EDUARDO LORENZETTI PELLINI

UM ARCABOUÇO PARA APLICAÇÕES EM TEMPO REAL EM SISTEMAS DE POTÊNCIA

São Paulo
2010

EDUARDO LORENZETTI PELLINI

UM ARCABOUÇO PARA APLICAÇÕES EM TEMPO REAL EM SISTEMAS DE POTÊNCIA

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Engenharia Elétrica.

São Paulo
2010

EDUARDO LORENZETTI PELLINI

UM ARCABOUÇO PARA APLICAÇÕES EM TEMPO REAL EM SISTEMAS DE POTÊNCIA

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Engenharia Elétrica.

Área de concentração:
Sistemas de Potência

Orientador:
Prof. Dr. Clóvis Goldemberg

São Paulo
2010

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, ... de .... de 20.....

Assinatura do autor: ....

Assinatura do orientador: ....

FICHA CATALOGRÁFICA

Pellini, Eduardo Lorenzetti

Um arcabouço para aplicações em tempo real em sistemas de potência / E.L. Pellini. – – ed.rev. – São Paulo, 2010.

183 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Energia e Automação Elétricas.

1. Sistemas elétricos de potência 2. Eletrônica embarcada 3. Simulação de sistemas 4. Tempo real 5. Frameworks I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Energia e Automação Elétricas II.t.

Clóvis,

por compartilhar comigo tanto de

sua experiência pessoal e profissional.

Você é inesquecível.

Você é o cara !

Agradecimentos

Para a minha estrela Alessandra, meu *pappa* ícone Mauro, minha *mamma* incansável Leonilda e meu mano irmão Luiz Fernando por todo o carinho, atenção e esforços dedicados a mim.

Aos amigos "*Jedi Master*" Giovanni Manassero Jr. e "*USA Samurai*" Renato Mikio Nakagomi pelas diversas empreitadas, risadas e terrores que enfrentamos juntos na POLI nos últimos 10 anos. E que venham mais anos de emoções.

Aos mestres Eduardo Cesar Senger, Walter Kaiser, Luiz Antonio Baccalá e Luiz Antonio Barbosa Coelho pela insistência, persistência e paciência em me motivar e me orientar para o término desse trabalho.

Aos amigos da OptSensys Osni Lisboa e José Carlos Juliano de Almeida pela confiança em meu trabalho, e pela ajuda em ceder boa parte do tempo para a conclusão dessa obra.

Aos amigos, colegas e professores da POLI: André Suzuki, Guido Stolfi, José Geraldo B. M. de Andrade, Josemir Coelho dos Santos, Luiz Cera Zanetta Jr. e Wilson Komatsu.

Ao amigo e Prof. Dr. Clóvis Goldemberg, que estará sempre em nossos corações, imortalizado por sua obra, sua excelência profissional, sua dedicação ao ensino e sua personalidade tão marcante e inesquecível. Clóvis, é muito difícil ficar sem nossas conversas técnicas mirabolantes, os papos sobre filmes eruditos e suas histórias maravilhosas da engenharia elétrica. Onde quer que você esteja, desejo que você possa ver e se emocionar com as sementes que você plantou em vida, enquanto elas nascem, crescem e florescem como suas tão queridas Primaveras. Obrigado por toda a confiança, amizade, carinho e por permitir que eu tenha participado de tantas coisas de sua vida e de sua família. Lutarei por fazer para alguém, aquilo que você fez por mim com tanta generosidade e amor. Até um dia meu amigo.

Resumo

A área de pesquisa e prototipagem de soluções para Sistemas de Potência compartilha, com outras áreas da engenharia, vários problemas relacionados a *software*, principalmente, os seus custos e o seu tempo de desenvolvimento. Mais especificamente, nas aplicações com sistemas embarcados e dispositivos computacionais em tempo real, a presença de fatores como a complexidade dos algoritmos, os requisitos críticos de desempenho e as restrições impostas pelo *hardware* da aplicação, fazem com que o desenvolvimento do *software* evolua de forma muito lenta. Finalmente, quando um projeto está terminado, seu *software* já é obsoleto ou completamente incompatível para uso em novos projetos, com um *hardware* novo. Esse trabalho apresenta uma possível solução a esses problemas, por meio de um arcabouço de *software* para aplicações em tempo real. Através de uma metalinguagem de descrição de fluxos de dados baseada em blocos, e de ferramentas de compilação e interpretação, esse arcabouço permite abstrair o projeto, a implementação e os testes da aplicação, do desenvolvimento das demais partes do dispositivo, favorecendo um projeto sistemático em módulos, o futuro reúso de códigos, a fácil manutenção de algoritmos e a execução de testes cruzados, entre plataformas, com previsibilidade de resultados. O arcabouço foi elaborado e testado em um cenário de Sistemas de Potência, com a criação de um relé digital de proteção contra sobrecorrentes, com aquisição de dados via barramento de processo da norma IEC 61850. Entretanto, toda a metodologia desse trabalho pode ser aplicada a qualquer outra área correlata, mediante a extensão de sua metalinguagem.

Abstract

The research and prototype area for Power System solutions share, among other engineering disciplines, several problems concerning software, mainly their developing time and costs. More specifically, in embedded system and real time computing devices applications, the presence of factors, such as the algorithm complexity, the critical performance requirements and other restrictions imposed by the application hardware, makes the software development to slowly evolve. Finally, when a project is over, its software is already obsolete or completely incompatible for use in other projects, with a new hardware. This work presents a possible solution to these problems, through a software framework for real time applications in Power Systems. Through a block based data stream description metalanguage, and compiling and interpreting tools, this framework allows the application design, implementation and testing procedures to be abstracted from the development of other device parts, permitting a systematic and modular project, the code reuse in the future, easy algorithm maintenance and the execution of cross-platforms tests with predictable results. The framework was created and tested in Power System scenarios, especially in the construction of a digital protection overcurrent relay, with data acquisition through the IEC 61850 process bus. However, the entire methodology of this study could be applied to any other related area, by extending its metalanguage with the appropriate building blocks.

Lista de Figuras

1 Metodologia de desenvolvimento em espiral, segundo Boehm. 21

2 Topologia típica de *hardware*. 34

3 Topologia típica de *software*. 40

4 Etapas típicas de desenvolvimento de um projeto. 53

5 Plataforma de *hardware*, *software* e a parte embarcada do arcabouço em HRTCS 81

6 Processo simplificado de síntese de *hardware* a partir de uma descrição comportamental de um circuito. 83

7 Tela do *software* MachineLogic com recursos de programação da linguagem "FBD". 86

8 Tela do *software* 4DIAC IDE para *design* de aplicações de automação e controle segundo a IEC 61499. 87

9 Funcionamento básico do *framework* "TB", de apoio ao desenvolvimento de um HRTCS para Sistemas de Potência. 91

10 Bloco elementar, generalizado, do "TB". 92

11 Funcionamento básico do compilador do "TB". 101

12 Blocos do sistema de teste do Apêndice C, numerados pelo processo de compilação. 102

13 Grafo de conectividade dos blocos do sistema de teste do Apêndice C. . . . 103

14 Destaque para os relacionamentos de dependência dos blocos 3, 6, 10 e 11 para com blocos sequenciais 5 e 9, no caso, que representam os integradores do sistema do ApêndiceC. 103

15 Grafo refinado do sistema de teste do Apêndice C após quebra de relacionamento dos blocos sequenciais, 5 e 9. 104

16 Ordem geral de execução dos blocos baseado no grafo de conectividade refinado. 105

17 Exemplo de máquina de estados para o funcionamento do interpretador "TB". 106

18 Saída da simulação de um dos *frameworks* "TB", do Vissim e do algoritmo em C puro para o caso teste. As curvas são muito semelhantes e estão sobrepostas. 115

19 Erro absoluto da saída do *framework* e do algoritmo puro em ANSI C, com relação ao resultado do Vissim, para o mesmo caso teste. 115

20 Diagrama de blocos modificado do sistema de teste para análise de granularidade. 125

21 Fluxo de dados simplificado de um IED de proteção. 131

22 Universal Dataset enviado no ASDU de um pacote GOOSE de valores amostrados. 137

23 Cenário de aplicação do IED desenvolvido nesse trabalho. 141

24 Diagrama de blocos do IED "A" especificado. 143

25 Pacotes GOOSE SV e GOOSE *station bus* capturados pelo *software* Wireshark. 146

26 Resultado do processamento digital de sinais para tratamento da corrente da fase A. 147

27 Sinais digitais internos da lógica de proteção implementada. 148

28 Pacotes GOOSE enviados na cadência errada devido a operação da rede Ethernet do computador PC, sem requisitos de desempenho em tempo real. 149

29 Kit de desenvolvimento com processador ARM9 STR912 da STMicroelectronics, excutando o *framework* “TB” desenvolvido. 150

