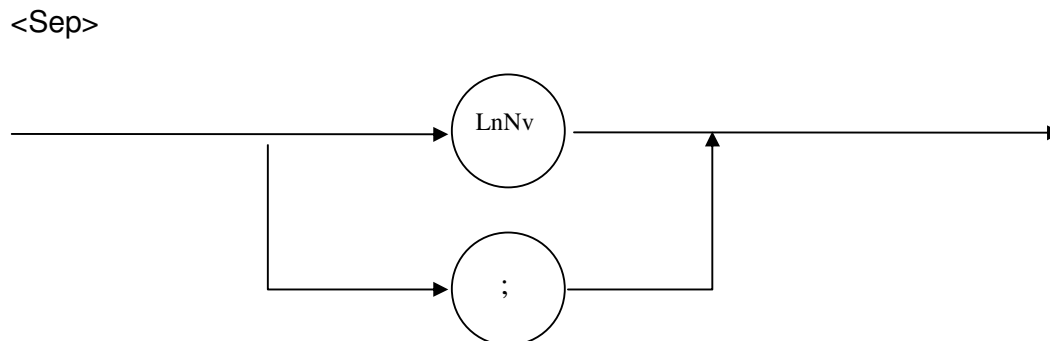
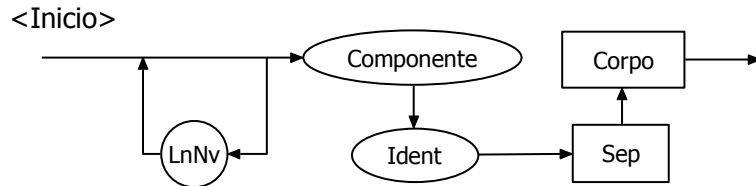
biblioteca de componentes, principalmente no que diz respeito aos componentes do sistema operacional.

Uma segunda sugestão é a expansão do ambiente construído, disponibilizando novas plataformas de *hardware/software*, novos componentes para reutilização e dar continuidade a novos estudos de caso baseados nesta ferramenta, melhorando o módulo de busca dos componentes na biblioteca de forma a otimizar os critérios de escolha de cada componente de acordo com os requisitos de uma determinada aplicação.