30 Diagrama de blocos do sistema utilizado nos testes do compilador e interpretador do arcabouço. 165

Lista de Tabelas

1 Desempenho da simulação teste. Tempos em [μs]. 119

2 Ocupação de memória. Valores em [bytes] e [%]. 121

3 Desempenho da simulação teste com bloco de maior granularidade. Tempos em [μs]. 125

4 Detalhes do tempo de execução de alguns blocos para uma iteração de processamento do arcabouço "TB". 145

5 Tempos obtidos para execução do *framework* no microcontrolador ARM9. . 151

Lista de Abreviaturas

ANSI *American National Standards Institute*

APDU *Application Protocol Data Unit*

API *Application Programming Interface*

ASDU *Application Service Data Unit*

ASIC *Application Specific Integrated Circuit*

ATP/EMTP *Alternative Transient Program / Electromagnetic Transient Program*

AVR *Automatic Voltage Regulator*

CAE *Computer Aided Engineering*

CASE *Computer Aided Software Engineering*

CAN *Controller Area Network*

CLP Controlador Lógico Programável

COMTRADE *Common Format for Transient Data Exchange*

CPLD *Complex Programmable Logic Device*

CPU *Central Processing Unit*

CSMACD *Carrier Sense Multiple Access with Collision Detection*

CSV *Comma Separated Values*

DASSL *Differential Algebraic System Solver*

DRAM *Static Random Access Memory*

DRE *Distributed Real-Time Embedded System*

DSP *Digital Signal Processor*

EDO Equações Diferenciais Ordinárias

EEPROM *Electric Erasable and Programmable Read Only Memory*

FBD *Function Block Diagram*

FIR *Finite Impulse Response*

FLOPS *Floating Point Operations per Second*

FPGA *Field Programmable Gate Array*

GOOSE *Generic Object Oriented Substation Event*

HAL *Hardware Abstraction Layer*

HiL *Hardware-in-the-Loop*

HPC *High Performance Computing*

HRTCS *Hard Real-Time Computing System*

IDE *Integrated Development Environment*

IEC *International Electrotechnical Commission*

IED *Intelligent Electronic Device*

IEEE *Institute of Electrical and Electronics Engineers*

IHM Interface Homem-Máquina

IL *Instruction List*

IPS *Instructions Per Second*

ISO *International Organization for Standardization*

LAN *Local Area Network*

PID Proporcional-Integral-Derivativo

PLC *Programmable Logic Controller*

PLD *Programmable Logic Device*

PSS *Power System Stabilizer*

RAM *Random Access Memory*

ROM *Read Only Memory*

RTOS *Real-Time Operating System*

RTL *Register Transfer Logic*

SDCD Sistema Digital de Controle Distribuído

SE Sistema Executivo

SFC *Sequential Function Chart*

SIMD *Single Instruction - Multiple Data*

SMP	*Symmetric Multiprocessing*
SO	Sistema Operacional
SRAM	*Static Random Access Memory*
SSC	Sistema de Supervisão e Controle
ST	*Structured Text*
USB	*Universal Serial Bus*
USP	Universidade de São Paulo
WAN	*Wide Area Network*
WCET	*Worst Case Execution Time*

Sumário

1 Introdução 19

1.1 Motivação 20

1.2 O tema "Um arcabouço para aplicações em tempo real em sistemas de potência" 22

1.3 Organização do trabalho 24

1.4 Comentários a respeito do documento 25

1.4.1 Forma 25

1.4.2 Língua 25

1.4.3 Nomes e marcas registradas 25

2 Sistemas computacionais para aplicações em tempo real determinístico 26

2.1 Sistemas embarcados 26

2.2 Simuladores *hardware-in-the-loop* 28

2.3 Comparação entre os sistemas 29

2.3.1 Diferenças 29

2.3.2 Semelhanças 30

2.4 Exemplos de aplicações em tempo real determinístico para Sistemas de Potência 33

2.5 Detalhes da arquitetura de *hardware* 34

2.5.1 Interfaces de entrada e interfaces de saída 34

2.5.2 Interfaces de comunicação 35

2.5.3 Unidades de processamento central 36

2.5.4 Memória 38

2.5.5 Outros subsistemas de apoio 39

2.6 Detalhes da arquitetura de *software* 39
2.6.1 Camada de abstração de *hardware* 40
2.6.2 Processos e tarefas 40
2.6.3 Sistemas operacionais e executivos 41
2.7 Detalhes das tarefas de *software* para Sistemas de Potência 44
2.7.1 Tarefas em simuladores em tempo real HiL 44
2.7.2 Tarefas em sistemas eletrônicos embarcados 49

3 Estado da arte de desenvolvimento de sistemas computacionais em tempo real determinístico 52

3.1 Dificuldades para criação de uma aplicação 52
3.1.1 Problema do particionamento 53
3.1.2 Programação 54
3.2 Soluções de metodologias de desenvolvimento 55
3.2.1 Métodos organizacionais 56
3.2.2 Elementos de *software* prontos 56
3.2.3 Reuso de código 57
3.2.4 Padrões de projeto 58
3.2.5 Técnicas de *co-design* 58
3.2.6 Plataformas 60
3.2.7 Arcabouços ou *frameworks* 61
3.2.8 Qual a solução para todos os problemas ? 62
3.3 Soluções disponíveis na técnica 63
3.3.1 Diretivas para desenvolvimento de *software* 63
3.3.2 Alternativas de linguagens de programação 64
3.3.3 Ambientes integrados de desenvolvimento 67
3.3.4 Soluções de plataformas de *hardware* 69
3.3.5 Sistemas operacionais e executivos 71
3.3.6 Ambientes dedicados para estudos de sistemas de potência 73

3.3.7 Arcabouços já desenvolvidos 74

4 Arcabouço para aplicações em tempo real em sistemas de potência 79

4.1 Características 79

4.2 Cenário de utilização 80

4.3 Valores agregados de outras áreas 81

4.3.1 Síntese de *hardware* de alto nível 81

4.3.2 Normas IEC 61131 e IEC 61499 86

4.3.3 Forma de processamento da metalinguagem 87

4.3.4 Portabilidade 88

4.3.5 Linguagem de implementação das ferramentas 88

4.3.6 Experiência de outros trabalhos 89

4.4 Desenvolvimento 89

4.4.1 Biblioteca de blocos *template* 90

4.4.2 Estrutura de um bloco *template* 92

4.4.3 Metalinguagem de descrição 96

4.4.4 Pré-processador 99

4.4.5 Compilador 100

4.4.6 Interpretador 105

4.4.7 Outros recursos e observações 107

5 Resultados e discussão 111

5.1 Testes de validação 112

5.1.1 Consistência numérica 114

5.1.2 Desempenho computacional 117

5.1.3 Ocupação de memória 121

5.1.4 Portabilidade a outras plataformas 123

5.1.5 Aumento da granularidade dos blocos 124

5.2 Aplicação em relé digital com IEC 61850 126

5.2.1 Detalhes da norma IEC 61850 126

5.2.2 Detalhes do desenvolvimento do IED com o *framework* “TB” 130

5.2.3 Testes da aplicação “TB” do IED de proteção 144

5.3 Comentários finais dos resultados 153

6 Conclusão **154**

6.1 Comentários gerais 154

6.2 Desenvolvimentos futuros 156

Apêndice A – Exemplo de código fonte para declaração de um bloco **158**

A.1 Arquivo header simIntegrator.h 158

A.2 Arquivo de implementação simIntegrator.c 159

Apêndice B – Exemplo de arquivo de entrada com o *design* **164**

B.1 Fluxo de dados 164

B.2 Listagem resultante na metalinguagem 165

B.3 Listagem equivalente em linguagem ANSI C 166

Apêndice C – Exemplo de arquivo de saída do compilador **168**

Apêndice D – Arquivo de configuração para a aplicação do relé **171**

D.1 Listagem resultante na metalinguagem 171

Referências **176**

1 Introdução

Desde a minha formação em Engenharia em 2000, participo do projeto e desenvolvimento de soluções (principalmente as embarcadas) para o setor de sistemas de potência, junto a Universidade de São Paulo e várias empresas da iniciativa privada. Nessas soluções, estive particularmente envolvido na área de *software* e *firmware* até que, mais recentemente, acabei por me envolver também na área de *hardware* dessas aplicações. Por conta dessa experiência, foi possível constatar que o tempo e o custo para a criação de um *software* são muito diferentes daqueles necessários para a criação de seu *hardware*.

Essa constatação, infelizmente, só é clara do ponto de vista do desenvolvedor da aplicação. Do ponto de vista do cliente, muitas vezes fui questionado sobre a razão pela qual um *firmware* tem um tempo de desenvolvimento muito superior ao de seu *hardware*? Muitas vezes estive presente em inúmeras discussões, a respeito do motivo pelo qual um *firmware* tem um custo muito mais elevado que o *hardware* em que vai ser executado? Muitos cabelos têm sido perdidos para convencer as pessoas que o prazo e o custo para a criação e a implementação de um *software* nunca serão inferiores aos do seu *hardware*, já que o desenvolvimento e o sucesso do primeiro depende necessariamente do *design*, da especificação e da implementação do segundo, incluindo as escolhas (certas ou erradas) feitas nesse processo.