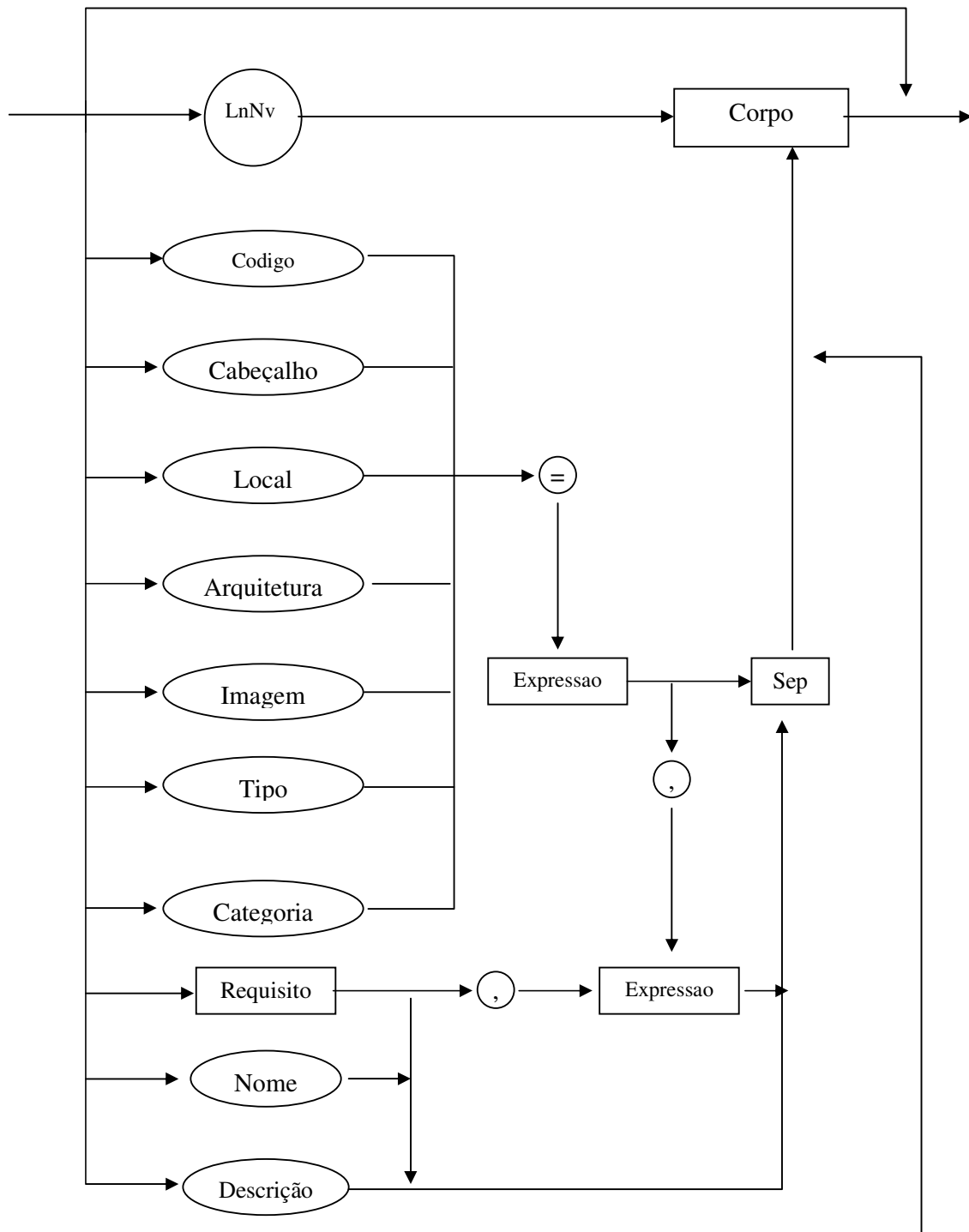
## ANEXO 1

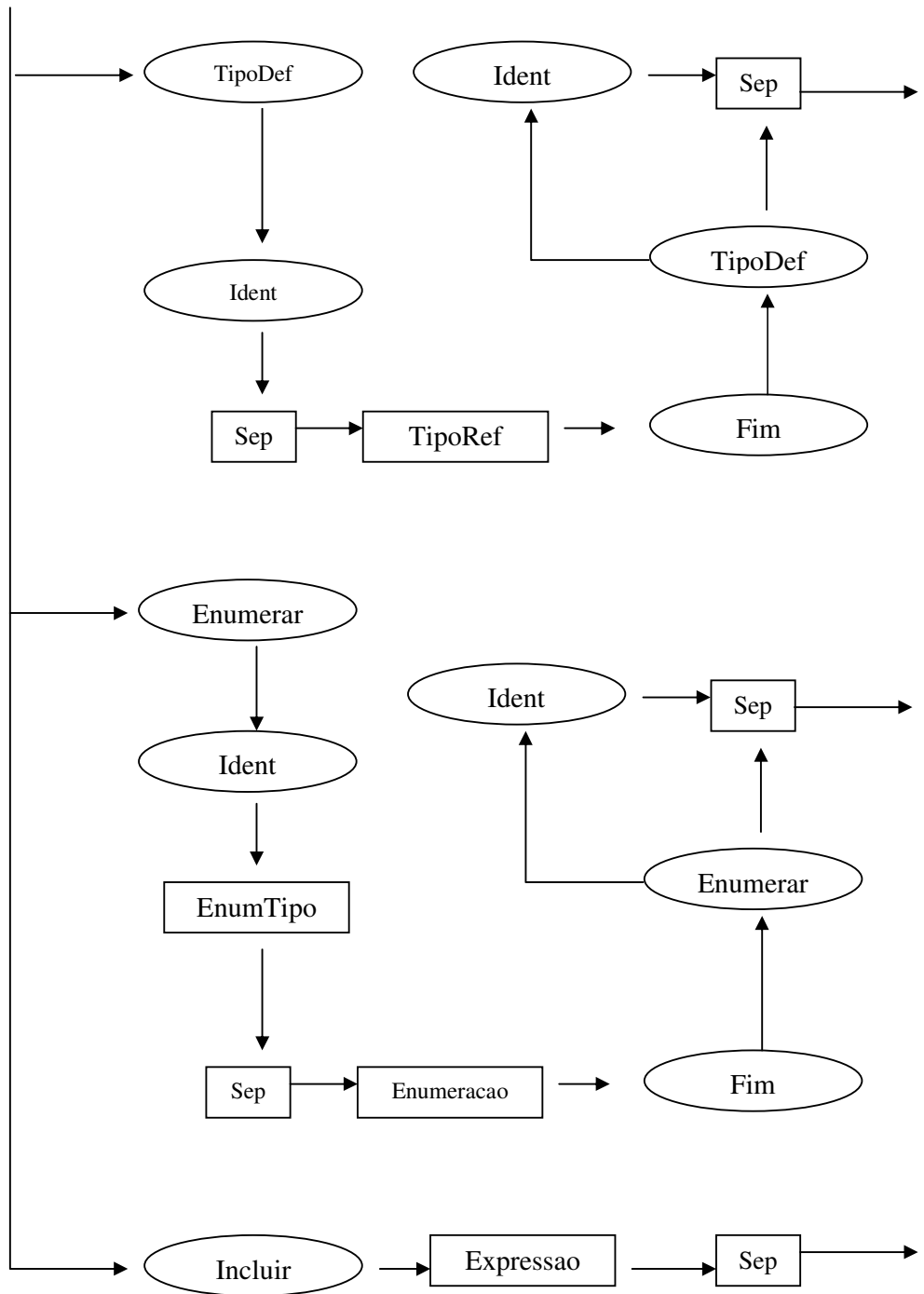
### DIAGRAMAS SINTÁTICOS DO INTERPRETADOR

Este anexo apresenta os diagramas sintáticos utilizados no interpretador da linguagem de descrição de componentes da Ferramenta Sigae. Basicamente, o diagrama sintático é composto por terminais e não terminais. Os terminais são as elipses e os não terminais são os retângulos.



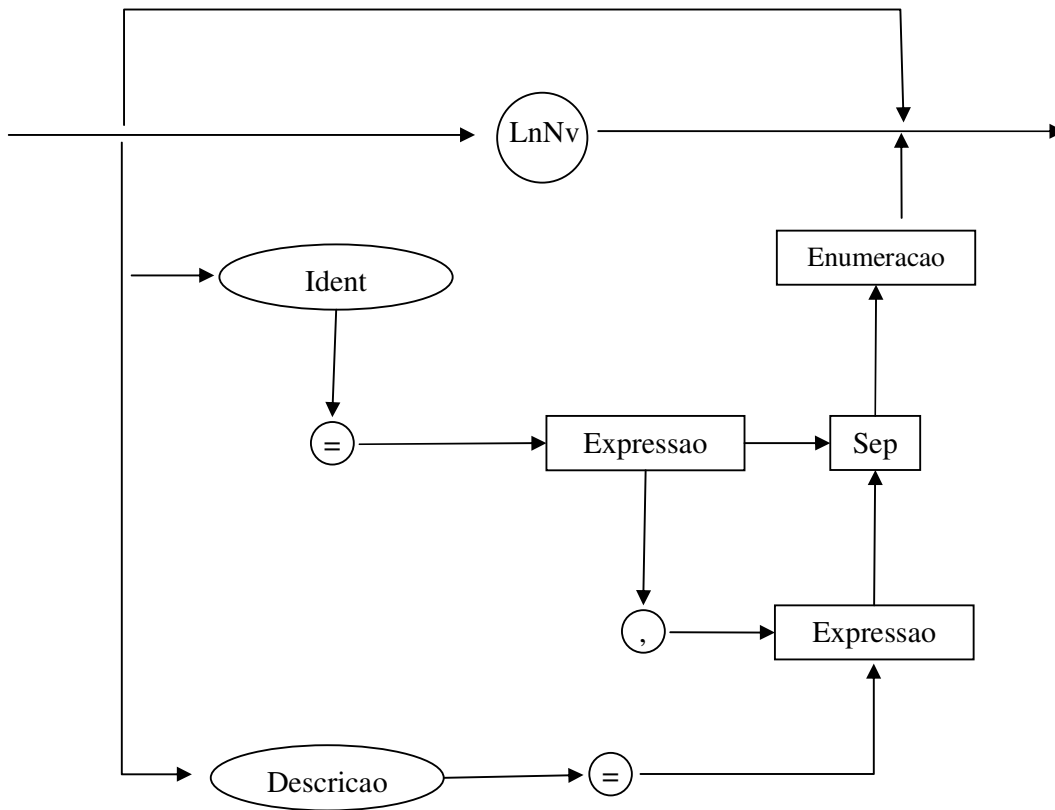
&lt;Corpo&gt;



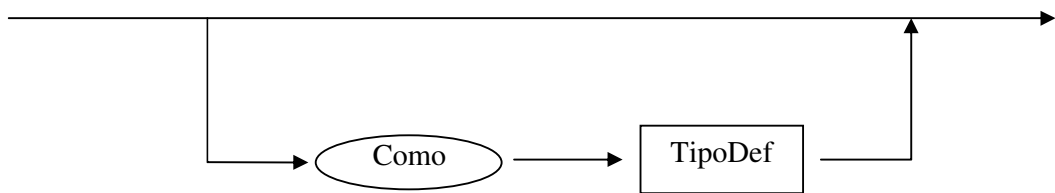




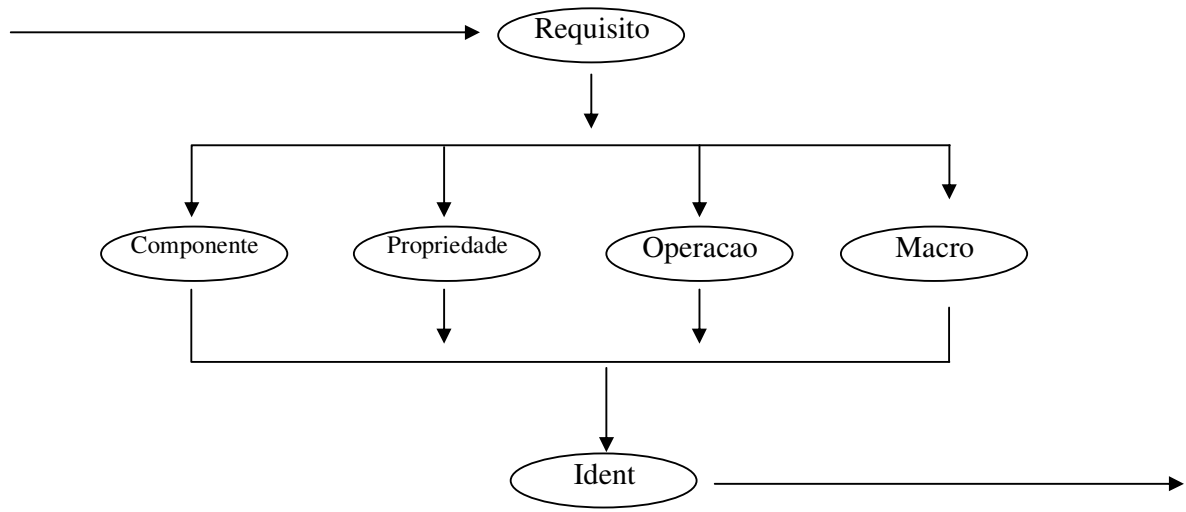
&lt;Enumeracao&gt;