Outro problema que agrava o custo e ao tempo de desenvolvimento de um *software* é o seu grau de liberdade. Apesar dos esforços e das técnicas para se estabelecerem as ficcionais "especificações", existe uma tendência dos clientes e usuários solicitarem mais e mais recursos ou aperfeiçoamentos, às vezes, mesmo antes do término de qualquer etapa do também utópico "cronograma". Isso se deve ao fato de que, em termos de algoritmos e programação, praticamente tudo é possível. Entretanto, a possibilidade de se realizar uma mudança não significa que esta seja simples ou rápida de ser implementada, bastando apenas a alteração de algumas linhas de código para se ter o resultado desejado. Algumas alterações ingênuas em poucas linhas de código de um programa funcional podem torná-lo completamente imprestável ou gerar um "bug" tão estranho que os programadores preferem reescrever partes do *software* a depurar o problema[1].

[1] Como diz Ganssle1999, "Qualquer idiota pode escrever código fonte". Ou seja, nunca espere que um código seja brilhante, simples, inteligível e auto-documentado. Muito pelo contrário, às vezes ele é uma "bomba relógio", que só pode ser 'desarmada' pelo próprio autor.

Uma ilustração desses fatos ocorreu em minha tese de mestrado (PELLINI, 2005), apresentada à Escola Politécnica da Universidade de São Paulo. Nesta foi apresentado um "Simulador em tempo real de hidrogeradores", que consiste em um esquema de *hardware* e *software* dedicados para a representação, em tempo real determinístico, do comportamento de um gerador síncrono de pólos salientes ligado a um barramento infinito. O projeto foi desenvolvido em quatro anos, dos quais 95% do tempo foram dedicados somente ao *software*, finalizado em quase seis vezes mais o prazo estipulado em cronograma, após várias "mutações" em suas especificações iniciais. As dificuldades enfrentadas foram advindas, em sua maior parte, de restrições e problemas encontrados ao se adaptar um *hardware* já existente para a aplicação desejada. O *software* do projeto, inicialmente escrito em linguagem de alto nível (ANSI C), acabou por ser finalizado com quase 80% do seu código escrito em *Assembly* apenas para cumprir os requisitos de desempenho. Apesar dos atrasos, o resultado alcançou todas as expectativas, entretanto, culminou em um único comentário: - "Que tal agora MODIFICAR O *SOFTWARE* do simulador mono-máquina para um simulador multimáquina, com quantos geradores quiser, controles ajustáveis pelo usuário, etc. etc. e etc ?"

A resposta a essa pergunta surge na forma de um desabafo. Cansado de ter de reescrever ou readaptar *software* para *hardware* e converter linguagens de alto nível para baixo nível para obter desempenho, tento solucionar o cenário de um simulador "reconfigurável", bem como outros cenários de desenvolvimento de aplicações embarcadas com requisitos de processamento em tempo real, com "Um arcabouço para aplicações em tempo real para sistemas de potência".

1.1 Motivação

Os argumentos salientados anteriormente, de que os custos de desenvolvimento de um *software* são maiores que os custos de desenvolvimento de um *hardware*, não são verdadeiros em quaisquer áreas ou aplicações. Entretanto, a afirmação é particularmente verdadeira para as áreas de pesquisa, desenvolvimento e prototipagem de algoritmos e dispositivos, e para uma categoria específica de equipamentos denominados "reconfiguráveis" ou "programáveis", ou seja, naqueles sistemas cujos algoritmos básicos devem ser versáteis e flexíveis para permitir modificações e ajustes a várias aplicações distintas.

Nessas áreas é praticamente impossível adotar-se um modelo de desenvolvimento do tipo *Waterfall* (ROYCE, 1970), onde, a partir de uma especificação inicial bem definida, chega-se ao dispositivo e ao produto final após um certo intervalo de tempo e com um custo previsível.

Observando com detalhes as áreas de desenvolvimento e prototipagem, percebe-se que

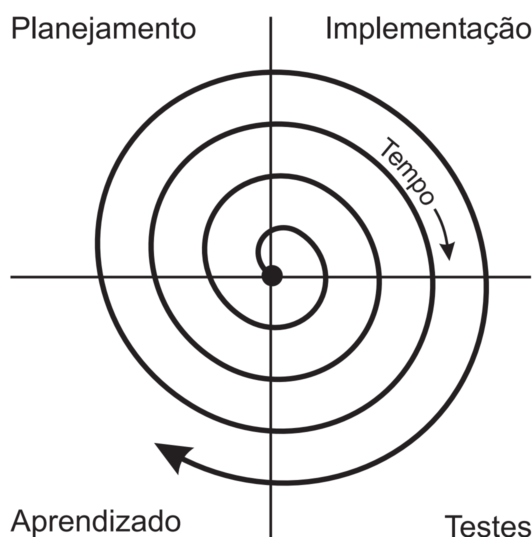

Figura 1: Metodologia de desenvolvimento em espiral, segundo Boehm.

essas costumam utilizar novas tecnologias e conceitos para se obter vantagens competitivas e elementos diferenciais para uma determinada aplicação. Entretanto, esses novos recursos muitas vezes não são "maduros" e outros problemas, sem precedentes, surgem durante a sua implantação. Além disso, mesmo durante a fase de especificação inicial do projeto, não há total clareza dos cenários de utilização ou de todos os problemas que irão surgir mediante a aplicação dessa nova solução. Dessa forma, diversas dificuldades (ou mesmo impossibilidades) técnicas surgem durante a linha original de desenvolvimentos e testes, que culminam no redesenho ou na adaptação das especificações iniciais e no reinício das atividades. Fazendo uma analogia ao mapa de um tesouro, os horizontes podem até estar bem definidos (encontrar o tesouro), mas o caminho não está claramente trilhado, ou é completamente desconhecido.

Nesse contexto, o desenvolvimento torna-se um processo contínuo, iterativo e baseado essencialmente na tentativa-e-erro. Esse modelo é bem conhecido e estudado na literatura, sendo apropriadamente chamado de "espiral", como descrito por (BOEHM, 1986) e (WILSON; RAUCH; PAIGE, 1992). Tal processo é claramente mais difícil e oneroso em termos de custos e tempo de mercado do que aqueles que permitem o uso do modelo *Waterfall*. O autor (BOEHM, 1986) ilustra o processo, como mostrado na Fig. 1.

Na Fig. 1, o comprimento da linha grossa representa o tempo gasto no processo de desenvolvimento desde o seu início até que todas as características e recursos sejam aceitos. Em uma interpretação matemática, o "custo" do processo poderia ser obtido pela integral de linha da espiral, com pesos diferentes ao longo do seu comprimento, correspondentes aos custos com homem/hora, infraestrutura e equipamentos de cada quadrante (planejamento,

implementação, testes e aprendizado).

Apesar desses fatos se aplicarem tanto para o desenvolvimento do *hardware* como do *software*, é notadamente nesse último que ocorrem os maiores gastos, dado seu grau de liberdade e sua flexibilidade para acomodar mudanças de especificações.

Para se resolver essas dificuldades e tornar o processo de desenvolvimento mais previsível, uma série de conceitos e técnicas têm sido desenvolvidas e utilizadas, tais como:

- Desenvolvimento sob plataformas consolidadas, como salientado por (SANGIOVANNI-VINCENTELLI; MARTIN, 2001);
- Utilização de padrões de projeto (do inglês, *Design Patterns*), como os apontados por (DOUGLASS, 1997), estimulando o "reúso" de rotinas e códigos já existentes e testados;
- Aplicações de normas, tais como a (IEC61131, 2003) e a (IEC61499, 2005), para padronizar linguagens, metodologias e documentações de projetos;
- Aplicações de ferramentas de apoio para testes, controle de versão, documentação e revisão, e;
- Aplicação de outros modelos de desenvolvimento, tais como o Agile (BEEDLE et al., 2001).

Entretanto, no caso de pequenos sistemas embarcados, sistemas de controle, simuladores em tempo real e algumas outras plataformas reconfiguráveis de pequeno porte, tais como reguladores, controladores e dispositivos eletrônicos inteligentes de automação e de proteção para o setor elétrico, não existe uma ferramenta de *design* ou uma metodologia padrão, abrangentes o suficiente e com o desempenho necessário, que permita aplicar os conceitos acima durante o desenvolvimento de um dispositivo. O "arcabouço" apresentado nesse trabalho pretende fornecer uma contribuição nesse nicho de desenvolvimento de sistemas.

1.2 O tema "Um arcabouço para aplicações em tempo real em sistemas de potência"

Esse arcabouço (do inglês, *framework*) consiste em uma metalinguagem de programação, na forma de blocos de instruções e funções tipicamente encontradas em sistemas embarcados e simuladores, sobretudo em Sistemas de Potência, que podem ser interligadas conforme a necessidade do usuário para formar sistemas maiores. A metalinguagem

é processada por uma máquina virtual que abstrai os detalhes da camada de *hardware* [2], tornando sua interface mais simples e transparente para o engenheiro de desenvolvimento de *software*, permitindo alterações de forma sistemática e com resultados previsíveis.

Apesar de semelhante em conceito a outras linguagens, metalinguagens e ferramentas, o *framework* descrito nesse trabalho difere dessas, pois pretende atingir simultaneamente os seguintes objetivos:

- Obter elevados graus de desempenho e determinismo em tempo real;
- Ocupar pouca memória, permitindo sua utilização também em pequenos dispositivos microcontrolados;
- Permitir a utilização de recursos de *hardware* específicos de algumas arquiteturas;
- Permitir uma arquitetura distribuída entre várias plataformas (homogêneas ou heterogêneas);
- Permitir o desenvolvimento dos testes em computadores *desktop* convencionais e permitir a portabilidade para outras plataformas, mantendo a coerência e a previsibilidade dos resultados, e;
- Ser um ambiente completamente aberto ao público para futuras contribuições e desenvolvimentos (segundo a licença GNU)[3].

As aplicações vislumbradas para essa ferramenta são aquelas onde existe a necessidade de um sistema:

- Eletrônico-digital, com inúmeros tipos de interfaces;
- Com núcleo microprocessado e com baixa ocupação de memória ROM e RAM;
- Que apresente um tempo de resposta a estímulos externos da ordem de microsegundos a dezenas de milisegundos;
- Com capacidade de processamento elevada;
- Com determinismo na cadência de execução de suas tarefas;
- Que possa ser facilmente reconfigurado e ajustado para outros cenários;
- Que possa ser desenvolvido, testado e simulado fora do seu ambiente nativo;

[2] Alguns autores preferem usar o termo *middleware* para esse tipo de arquitetura de *software*.

[3] Ao invés de "Copyright - All right reserved", esse trabalho se baseia no "Copyleft - All rights reversed", segundo Don Hopkins e a *Free Software Foundation* de Richard Stalman.

- Que possa ser portado a várias plataformas, e;
- Que possa acompanhar as mudanças e a evolução do *hardware*, sem maiores impactos no redesenho do *software*.

Tais aplicações remetem tanto ao tema de sistemas embarcados quanto ao tema de simuladores em tempo real, razão pela qual ambos são citados diversas vezes ao longo desse trabalho e considerados como um único tema: "Sistemas Computacionais de Tempo Real Determinístico" - apesar de também possuírem algumas diferenças notáveis.

Todos os critérios e exemplos utilizados para o *design* das ferramentas foram recolhidos de cenários de aplicações na área de Sistemas de Potência, em função da empatia do autor pelos assuntos dessa área, de sua multidisciplinaridade, dos seus requisitos técnicos ímpares de tempo de resposta, de capacidade de processamento e de interfaces de entrada, saída e comunicação. Entretanto, essa metodologia pode ser aplicada, com sucesso, em quaisquer outras áreas correlatas.

1.3 Organização do trabalho

O Capítulo 2 apresenta uma contextualização dos requisitos das aplicações em Sistemas de Potência, além de detalhes do projeto de *hardware* e *software* dessas aplicações. Esse capítulo corresponde a uma "análise de domínio" desses tipos de aplicações, e é necessária para o estabelecimento da abrangência do arcabouço desenvolvido nesse trabalho.

No Capítulo 3 são expostos os problemas de desenvolvimento, principalmente de *software*, além de uma revisão do estado da arte de plataformas e técnicas que se propõem a implantar soluções para essa área. Cada solução disponível na técnica é avaliada quanto a sua estrutura, desempenho e capacidade de atendimento aos requisitos da área de simuladores e sistemas embarcados para aplicações em Sistemas de Potência.

O Capítulo 4 apresenta o desenvolvimento de uma dessas soluções: um arcabouço de *software* para auxílio na criação, prototipagem e testes de aplicações embarcadas em tempo real em sistemas de potência, incluindo o *design* de sua metalinguagem de apoio e o projeto e a implementação de suas ferramentas de compilação e análise.

O Capítulo 5 apresenta alguns resultados da aplicação desse arcabouço no desenvolvimento de um pequeno sistema dinâmico, para operação em tempo real, com o propósito de validar matematicamente os resultados e verificar o desempenho do sistema. Ao final, é mostrada a aplicação desse arcabouço no desenvolvimento de um relé de proteção programável de sobrecorrente temporizada, com recursos da norma IEC 61850 para comunicação horizontal em barramento de processo e automação.

O Capítulo 6 apresenta as conclusões e as sugestões para aplicações e melhorias futuras.

Ao final são apresentados apêndices contendo: um código-fonte com um exemplo de declaração de um bloco do arcabouço em linguagem ANSI C, um exemplo de codificação de um sistema dinâmico usando a metalinguagem do arcabouço, um exemplo de saída de dados do compilador e a listagem da meta-linguagem utilizada para constituir a aplicação do relé de proteção com IEC 61850.

1.4 Comentários a respeito do documento

1.4.1 Forma

Esse trabalho possui formatação e *layout* realizados com base nas recomendações das normas NBR6024, NBR6027, NBR6028 e NBR1472 e da "Norma de apresentação de dissertações e teses da Escola Politécnica da Universidade de São Paulo".

O documento foi elaborado em TEX , utilizando as classes padronizadas da (*ABNTEX*, 2010), o estilo "Poli" desenvolvido por (HüBNER, 2009) e a distribuição MikTEX, versão 2.8.

1.4.2 Língua

Ao longo do texto, muitas vezes foi dada preferência ao uso de termos e acrônimos, tradicionalmente empregados no meio técnico, em língua inglesa. Em cada ocasião em que esses termos são citados, é feito um parênteses com suas respectivas (e possíveis) traduções para a língua portuguesa.

Foram seguidas as orientações do Segundo Protocolo Modificativo do Acordo Ortográfico de 1990 da Língua Portuguesa, definido pela Comunidade dos Países de Língua Portuguesa e adotado pelo Brasil em 2008.

1.4.3 Nomes e marcas registradas

Ao longo do texto são feitas diversas citações a empresas, produtos, tecnologias, marcas e patentes.

Foram omitidas as considerações legais e os símbolos de *trademark* e *copyright*, devendo o leitor atentar que tais nomes e marcas são protegidas por direito autorais e são propriedades exclusivas das respectivas empresas citadas ao longo do texto.

2 Sistemas computacionais para aplicações em tempo real determinístico

Há dois cenários de importância, principalmente para a área de Sistemas de Potência, com aplicações que possuem requisitos de desempenho em tempo real (LAPLANTE, 1996):

- Os dispositivos embarcados, e;
- Os simuladores em tempo real.

Como será mostrado nesse capítulo, esses dois cenários de aplicação compartilham muitas características comuns, sobretudo as dificuldades, custos e tempo para o desenvolvimento de seu *software*.

2.1 Sistemas embarcados

O termo "Sistemas Embarcados" (do inglês, *Embedded Systems*) não possui uma definição clara e única (NOERGAARD, 2005). Um conceito amplo, mas preciso, é aquele encontrado em (HENZINGER; SIFAKIS, 2007), que diz: - "Um sistema Embarcado é um artefato de engenharia que envolve computação, sujeito a um conjunto de restrições físicas, advindas do meio externo, que afetam a forma como essa computação deve ser executada e como essa deve reagir mediante solicitações externas". Nesse contexto, é possível classificar a maioria desses sistemas como sendo uma classe de dispositivos eletrônicos:

- Dedicados;
- Com algum núcleo digital microprocessado;
- Com recursos limitados de interface, memória, capacidade de processamento, consumo de energia e configuração;
- De funcionamento autônomo;

- Instalados no local de sua aplicação [1].

Esses sistemas podem ser encontrados em inúmeras aplicações que, apesar de possuírem requisitos técnicos diferentes, são soluções que apresentam arquiteturas muito semelhantes de *hardware* e *software* (NOERGAARD, 2005).

Na atualidade, é possível encontrar um sistema embarcado em um simples *Flash-Drive* USB, ou até em um conjunto de sistemas embarcados distribuídos ao longo de uma rede de comunicação dentro de um veículo ou de uma aeronave moderna. Tais sistemas podem estar empregados nas mais diversas áreas, incluindo todas as áreas das Engenharias, e até mesmo Medicina e Biotecnologia. A parcela do mercado mundial representada por esses sistemas é importante, com um valor de mercado de mais de 90 bilhões de dólares em *hardware* e mais de 2,2 bilhões em *software*, com taxas de crescimento de cerca de 4% ao ano, segundo dados de 2009 (JOSHI, 2009).