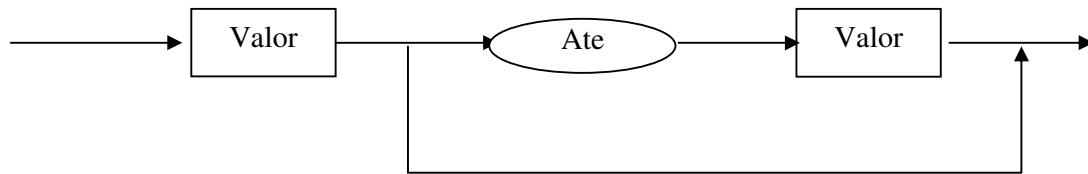
&lt;EnumTipo&gt;



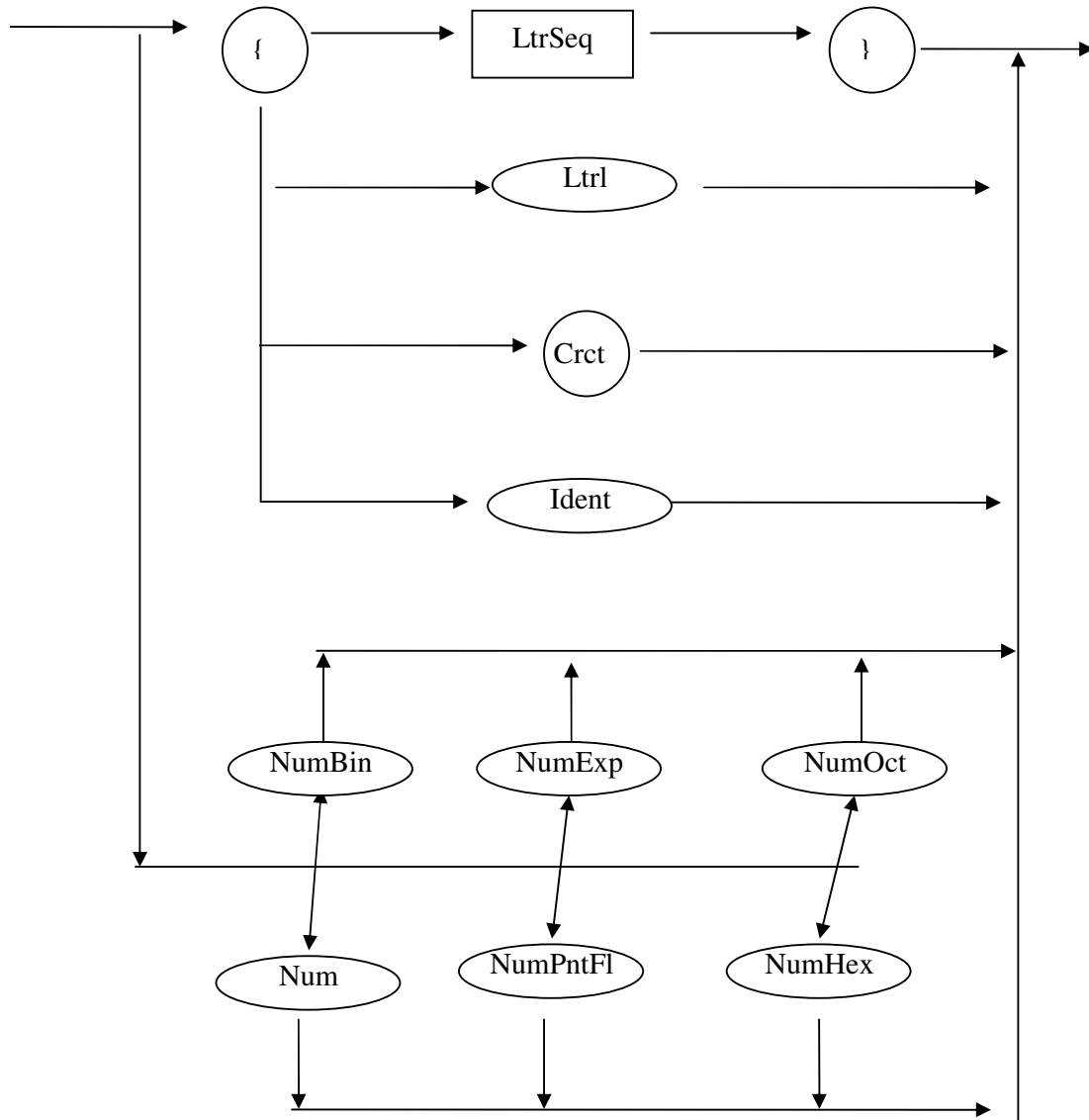
&lt;Requisito&gt;



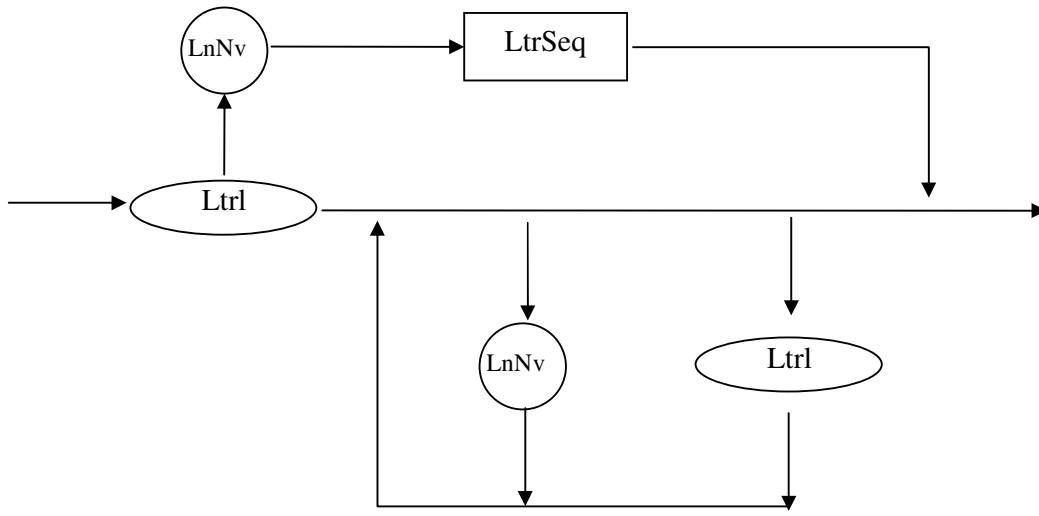
&lt;Expressão&gt;