Uma parcela considerável desses dispositivos embarcados são empregados em sistemas de controle e automação, onde permanecem continuamente executando tarefas de leitura de grandezas externas, tarefas de cálculo de seus algoritmos internos e tarefas de publicação de suas decisões de controle ao mundo exterior. Essa operação contínua, em *loop* (cíclica), revela outro requisito importante desses sistemas, que é a cadência com que essas operações são executadas. Para que um dispositivo dessa categoria se comporte de forma correta e previsível, é imprescindível que haja um mecanismo de sintonia[2] e sincronismo[3] de suas operações com o tempo decorrido no mundo real. Esse mecanismo pode ser realizado empregando-se um relógio temporizador interno, bem como sinais externos de sincronização. De qualquer forma, tais dispositivos são denominados, apropriadamente, de “Sistemas Embarcados em Tempo Real” por (NOERGAARD, 2005) e (GANSSLE, 1999) ou, no caso específico de aplicações distribuídas, *Distributed Real-Time Embedded Systems* (DREs), como citado por (EMERSON; NEEMA; SZTIPANOVITS, 2006).

Alguns tipos de dispositivos embarcados existentes no mercado possuem as mesmas premissas básicas de execução de algoritmos em tempo real determinístico, além de uma determinada capacidade computacional livre, para a programação e a execução de algoritmos de controle e automação criados pelo próprio usuário. Esses dispositivos são os Controladores Lógicos Programáveis (CLP’s) (do inglês, *Programmable Logic Controllers* (PLCs)) e Dispositivos Eletrônicos Inteligentes (do inglês, *Intelligent Electronic Devices* (IED’s)).

[1] Alguns autores destacam que é dessa característica que surge o nome *embedded systems* “sistema embarcado” (do inglês, *embedded systems*), ou seja, aquele sistema embutido na sua própria aplicação.

[2] Sintonia - quando a frequência de operação é próxima da frequência desejada.

[3] Sincronismo - quando o atraso com relação a uma referência de tempo é inferior a um patamar máximo desejado.

Devido a multidisciplinaridade na área de Sistemas de Potência, há uma grande variedade de aplicações desses dispositivos embarcados, desde pequenos sensores eletrônicos inteligentes até complexos dispositivos eletrônicos, programáveis e reconfiguráveis, nas áreas de monitoramento de grandezas elétricas, automação, proteção de sistemas elétricos, controle, telemetria, transmissão de dados, etc.

2.2 Simuladores *hardware-in-the-loop*

O papel e a importância dos "simuladores" na ciência em geral são facilmente verificados na literatura técnica, desde as áreas exatas da engenharia até os campos da medicina e das ciências econômicas, como pode ser visto no trabalho de (HARTMANN, 2005).

Para os cenários de engenharia, várias personalidades evidenciam em (USNSF, 2006) as virtudes e os benefícios de plataformas de simulação. Desde os antigos simuladores analógicos, até os simuladores digitais contemporâneos, todos são ferramentas valiosas nas áreas de estudo, projeto, desenvolvimento, treinamento e ensino (GREGA, 1999).

Como mostrado em (PELLINI, 2005), uma categoria importante de simuladores baseados em *hardware* digital são aqueles que apresentam a capacidade de calcular seus modelos matemáticos, e apresentar seus resultados, na cadência temporal que um sistema verdadeiramente apresentaria durante seu funcionamento.

Esses chamados "Simuladores em tempo real" são indispensáveis nos ambientes de engenharia assistida por computador (em inglês, *Computer Aided Engineering* (CAE)), sendo utilizados para treinamento de usuários e sistemas, prototipagem virtual e ensaios em malha-fechada com dispositivos externos reais para projeto, desenvolvimento e comissionamento.

A essas aplicações em malha fechada dá-se o nome de Simulações *Hardware-in-the-Loop* (HiL)[4], onde um dispositivo real é conectado a um simulador digital que, por sua vez, emula uma determinada planta real por meio da execução de algoritmos e modelos matemáticos (PELLINI, 2005). É importante ressaltar que, da mesma forma como ocorre nos sistemas embarcados, nessas aplicações também é imprescindível a sintonia e o sincronismo entre o tempo simulado e o tempo real, além do papel das interfaces de comunicação entre o mundo exterior e o simulador (e vice-versa) para o intercâmbio das informações e sinais pertinentes.

[4] O termo HiL ou *hardware-in-the-loop* é utilizado em alguns momentos ao longo desse trabalho. Apesar de também ser amplamente utilizado na literatura técnica, deve-se atentar que o termo é muito pouco apropriado. O termo é muito vago a respeito de onde é empregado o *hardware* em uma dessas montagens em malha fechada. Ou seja, o quê esse hardware representa (a planta ou o controlador) ? Se há *software* envolvido (o que é usual) ? Entre outros problemas.

Existe um grande portfólio de produtos e tecnologias de simuladores disponíveis para Sistemas de Potência. Enquanto alguns simuladores são mais genéricos e podem ser empregados em várias áreas, outros são extremamente especializados, possuindo uma ampla variedade de modelos matemáticos já implementados.

2.3 Comparação entre os sistemas

A necessidade de comparação entre "sistemas embarcados em tempo real" e "simuladores para aplicações *hardware-in-the-loop*" baseia-se no fato de ambos possuírem muitas semelhanças em termos de *hardware* e de *software*. Sobretudo na área de *software*, será mostrado que ambos os sistemas compartilham as mesmas dificuldades para a criação, desenvolvimento e testes.

2.3.1 Diferenças

Existem várias diferenças notáveis entre esses sistemas, que são devidas aos requisitos ou restrições impostas por suas respectivas aplicações.

Para ilustrar tal fato, pode-se analisar um exemplo qualquer na área de Sistemas de Potência, como um dispositivo embarcado instalado dentro de um cubículo de uma subestação de distribuição de energia elétrica. Nesse caso, o dispositivo deverá apresentar as seguintes características:

- Tamanho físico reduzido e elevado grau de proteção do invólucro ou chassi;
- Grau elevado de proteção e imunidade contra interferências, intempéries e vibrações;
- Baixo consumo de energia e dissipação;
- Desempenho computacional e memória restritos;
- Baixa taxa de falhas e alta confiabilidade;
- Suporte para redundância e para recuperação de falhas, e;
- Operação autônoma, completamente desassistida.

No caso de PLC's e IED's, além das características acima, esses devem apresentar também: maior capacidade de processamento e armazenamento (para uma aplicação desenvolvida pelo usuário), maior quantidade, variedade de tipos, desempenho e robustez de interfaces de entrada, saída e comunicação, interface IHM no painel do instrumento,

ausência de partes móveis, possibilidade de uso contínuo e desassistido por vários anos, etc.

Por outro lado, nenhuma dessas características é estritamente necessária para a utilização de um simulador em tempo real HiL na área de Sistemas de Potência. Os simuladores apresentam as seguintes características:

- Não possuem restrições para seu tamanho físico;
- Não necessitam de isolamento galvânico elevado ou maior grau de imunidade;
- Devem ser bastante versáteis e programáveis, podendo ser aplicados a vários tipos de cenários;
- Podem apresentar consumos de energia elevados;
- Podem utilizar computadores mais poderosos;
- Podem apresentar partes móveis em seu interior (disco rígido, ventiladores, etc.);
- Podem apresentar interface IHM rica em recursos, e;
- Podem até mesmo falhar.

A falta de necessidade de confiabilidade e robustez advém do fato de que os simuladores normalmente são empregados como ferramentas de laboratório para testes e ensaios, ambientes que costumam contar com a supervisão de operadores e engenheiros. Em caso de quaisquer problemas, os testes podem ser interrompidos a qualquer momento e reiniciados.

2.3.2 Semelhanças

De uma forma geral, em ambos os sistemas há uma combinação de um *hardware* eletrônico microprocessado e um *software* de gerenciamento e execução de tarefas computacionais (NOERGAARD, 2005).

No caso de PLCs e IED's, um fato importante é que um simulador em tempo real pode ser implantado em um desses sistemas embarcados com uma certa facilidade, bastando a codificação dos algoritmos de interesse.

Um exemplo desse tipo de abordagem é mostrado em (RAJASHEKARA et al., 1990), por meio de um simulador de gerador síncrono implementado dentro de um PLC de alto desempenho.

Em outro caso, mostra-se que um simulador pode ser implantado simultaneamente aos algoritmos nativos de controle de uma planta externa em uma mesma plataforma de um PLC. Dessa forma, ao invés de se conectar o PLC a um simulador externo para testes (aplicação HiL), pode-se utilizar a capacidade computacional ociosa do próprio PLC para também simular internamente os algoritmos dos modelos da planta real. Durante a parametrização do PLC, pode-se habilitar ou não esse recurso interno de simulação para fazer uso de seus benefícios. Um exemplo do emprego de um PLC como simulador de sua própria planta é demonstrado em (SANTOS, 2006), resultando em uma montagem simples e prática para o comissionamento e os testes em laboratório e no campo. Entretanto, deve-se ressaltar que tal solução também é potencialmente perigosa, uma vez que, caso ocorra algum erro ou problema interno, o controlador pode passar a operar a planta simulada ao invés da planta real, deixando essa completamente fora de controle.