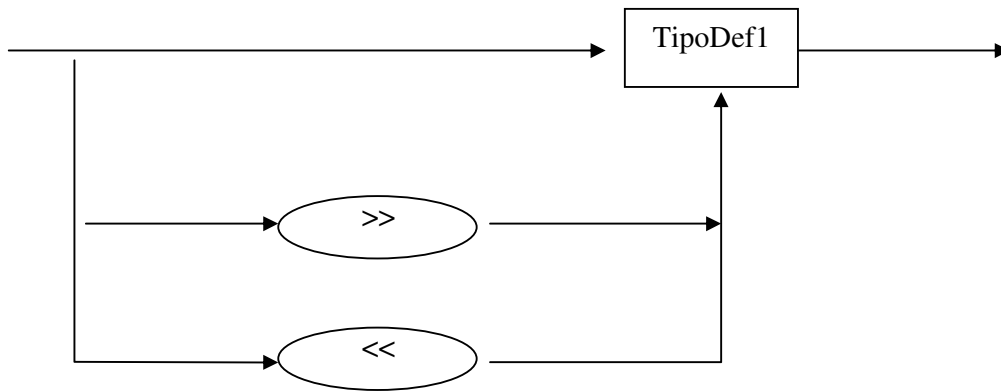
&lt;Valor&gt;



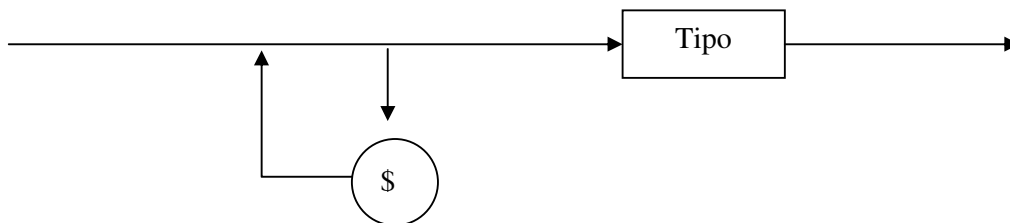
&lt;LtrSeq&gt;



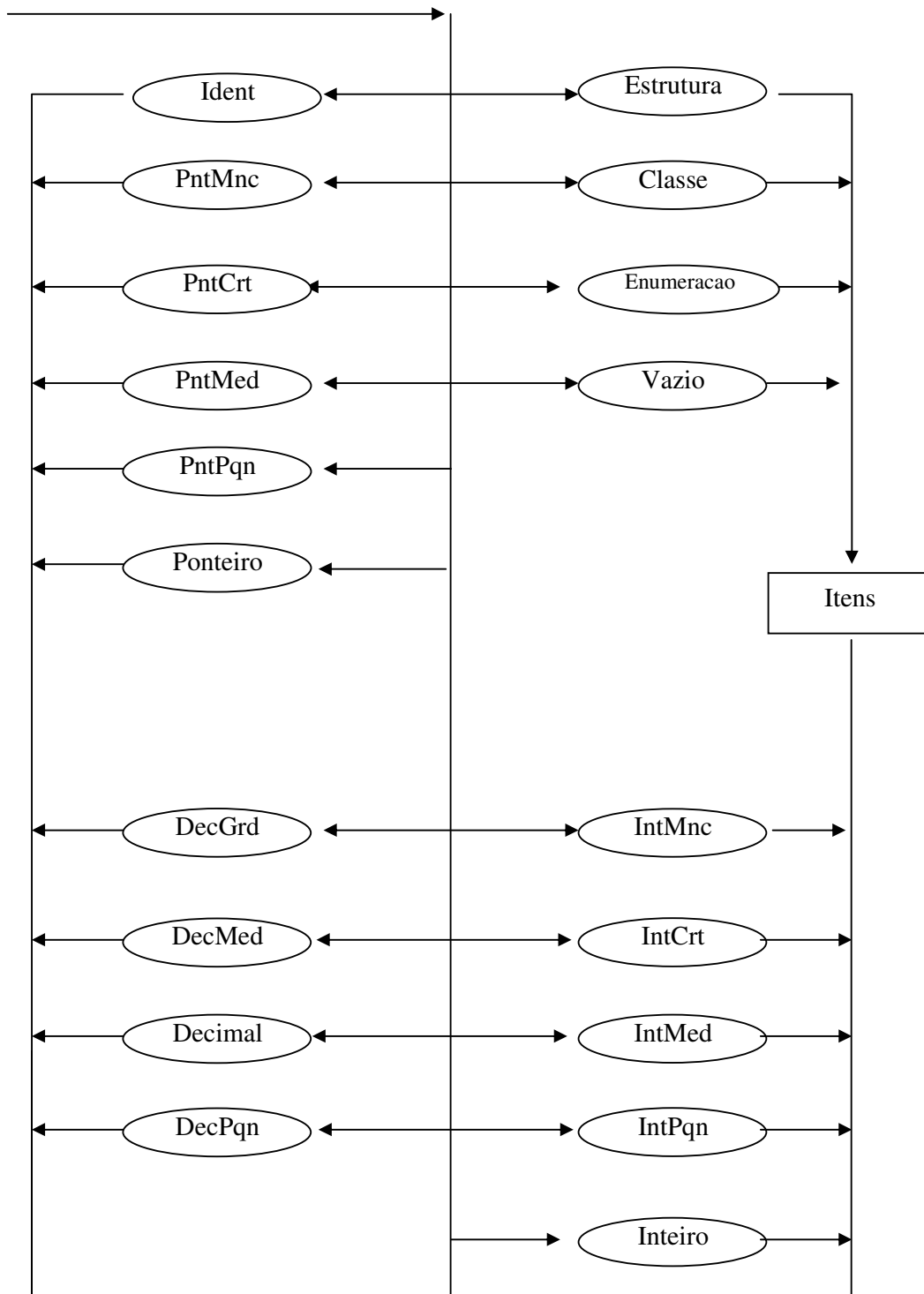
&lt;TipoDef&gt;