Da mesma forma, um dispositivo de simulação em tempo real pode ser utilizado como um PLC ou um IED, bastando somente ser programado para tal (caso sua estrutura de *software* permita a livre implementação de algoritmos). Assim, a estrutura de *hardware* e *software* de um simulador pode ser empregada, por exemplo, para criar um relé digital de proteção ou até um conjunto de relés de proteção, todos em um mesmo equipamento, para finalidades de desenvolvimento de novos algoritmos e estratégias de controle.

Deve-se ressaltar, entretanto, que a aplicação de sistemas embarcados, PLCs e IED's como simuladores, ou vice-versa, também pode ser inviável em alguns casos onde os requisitos da aplicação demandam algumas das características exclusivas de uma ou de outra arquitetura, como mostrado anteriormente.

De qualquer forma, principalmente devido às otimizações no consumo de energia e ao aumento da capacidade de processamento, além do surgimento de outras tecnologias de apoio (alta integração de circuitos, memórias não voláteis de estado sólido, barramentos compactos de alta velocidade, etc.), existe uma forte tendência de que as versatilidades e potencialidades presentes nos simuladores da atualidade possam ser encontradas dentro de qualquer IED ou PLC em poucos anos, ou ainda, de que os recursos de confiabilidade (mecanismos de redundância, eletrônicas de maior robustez, etc.) sejam incorporados ao *hardware* dos simuladores, criando uma única classe de dispositivos com alta capacidade de processamento e elevada robustez.

Nesse trabalho, foi dado enfoque aos sistemas microprocessados que possuem os seguintes requisitos operacionais na concepção de sua arquitetura:

I. Interação com o processo externo entre 2,0 [ms] a 20,0 [μs] [5].

[5] A razão pela escolha desses limites de tempo para a taxa de amostragem será explicada adiante nesse trabalho.

II. Percepção dos eventos externos;

III. Processamento das informações desses eventos;

IV. Resposta ao mundo exterior;

V. Operação em sintonia e sincronismo com o processo externo (mundo real);

VI. Possibilidade de mudanças em seus algoritmos e parâmetros operacionais.

Na realidade, os requisitos *I*, *II*, *III* e *IV* estão intimamente relacionados para que se possa garantir o requisito *V*. Por exemplo, para um sistema que deve interagir com sua aplicação a cada 100,0 [μs] em tempo real, se cada requisito acima for responsabilidade de uma tarefa de *software*, o *hardware* do dispositivo deve ser capaz de garantir que a execução de todas essas tarefas seja feita dentro desses exatos 100,0 [μs]. Se o dispositivo demorar demais para processar e responder ou, ao contrário, se responder de forma antecipada, a aplicação externa poderá ficar comprometida.

O desempenho do conjunto de *hardware* e *software* é fundamental para o funcionamento tanto de Sistemas Embarcados quanto de Simuladores em Tempo Real.

Segundo (GANSSLE, 1999), uma das características mais importantes desses sistemas que devem processar periodicamente um conjunto de tarefas em um determinado tempo é o seu *jitter* - ou a variação *local* do tempo de resposta do sistema ao longo do tempo - e o seu *wander*[6] - ou a variação *global* desse tempo de resposta.

Segundo (LAPLANTE, 1996), o desempenho requerido pela aplicação pode ser mais acirrado (baixo *jitter* e *wander*) nos sistemas denominados de tempo real determinístico (do inglês, *hard real-time*[7]), ou mais relaxado (maiores *jitter* e *wander*) nos sistemas de tempo real não-determinístico ou *quasi*-determinístico (do inglês, *soft real-time*).

O requisito *VI*, colocado anteriormente, denota a principal característica dos sistemas em tempo real tratados nesse trabalho: a categoria de dispositivos que não possui seus algoritmos e parâmetros operacionais rigidamente definidos, ou seja, aqueles sistemas utilizados para a prototipagem e os testes de algoritmos e estratégias de controle, e os sistemas que devem ser intrinsecamente programáveis, como os CLP's e IED's multifunção para a proteção de sistemas elétricos.

[6] Ou *drift*, como preferem alguns autores.

[7] O termo *hard-real time* também é traduzido, por alguns autores, como "tempo real crítico". Entretanto, deve-se notar que o aspecto crítico de tais sistemas é seu determinismo no acompanhamento do tempo real, e não que seu sistema seja necessariamente parte de um sistema "crítico".

2.4 Exemplos de aplicações em tempo real determinístico para Sistemas de Potência

Os Sistemas Embarcados e os Simuladores HiL que compartilham os mesmos requisitos operacionais citados anteriormente são chamados, nesse trabalho, de "Sistemas Computacionais em Tempo Real Determinístico", do inglês *Hard Real-Time Computing Systems* (HRTCS's). Alguns exemplos de utilização em Sistemas de Potência estão nas áreas de desenvolvimento, testes, comissionamento, depuração e treinamento de:

- Dispositivos de *hardware* eletrônicos, analógicos e digitais e seus *firmwares*, para:
  - Transdutores e sensores de grandezas elétricas;
  - Acionamento e comando de pontes de tiristores, inversores e conversores eletrônicos;
  - IED's para multimedidores, oscilógrafos, relés de proteção e dispositivos especiais;
  - CLP's;
- Controladores e algoritmos de controle para:
  - Reguladores de velocidade de turbinas hidráulicas e a vapor;
  - Reguladores de tensão de excitatrizes estáticas, rotativas e mistas;
  - Sistemas Digitais de Controle Distribuído (SDCD's), e de controle conjunto de unidades geradoras;
- Algoritmos de processamento digital de sinais para:
  - Medição de grandezas elétricas em multimedidores e relés de proteção;
  - Dispositivos de monitoramento de vida útil de equipamentos;
- Algoritmos e funções de proteção de relés;
- Algoritmos e funções de controle e automação.

Aplicações reais podem ser vistas nas referências e nos trabalhos de (PELLINI, 2005) e (SANTOS, 2006). Referências mais recentes de aplicações podem ser vistas em (ZHANG et al., 2007), (APOSTOLOPOULOS; KORRES, 2008), (PALLA; SRIVASTAVA; SCHULZ, 2007) e (ROITMAN; SOLLERO; OLIVEIRA, 2004).

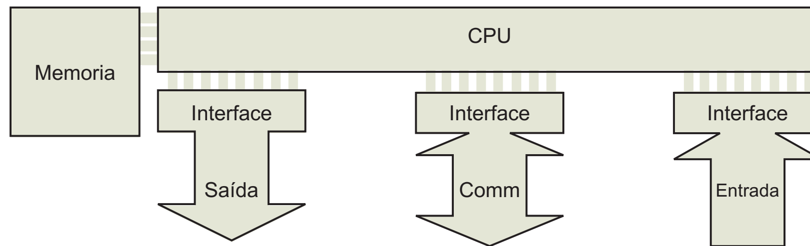

Figura 2: Topologia típica de *hardware*.

2.5 Detalhes da arquitetura de *hardware*

Pode-se observar que para as aplicações citadas, o *hardware* desses dispositivos é composto basicamente, por:

- Interfaces de entrada e de saída;
- Interfaces de comunicação;
- Unidades de processamento central (CPU);
- Memória.

Esse esquema pode ser visto na Fig. 2. Esses e outros componentes relevantes são apresentados com mais detalhes a seguir.

2.5.1 Interfaces de entrada e interfaces de saída

As interfaces de entrada[8] podem estar associadas a botões, chaves, potenciômetros e a uma miríade de sensores e de transdutores, que permitem a leitura e a conversão de grandezas físicas do processo externo (tensões, correntes, potências, forças, velocidades, rotação, etc.) para o universo digital do dispositivo.

Da mesma forma, as interfaces de saída[9] podem estar associadas a transdutores, relés, atuadores, válvulas, acionamentos, etc. que permitem converter as ordens e comandos digitais do dispositivo de volta às grandezas físicas do processo externo.

[8] Elementos sensores.
[9] Elementos atuadores.

Mais detalhes e exemplos de interface de entrada e saída podem ser obtidas de (PELLINI, 2005), (GREGA, 1999), (WU; FIGUEROA; MONTI, 2004) e (OGATA, 1998).

Os requisitos típicos de resolução e as taxas de aquisição e síntese de sinais analógicos e digitais variam em diferentes aplicações. Para os valores analógicos amostrados e sintetizados, tipicamente são utilizadas resoluções de 10 (controle eletrônico de motores), 12, 14 e 16 bits (IED's, multimedidores).

Alguns exemplos de frequências de aquisição e síntese são: controle eletrônico de motores com sinais PWM de acionamento, com taxas de 1,0 a 5,0 [kHz] (MOHAN; UNDELAND; ROBBINS, 1995); pulsos de chaveamento de tiristores em conversores de eletrônica de potência com taxas de 20,0 a 50,0 [kHz] (MOHAN; UNDELAND; ROBBINS, 1995); IED's oscilógrafos e relés digitais de proteção com taxas de aquisição e síntese de até 15,0 [kHz] (SENGER et al., 2008) e CLP's com taxas entre 1,0 e 20,0 [kHz] (SANTOS, 2006).

Com o advento de CPU's mais modernas e circuitos dedicados para o processamento digital de sinais (*Digital Signal Processors* (DSP's) e *Application Specific Integrated Circuits* (ASIC's)), existe uma tendência de se utilizar conversores A/D (analógico-digital) e D/A (digital-analógico) de maior resolução (com 18, 20 e 24 bits), além de sinais digitais amostrados e sintetizados com maiores frequências.

2.5.2 Interfaces de comunicação

Diferente do que é tradicionalmente exposto em trabalhos na área de automação e controle (OGATA, 1998), nessa obra foi criada uma categoria específica para as interfaces de comunicação. Entretanto, convém ressaltar que essas também são essencialmente interfaces de entrada e saída de dados, mas que possuem, atualmente, outros atributos associados e devem ser observados com mais atenção.

No passado, tais interfaces desempenhavam papéis laterais à aplicação, como nos primeiros Sistemas de Supervisão e Controle (SSC's) da década de 80, servindo apenas para a sinalização de estados e acionamentos de forma não-determinística, ou seja, a eventual indisponibilidade do canal de comunicação ou das informações, acarretava apenas na impossibilidade de se determinar um estado ou de se comandar um determinado equipamento à distância, sem o prejuízo das funções de automação locais ou quaisquer outros impactos na segurança ou na confiabilidade da aplicação.

Atualmente, as interfaces de comunicação desempenham um papel muito mais importante, em função da popularização dos esquemas de computação paralela distribuída e dos esquemas de automação e controle distribuídos em chão de fábrica e nas subestações. A arquitetura de *hardware* dessas interfaces passou por uma grande revolução nas últimas décadas, principalmente com respeito à velocidade de comunicação, robustez e compri-

mento de enlace físico. Tais interfaces evoluíram da década de 80, de simples barramentos seriais, ponto-a-ponto, com velocidade entre 300 a 1200 [bps] e comprimento de algumas dezenas de metros, às atuais redes de comunicação LAN/WAN, utilizando métodos de acesso do tipo CSMA/CD, velocidades de mais de 1,0 [Gbps] e avançados protocolos de comunicação.

Atualmente, nos cenários de computação paralela distribuída, um determinado algoritmo é dividido em vários processos computacionais, espalhados para serem executados em diferentes computadores, conectados entre si por redes de comunicação. O transporte dos dados entre os processos é feito inteiramente através dessa rede, cuja taxa de transferência afeta o desempenho global do sistema distribuído.

Nos cenários de automação e controle distribuídos, muitas das informações críticas, que antigamente eram transmitidas por meio de canais analógicos e digitais dedicados, hoje são agregadas em pacotes de dados e enviadas a um ou vários dispositivos, de forma prioritária[10], através de redes de comunicação.

Esse é o caso de vários protocolos de chão de fábrica, tais como: ProfiBus, DeviceNet e *Controller Area Network* (CAN), além do protocolo de mensagens rápidas do tipo *Generic Object Oriented Substation Event* (GOOSE), da norma (IEC61850, 2003) para automação de subestações, e do protocolo de sincronismo de alta precisão (IEEE1588, 2008) para instrumentação, medição e sistemas de controle.

As interfaces de comunicação hoje devem ser robustas, confiáveis, possuir esquemas de redundância e, principalmente, apresentar baixa latência e alto determinismo no envio das informações (IEEE1588, 2008). Esses fatos trazem implicações sérias no desenvolvimento do *software*, como será visto adiante.

2.5.3 Unidades de processamento central

As *Central Processing Units* (CPU's) possuem um papel fundamental para o desempenho esperado de processamento em tempo real requerido por uma aplicação. Dado um conjunto de tarefas, é papel do processador executá-las em um determinado intervalo de tempo, determinado por sua capacidade de execução de instruções.

A capacidade de processamento de uma plataforma digital, em geral, não pode ser estimada simplesmente em função da velocidade do seu relógio (*clock*) interno (MACNEIL, 2004). Isso se deve ao fato de que a capacidade de processamento depende essencialmente da arquitetura da plataforma, do tipo de instrução executada e do tipo de dado envolvido.

Em dispositivos de arquitetura mais simples, como microcontroladores, algumas ope-

[10]Com requisitos de tempo muito bem definidos.

rações duram mais de um ciclo de *clock* para serem executadas e são feitas apenas com dados elementares, de 8 [*bits*]. Por outro lado, os dispositivos digitais modernos possuem muitos recursos particulares em suas arquiteturas que permitem um grande aumento na capacidade de execução de instruções por ciclo de *clock*, além da operação com dados mais complexos, de 16, 32, 64, 128 [bits] ou mais.

Esse é o caso de alguns processadores de última geração, DSP's e ASIC's, que fazem uso de técnicas de processamento paralelo (múltiplos núcleos de processamento, múltiplas unidades funcionais, etc.), processamento vetorial (instruções do tipo *Single Instruction - Multiple Data* (SIMD)), *pipelines*, execução especulativa, decodificação de múltiplas instruções por ciclo, memórias cache, etc.

No caso específico de processamento paralelo simétrico ou *Symmetric Multiprocessing* (SMP), como ocorre nas arquiteturas mais contemporâneas "x86" da (INTEL, 2006) e (AMD, 2006), múltiplos núcleos processadores idênticos são conectados a um barramento local comum e compartilhado, permitindo a divisão de tarefas entre as unidades de processamento e um aumento do desempenho global.

Outro cenário importante são os atualmente populares ASIC's, dispositivos flexíveis, principalmente de lógica digital, com arquitetura "customizável". Esses dispositivos são criados a partir de *Programmable Logic Device*s (PLDs), tais como *Field Programmable Gate Array*s (FPGAs) e *Complex Programmable Logic Device*s (CPLDs), que possuem vários recursos de *hardware* digital embutidos em seus encapsulamentos, tais como portas lógicas, *flip-flops*, somadores, multiplicadores, multiplexadores, etc., que podem ser interconectados conforme o desejo do projetista.

A "programação" de um ASIC é feita através de uma linguagem que descreve o comportamento desejado do circuito integrado quando esse é submetido a determinadas excitações externas. Essa linguagem é "compilada" através de um processo denominado "síntese comportamental de alto nível". Além de originar verdadeiras arquiteturas de processadores, especializados no processamento de determinados fluxos de dados em alta velocidade, esses circuitos integrados são interessantes em aplicações embarcadas e simuladores pois propiciam ao *hardware* um certo grau de flexibilidade, próximo daquele do *software*, podendo ser modificado conforme a necessidade da aplicação.

Vale ressaltar que, para que seja possível utilizar todos os recursos dessas CPU's, DSP's e ASIC's, o *software* deve ser especialmente adaptado, como será comentado adiante.

A unidade *Instructions Per Second* (IPS) e seus múltiplos podem ser utilizadas para se medir a capacidade média do processamento de instruções de uma plataforma computacional. Entretanto, como comentado por (MACNEIL, 2004), esse número só pode ser

utilizado como um índice de comparação de desempenho entre dispositivos de arquiteturas semelhantes, e representa o desempenho médio de execução de todos os tipos de instruções do processador.

Em aplicações matemáticas intensivas em ponto flutuante, como aquelas executadas em um simulador, é mais interessante referenciar o desempenho computacional de uma plataforma através da unidade *Floating Point Operations per Second* (FLOPS), por ser mais representativa do tipo de instrução e dos dados utilizados nos cálculos.

A escolha das arquiteturas e das unidades de processamento nos dispositivos HRTCS é feita com base nos requisitos impostos pela aplicação, tais como: número e tipos de interfaces, quantidade de pinos, capacidade de memória, quantidade de interrupções, ambiente de desenvolvimento (linguagens, bibliotecas), desempenho e custo.

É interessante frisar que o custo não é simplesmente o custo unitário de uma determinada CPU, mas sim, todo o conjunto do custo da CPU e das ferramentas de desenvolvimento, depuração e do custo em homem-hora para a engenharia da aplicação e do produto.

A relação entre o desempenho e o custo é uma das mais relevantes no caso dos sistemas reparametrizáveis, onde deve ser escolhida aquela CPU que possui o desempenho mais adequado pelo menor custo. Segundo (BALL, 1996), mais especificamente nos dispositivos embarcados, onde existe também uma preocupação com o consumo energético e o espaço físico, existe a necessidade de se considerar, ainda, a potência dissipada pela CPU nessa equação, além de outros fatores, como o tempo de vida do projeto.

2.5.4 Memória

A memória desempenha um papel fundamental no funcionamento de um dispositivo HRTCS. Ela é responsável por armazenar as instruções e os dados não-voláteis e voláteis dos processos, na forma de *Read Only Memory*s (ROMs) do tipo EEPROM, Flash, etc. e *Random Access Memory*s (RAMs), do tipo SRAM, DRAM, etc.

As interfaces elétricas, a velocidade e o modo de acesso a essas memórias (barramentos) afetam o desempenho global do sistema, já que o funcionamento da CPU depende da obtenção dos dados e das instruções armazenados nessas memórias.