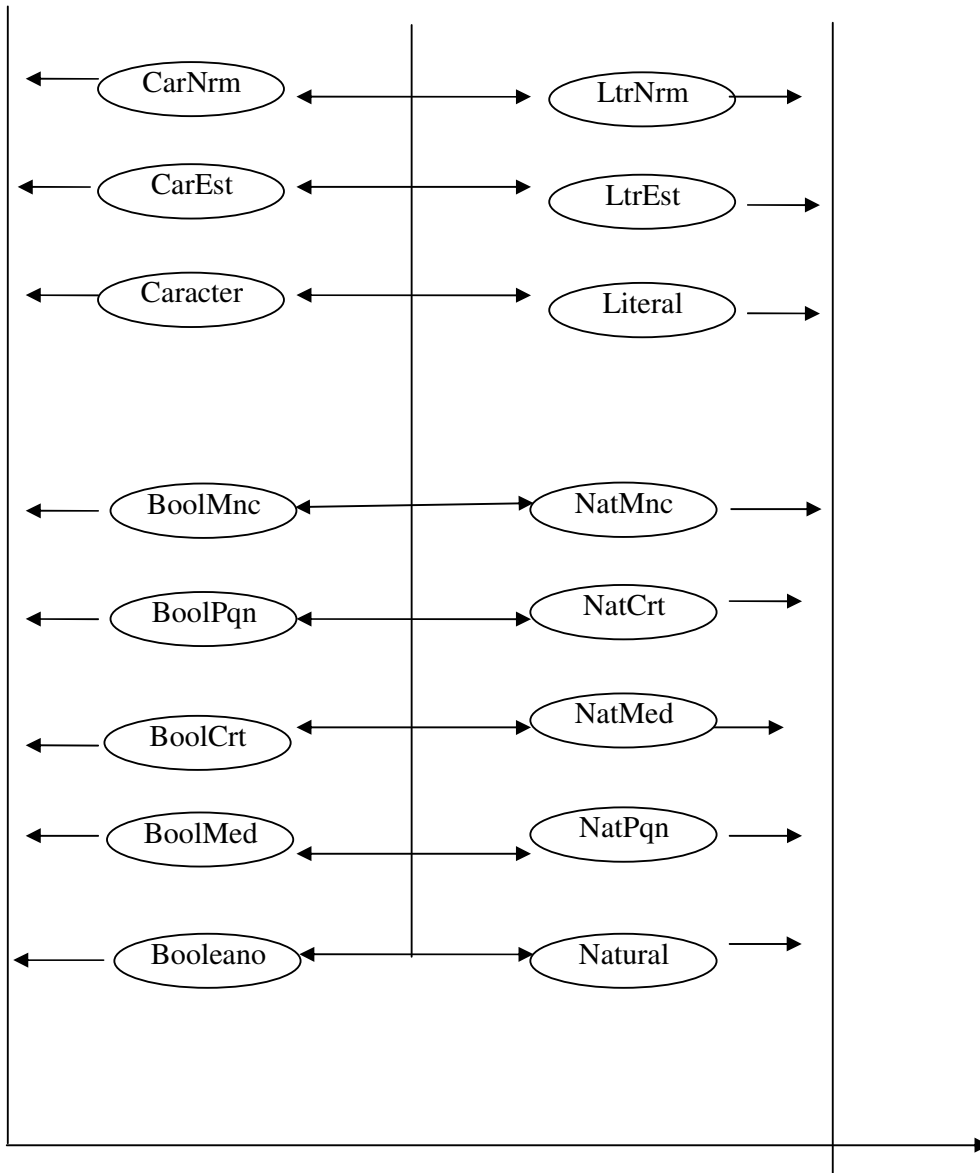


&lt;TipoDef1&gt;

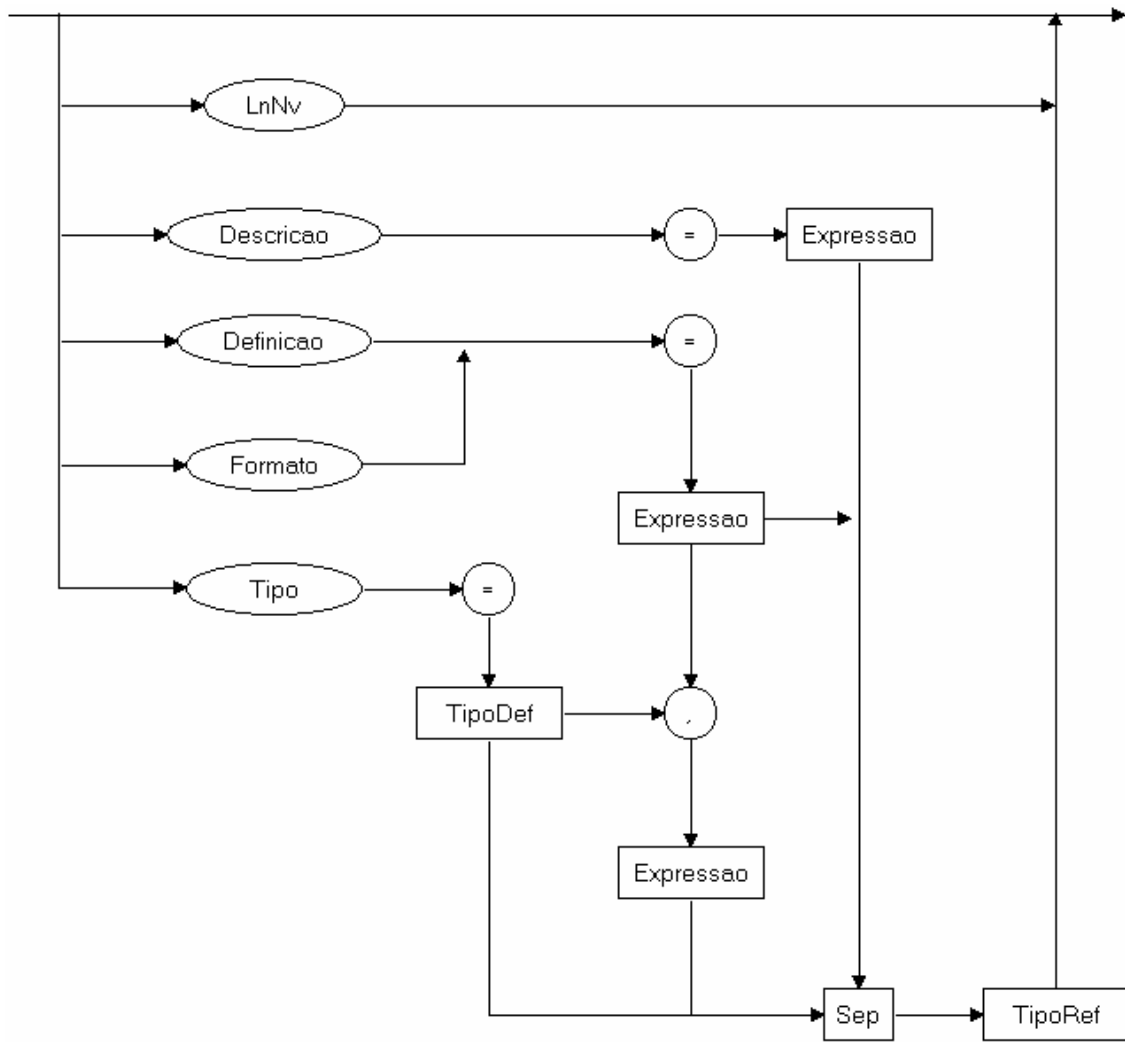


&lt;Tipo&gt;

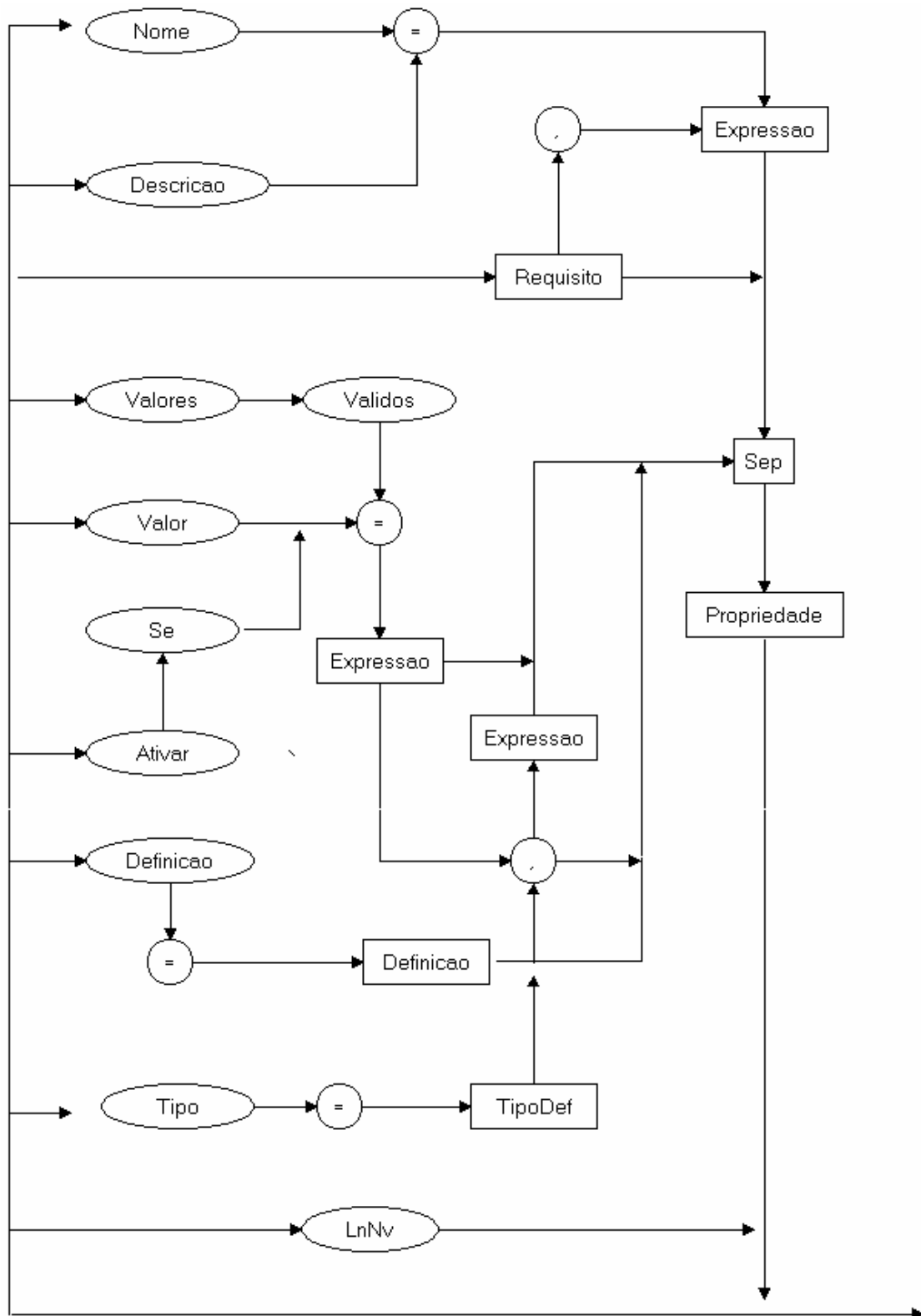




&lt;TipoRef&gt;

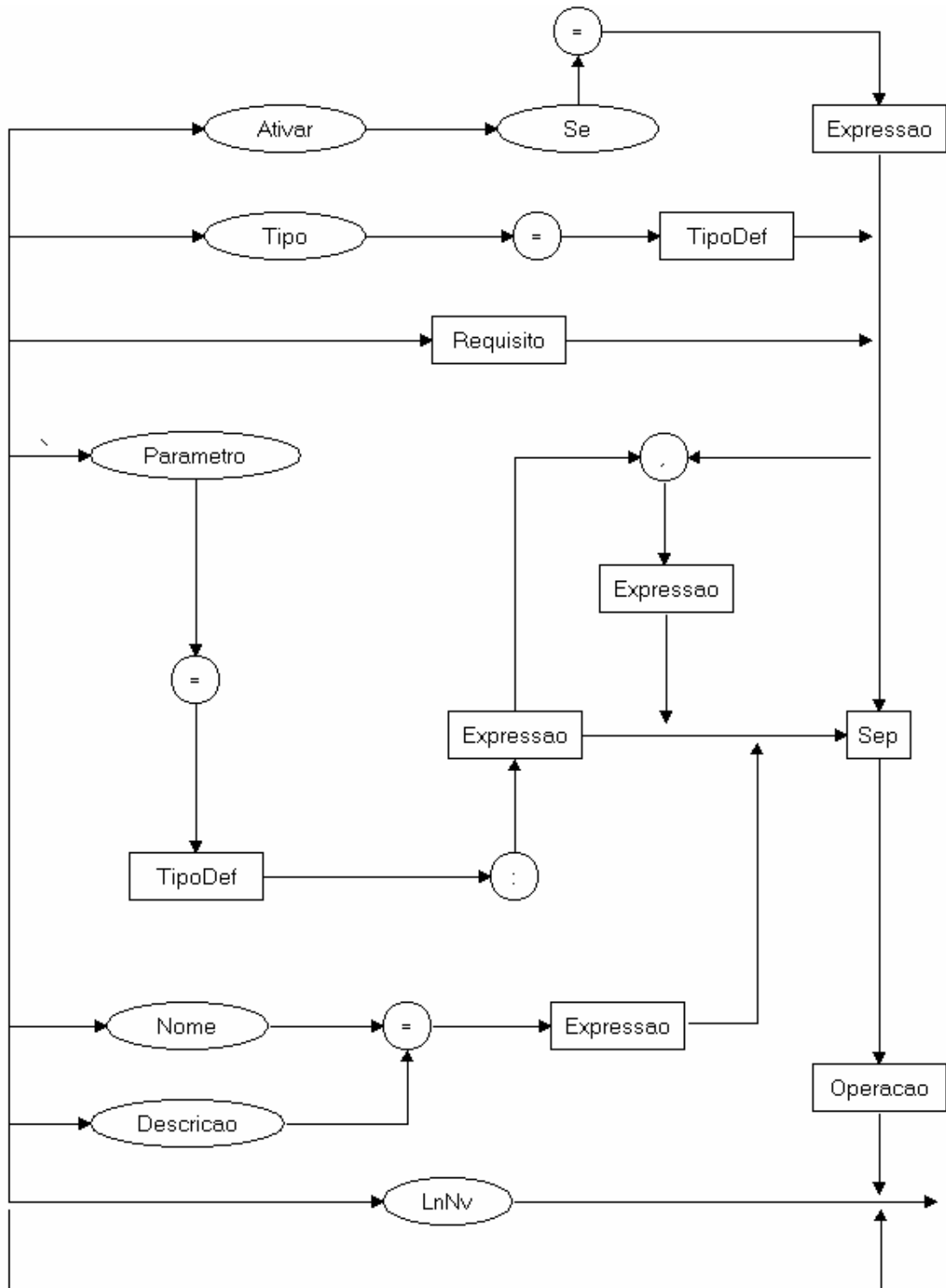


&lt;Propriedade&gt;





&lt;Operacao&gt;



## ANEXO 2

### DESCRIÇÃO DAS AÇÕES SEMÂNTICAS DO INTERPRETADOR

Conforme já comentado no Capítulo 3 desta dissertação, a medida em que são manipuladas informações sobre os componentes, especificamente dentro da linguagem de descrição de componentes LDEC, são disparadas ações semânticas de verificação da consistência das informações.

---

#### *Ações de inicialização da análise:*

Ação 1: Início da análise : verifica se foi indicado o destino dos dados, limpa as variáveis de controle, limpa o destino dos dados, (coloca o identificador do componente no destino e coloca o nome do arquivo de descrição no destino)

Ação 2: Final da análise: Verifica se foram indicados o identificador, o nome do arquivo e a categoria, Verifica se foi indicada a arquitetura, Verifica se foi indicado pelo menos um arquivo de código e de cabeçalho, Verifica se os tipos de dados do usuário utilizados em uma operação, propriedade, enumeração e outras definições de tipos do usuário. Verifica se foi indicado o nome do componente e, caso não se tenha indicado, utiliza o identificador como nome.

---

#### **Ações para obter os valores:**

Ação 3: Informação atual, Guarda em uma variável de controle a última informação pedida.

Ação 4: Tipo de requisito. Guarda em uma variável de controle o tipo do requisito.

Ação 5: Identificador do requisito. Guarda numa variável de controle o identificador do requisito.

Ação 6: Inclusão de códigos de outros arquivos. Carrega o arquivo e chama os analisadores léxico, sintático e semântico para analisar este texto.

Ação 7: Início de um grupo de valores. Guarda numa variável de controle o primeiro.

Ação 8: Final de um grupo de valores. Coloca no destino o primeiro valor juntamente com o segundo.

Ação 9: Convertendo valores textuais. Converte os tokens numéricos que estão em texto em números para a máquina. Remove as aspas dos tokens literais e converte os caracteres especiais. Coloca numa variável de controle o token encontrado.

Ação 10: Inserindo final de linha. Acrescenta no literal uma quebra de linha.

Ação 11: Inserindo outro texto. Acrescenta no literal um outro literal.

Ação 12: Verificando se o tipo é aceito. Se for um tipo de dados do usuário, verifica se foi declarado. Guarda o tipo em uma variável de controle.

Ação 13: Item de estrutura ou classe. Guarda numa variável de controle o item da estrutura.

Ação 14: Tipo do item de estrutura ou classe. Guarda numa variável de controle o tipo do item da estrutura.

Ação 15: Ponteiro distante. Guarda numa variável de controle a indicação de que o ponteiro se refere a uma área de memória fora dos limites do programa.

Ação 16: Ponteiro próximo. Guarda numa variável de controle a indicação de que o ponteiro se refere a uma área de memória dentro dos limites do programa.

Ação 17: Ponteiro. Guarda numa variável de controle a indicação de que o tipo de dado é um ponteiro.

---

### **Ações para obter as informações gerais sobre o componente:**

Ação 100: Nome do componente. Verifica se o que foi encontrado é um literal.

Verifica se não está sendo chamado outra vez. Coloca no destino o nome do componente.

Ação 101: Descrição do componente. Verifica se o que foi encontrado é um literal. Coloca no destino a descrição do componente.

Ação 102: Arquivo de código. Verifica se o que foi encontrado é um literal. Coloca no destino o nome do arquivo de código.

Ação 103: Descrição do arquivo de código. Verifica se o que foi encontrado algo e se é um literal.

Ação 104: Arquivo de cabeçalho. Verifica se o que foi encontrado é um literal. Coloca no destino o nome do arquivo de cabeçalho.

Ação 105: Descrição do arquivo de cabeçalho. Verifica se o que foi encontrado algo e se é um literal.

Ação 106: Locais dos arquivos extras. Verifica se o que foi encontrado é um literal. Coloca no destino o nome do local extra.

Ação 107: Descrição do local dos arquivos extras. Verifica se o que foi encontrado algo e se é um literal.

Ação 108: Arquitetura do processador/microcontrolador usado. Verifica se o que foi encontrado é um identificador. Verifica se não está sendo chamado outra vez. Se estiver definida a arquitetura então compara com o que foi encontrado, senão coloca na variável. Indica que já foi chamada esta ação.

Ação 109: Descrição do local dos arquivos extras. Verifica se o que foi encontrado algo e se é um literal.

Ação 110: Imagem representativa. Verifica se o que foi encontrado é um literal. Verifica se não está sendo chamado outra vez. Coloca no destino o nome do arquivo de imagem.

Ação 111: Descrição da imagem representativa. Verifica se o que foi encontrado algo e se é um literal.

Ação 112: Tipo de componente. Verifica se o que foi encontrado é um identificador. Verifica se não está sendo chamado outra vez. Verifica se o identificador é “Dispositivo”, “Sistema” ou “Aplicacao”. Coloca no destino o tipo encontrado.

Ação 113: Descrição do tipo de componente. Verifica se o que foi encontrado algo e se é um literal.

Ação 114: Categoria do componente. Verifica se o que foi encontrado é um literal. Verifica se não está sendo chamado outra vez. Coloca no destino a categoria encontrada.

Ação 115: Descrição da categoria. Verifica se o que foi encontrado algo e se é um literal.

Ação 116: Requisitos do componente. Verifica se o que foi encontrado é um identificador. Coloca no destino o requisito encontrado.

Ação 117: Descrição do requisito. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição do requisito encontrado.

---

### **Ações para obter as informações sobre uma propriedade do componente:**

Ação 200: Início da propriedade. Verifica se o que não existe uma outra propriedade com o mesmo nome. Coloca no destino a indicação da propriedade.

Ação 201: Final da propriedade. Verifica se o que foi indicado o tipo da propriedade. Verifica o tipo do valor inicial da propriedade.

Ação 202: Nome da propriedade. Verifica se o que foi encontrado é um literal. Verifica se não está sendo chamado outra vez. Coloca no destino o nome da propriedade.

Ação 203: Descrição da propriedade. Verifica se o que foi encontrado é um literal. Coloca no destino a descrição da propriedade.

Ação 204: Tipo da propriedade. Coloca no destino o tipo da propriedade.

Ação 205: Descrição do tipo da propriedade. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição do tipo da propriedade.

Ação 206: Valor inicial da propriedade. Coloca no destino o valor inicial da propriedade.

Ação 207: Descrição do arquivo de cabeçalho. Verifica se o que foi encontrado algo e se é um literal.

Ação 208: Valores válidos. Coloca no destino os valores.

Ação 209: Descrição dos valores válidos. Verifica se o que foi encontrado algo e se é um literal.

Ação 210: Ativação através de uma condição. Coloca no destino a condição.

Ação 211: Descrição da ativação através de uma condição. Verifica se o que foi encontrado algo e se é um literal.

Ação 212: Definição da propriedade. Coloca no destino o tipo da propriedade.

Ação 213: Descrição da definição da propriedade. Verifica se o que foi encontrado algo e se é um literal.

Ação 214: Requisitos da propriedade. Verifica se o que foi encontrado é um identificador. Coloca no destino o requisito encontrado.

Ação 215: Descrição do requisito. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição do requisito encontrado. Ações para obter as informações sobre uma operação do componente:

Ação 300: Início da operação. Coloca no destino a indicação da operação.

Ação 301: Final da operação. Verifica se o que não existe uma outra operação com o mesmo nome. Verifica o tipo do valor inicial da operação.

Ação 302: Nome da operação. Verifica se o que foi encontrado é um literal. Verifica se não está sendo chamado outra vez. Coloca no destino o nome da operação.

Ação 303: Descrição da operação. Verifica se o que foi encontrado é um literal. Coloca no destino a descrição da operação.

Ação 304: Tipo da operação. Coloca no destino o tipo da operação.

Ação 305: Descrição do tipo da operação. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição do tipo da operação.

Ação 306: Tipo do parâmetro da operação. Coloca no destino o tipo do parâmetro da operação.

Ação 307: Identificador do parâmetro da operação. Coloca no destino o identificador do parâmetro da operação.

Ação 308: Descrição dos parâmetros. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição do parâmetro da operação.

Ação 309: Ativação através de uma condição. Coloca no destino a condição.

Ação 310: Descrição da ativação através de uma condição. Verifica se o que foi encontrado algo e se é um literal.

Ação 311: Requisitos da operação. Verifica se o que foi encontrado é um identificador. Coloca no destino o requisito encontrado.

Ação 312: Descrição do requisito. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição do requisito encontrado. Ações para obter as informações sobre um tipo de dado do usuário:

Ação 400: Início do tipo. Verifica se o que não existe um outro tipo com o mesmo nome. Coloca no destino o tipo.

Ação 401: Final do tipo. Verifica se o tipo de que origina, caso seja um tipo definido pelo usuário, foi declarado.

Ação 402: Descrição do tipo. Verifica se o que foi encontrado é um literal. Coloca no destino a descrição do tipo.

Ação 403: Tipo no qual se origina. Coloca no destino o tipo que o tipo do usuário no qual se origina.

Ação 404: Descrição do tipo no qual se origina. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição do tipo da operação.

Ação 405: Formato usado em uma caixa de texto. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino o formato.

Ação 406: Descrição do formato. Verifica se o que foi encontrado algo e se é um literal.

Ação 407: Arquivo onde está a definição. Coloca no destino o nome do arquivo onde foi definido o tipo.

Ação 408: Descrição do arquivo onde está a definição. Verifica se o que foi encontrado algo e se é um literal. Ações para obter as informações sobre um tipo de dado do usuário:

Ação 500: Início da enumeração. Verifica se o que não existe uma outra enumeração com o mesmo nome. Coloca no destino a enumeração.

Ação 501: Final da enumeração. Verifica se foram indicados os valores constantes.

Ação 502: Tipo da enumeração. Coloca no destino o tipo da enumeração.

Ação 503: Descrição da enumeração. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição do tipo da operação.

Ação 504: Adiciona uma constante. Verifica se o que não existe uma outra constante com o mesmo nome. Coloca no destino a constante.

Ação 505: Indica o valor da constante. Verifica se o tipo da constante é o mesmo da enumeração. Coloca no destino o valor da constante.

Ação 506: Descrição da constante. Verifica se o que foi encontrado algo e se é um literal. Coloca no destino a descrição da constante.

## BIBLIOGRAFIA

- (Aho, 1995) AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores – Princípios, Técnicas e Ferramentas**. Editora LTC, 1995.
- (Booch, 1994) BOOCH, Grady. **Object Oriented Analysis and Design, with Applications**. 2.ed. [S.l.]: The Benjamin/Cummings, 1994.
- (Composes, 2003) **CompOSES – Sistemas Operacionais baseados em Componentes**. URL: <http://www.lacpad.inf.ufsc.br/composes.html>. Acessado em 28/08/2003.
- (Douglas, 98) DOUGLASS, B. P. **Real-Time UML: Developing Efficient Objects for Embedded Systems**. [S.l.]: Addison-Wesley, 1998.
- (Duarte, 2003) DUARTE, M. L., DOTTI, F. L. **Uma Análise de Especificação para Sistemas Distribuídos**. Published by the Campus Global – FACIN – PUCRS, 2003.
- (eCos, 2003) URL: <http://www.redhat.com/embedded/technologies/ecos/>
- (Engels *at al*, 2003) ENGELS, G., KÜSTER, J. AND HECKEL, R. **A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models**. Paderborn, Germany, 2003.
- (Ford *at al*, 2003 ) FORD, B., MAREN, K.V., LEPREAU, J., *at.al*. **The Flux OS ToolKit: Reusable Components for OS Implementation**. Department of Computer Science, University of Utah, 2003.



- (Friedrich, 2000) FRIEDRICH, L. F., JR., J. W. H., STANKOVIC, J. A., HUMPHREY, M. A.  **$\mu$ OS: A Component-based, Dynamically Reconfigurable Kernel for Embedded Computing.** 2000.
- (Gervini, 2003) Gervini, A. I., Wagner F. R. **Criação dos Módulos Funcionais de uma Biblioteca para um Sistem Operacional de Tempo Real.** Artigo. Universidade Federal do Rio Grande do Sul, 2003.
- (Gesser, 2004) GESSER, Carlos Eduardo. **GALS - Gerador de Analisadores Léxicos e Sintáticos.** URL: <http://www.inf.ufsc.br/~gesser/>. Acessado em 29/03/2004.
- (Giese at al, 2003) GIESE, H., TICHY, M., BURMESTER, S., SCHÄFER, W. AND FLAKE, S. **Towards the Compositional Verification of Real-Time UML Designs.** *Software Engineering Group University of Paderborn. Germany, 2003.*
- (Lavazza, 2003) Lavazza, L. **Combining UML and formal notations for modelling real-time systems.** Milano, Italia. 2003.
- (Marcon, 2002) MARCON, C. A. M., CALAZANS, NEY L. V., MORAES, F. G. **Modelagem e Descrição de Sistemas Computacionais – Um Estudo de Caso de Comparação entre as Linguagens VHDL e SDL.** Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) - Faculdade de Informática (FACIN). Porto Alegre, 2002.
- (Massa, 2003) Massa, A. **Embedded Software Development With Ecos.** Prentice Hall. New Jersey, 2003.
- (Ommering at al, 2000) OMMERING, R. V., LINDEN, F. V. D., KRAMER, J., MAGER, J. **The Koala Component Model for Consumer Eletronics Software.** IEEE 7, 2000.
- (Oreizy, 2003 ) OREIZY, P. MEDVIDOVIC, N. TAYLOR, R. N. **Architecture-Based Runtime Software Evolution.** Information and Computer Science University of California, Irvine USA, 2003.
- (OSkit, 2003) URL: <http://www.cs.utah.edu/flux/oskit>
- (Salmito, 2003) SALMITO, T. **SystemC 2.0.1.** URL: <http://www.dimap.ufrn.br/~ivan/Topicos/SystemC%202.0.1.ppt>. Acessado em 02/11/2003.
- (Sangiovanni- SANGIOVANNI-VINCENTELLI, A. MARTIN, G. **Plataform-based desing and software design methodology for**

- Vincentelli et al., 2001). **embedded system**. November-december IEEE 2001.
- (Setzer, 1986) SETZER, V. W., MELO, I.S.H. **A Construção de um Compilador**. Editora Campus, 1986.
- (Silva et al., 1999) SILVA JR., D., COELHO JR., NUNES, C. J.; FERNANDES, A. O.; MATA, J.da., **Conexão de Sistemas Embutidos a Internet**. URL:  
<http://www.lecom.dcc.ufmg.br/~dgns/pubs/1999/ein99.pdf>.  
 Acessado em 25/09/2003
- (Souza, 2000) SOUZA, I. F. **Um Metamodelo da Linguagem de Modelagem, Real Time UML, de Suporte à Criação de Dicionário de Dados para Ferramentas de Modelagem de Sistema Tempo Real, Visando a Verificação de Consistência dos Modelos**. Porto Alegre: CPGCC da UFRGS, 1997.
- (Stankovic, 2000) STANKOVIC, J. A. **VEST: A Tool for Constructing and Analyzing Component Based Operating Systems for Embedded and Real-Time Systems**. Department of Computer Science. University of Virginia. 2000.
- (Urting et al., 2003) URTING, D. BERBERS, Y., VAN BAELEN, S. ET AL. **A Tool for Component Based Design of Embedded Software**. Department of Computer Science Catholic University of Leuven Celestijnenlaan 200A, B-3001 Leuven, Belgium, 2004.
- (Villela, 2001) VILLELA, C. V. **Ambiente Baseado em Componentes para o Desenvolvimento de Sistemas Computacionais Microcontrolados Distribuídos**. Dissertação de Mestrado. UFRGS, 2001.
- (Wagner, 2001) WAGNER, F.R, CARRO, L. **Sistemas Operacionais Embarcados**. November-december IEEE 2001.
- (Wolf, 2001) WOLF, W. **Computers as Components: Principles of Embedded Computing System Design**. Morgan Kaufmann Publishers. San Diego, 2001.
- (Zelenovshy, 2003) ZELENOVSHY, R.. MENDOÇA, A.. **Introdução aos Sistemas Embutidos**. URL:  
<http://www.mzeditora.com.br/artigos/embut.htm>. Acessado em 30/10/2003.