Algumas arquiteturas modernas de CPU's possuem memórias de grande largura de barramento (16, 32, 64 [bits] ou mais), e topologias de memória distribuídas com diferentes velocidades de acesso, tais como registradores e memórias cache (de alta velocidade, internas ao processador) e memórias em barramentos locais (de menor velocidade, externas ao processador).

A escolha da topologia correta das memórias também é importante. Em arquiteturas SMP, onde dois ou mais núcleos de processadores idênticos são posicionados em um mesmo barramento local, uma topologia desfavorável das memórias compartilhadas entre os núcleos de processamento pode criar "gargalos" na transferência de dados, ocasionando uma queda ao invés de um aumento no desempenho.

2.5.5 Outros subsistemas de apoio

Todo *hardware* necessita de outros circuitos eletrônicos e digitais de apoio para o seu funcionamento. Nesse trabalho, esses componentes não são comentados com mais detalhes, contudo deve-se atentar que eles são fundamentais para o correto funcionamento e um bom desempenho de um dispositivo HRTCS.

Alguns exemplos desses circuitos são:

- Fonte de alimentação;
- Circuitos de tratamento de sinais analógicos e digitais (para acondicionamento dos sinais, isolamento galvânico, filtragem *anti-aliasing*, ajustes de ganho e *offset*, etc.);
- Circuitos de conversão analógico-digital e digital-analógico;
- Circuitos da camada física de interfaces de comunicação;
- Circuitos de relógio (*clock*) e PLL's para sintonia e sincronização;
- Amplificadores e circuitos de acionamento de potência, etc.

Mais detalhes podem ser obtidos nas obras de (BALL, 1996), (BALL, 2001) (GANSSLE, 1999), (CATSOULIS, 2005) e (MOHAN; UNDELAND; ROBBINS, 1995).

2.6 Detalhes da arquitetura de *software*

A topologia básica de *software* desses sistemas é composta por:

- Sub-rotinas controladoras do *hardware* (*Drivers*);
- Processos e tarefas, e;
- Sistema operacional ou executivo.

Esse esquema pode ser visto na Fig. 3 e será mostrado com mais detalhes a seguir.

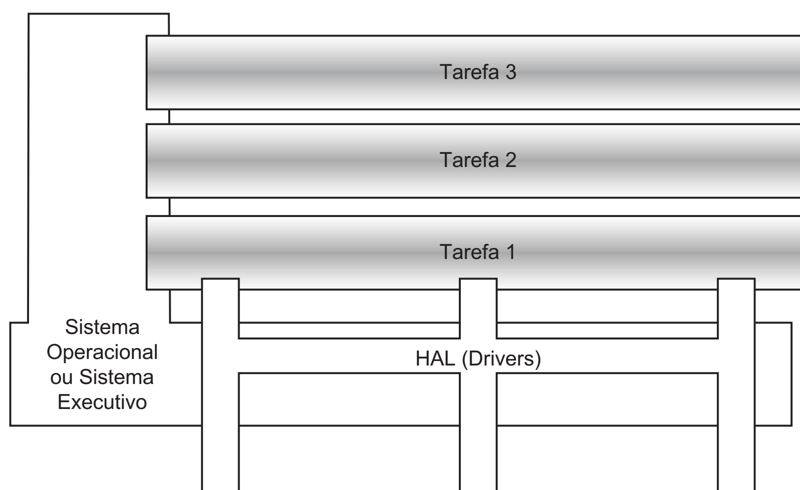

Figura 3: Topologia típica de *software*.

2.6.1 Camada de abstração de *hardware*

As sub-rotinas controladoras do *hardware*, também chamadas de *drivers*, conformam a chamada "Camada de abstração de *hardware*" (do inglês, *Hardware Abstraction Layer* (HAL)).

Esses drivers de dispositivos são responsáveis por permitir o acesso do sistema operacional/executivo e suas tarefas aos demais recursos disponíveis no *hardware*. Tais rotinas são codificadas, parte em linguagens de alto nível (ANSI C), e parte em linguagem de máquina (Assembly), por razões de desempenho.

A utilização dessa camada de rotinas pré-programadas também tem como objetivo a padronização de interfaces com os recursos do *hardware*, bem como esconder detalhes intrínsecos das arquiteturas ao programador, para favorecer a portabilidade de uma determinada aplicação a outras plataformas sem que haja a recodificação de todo o *software*.

2.6.2 Processos e tarefas

Segundo (TANENBAUM, 2007), em sistemas computacionais um processo é conjunto independente de instruções e dados que, quando em execução, desempenha um determinado comportamento algorítmico desejado. Uma tarefa, por sua vez, é um conjunto menor de instruções e dados, que depende e/ou compartilha dados e instruções com outras tarefas e processos em execução.

Os HRTCS's possuem uma estrutura básica composta por um processo básico que contém um conjunto de tarefas de alta prioridade, codificadas em uma dada linguagem

de programação e executadas por um sistema operacional ou executivo.

As tarefas principais são:

- A varredura dos sinais das interfaces de entrada e comunicação;
- A execução dos algoritmos matemáticos da aplicação;
- A síntese dos resultados e sinais em interfaces de saída e de comunicação.

Como mostrado em (PELLINI, 2005), tipicamente essas tarefas são executadas ciclicamente (em *loop*) dentro de um determinado processo. A taxa de execução dessas tarefas e a relação entre o tempo interno decorrido e o tempo real devem ser gerenciados por algum mecanismo de sincronização.

Outras tarefas de menor prioridade também podem ser executadas no tempo computacional ocioso, tais como: serviços de protocolos de comunicação não prioritários, tarefas de monitoramento de *hardware*, interface com usuário, etc.

Deve-se ressaltar, entretanto, que em sistemas de processamento paralelo em tempo real e sistemas de automação e controle modernos, que utilizam arquiteturas distribuídas e modelos de comunicação de dados entre dispositivos, a tarefa de alguns protocolos de comunicação é da mesma prioridade dos algoritmos da aplicação, como o processamento de mensagens GOOSE na norma (IEC61850, 2003), ou a interpretação de mensagens em barramentos de processo, como aquelas presentes em barramentos CAN (ISO11898, 2003), FieldBus Foundation (FIELDBUS, 2010) e GOOSE-SV da norma (IEC61850, 2003).

2.6.3 Sistemas operacionais e executivos

O mecanismo de gerência das tarefas prioritárias e não-prioritárias é realizado por um Sistema Operacional (SO) ou um Sistema Executivo (SE).

2.6.3.1 Características dos sistemas operacionais

Segundo (TANENBAUM, 2007), um SO consiste em uma estrutura de *software* responsável pelo gerenciamento dos recursos do sistema para cada um dos processos em execução no computador.

O SO é formado por uma rotina principal, denominada de núcleo (do inglês, *kernel*), responsável por executar os processos e as rotinas de chamada do sistema (do inglês, *System calls*), permitindo o acesso dos processos aos dispositivos de *hardware*.

No núcleo do SO é executado o processo mais importante, o chamado escalonador (do inglês, *scheduler*), que permite o multiprocessamento, ou seja, a divisão do tempo de computação entre várias linhas de execução. O escalonador possui uma fila de processos ativos, agendados para execução. Seu papel é gerenciar continuamente qual processo será executado em qual CPU e por quanto tempo. Tipicamente, o escalonador permanece atribuindo uma determinada "quanta" de tempo[11] a cada processo, para que esse tenha o direito de uso da CPU naquele momento. Caso essa "quanta" acabe, o escalonador realiza a mudança de contexto ou preempção na CPU, interrompendo o processo atual (preservando seus estados e informações) e passando o direito de uso da CPU ao próximo da lista.

O resultado é uma coleção de processos, que parecem ser executados de forma concorrente, como se existissem diversas CPU's operando de forma simultânea. Em SOs modernos, os recursos de multiprocessamento podem ser utilizados dentro de um mesmo processo, quando se divide a carga computacional em diversas tarefas (*threads*) menores que também são escalonadas pelo *kernel* junto com as *threads* dos demais processos, nos chamados SOs multitarefa, ou *multithreading*.

Deve-se atentar que a programação num ambiente de SO multiprocessado e multitarefa traz paradigmas importantes para o projetista de *software* a respeito da concorrência, e da disputa de recursos entre processos. Tais disputas podem levar uma aplicação desenvolvida a situações de longa espera, enquanto aguarda a liberação de um determinado recurso (*resource starvation*), ou até à parada total, quando o processo aguarda por recursos que nunca são liberados pelos demais processos (*deadlock*) (TANENBAUM, 2007).

Do ponto de vista do tempo verdadeiramente decorrido, um processo ou uma tarefa são executados de forma descontínua desde a sua criação até o seu término. Essa descontinuidade resulta em um aumento entre o tempo esperado de execução e o tempo utilizado para processamento, podendo ocasionar atrasos e latências no tempo de resposta esperados(SILBERSCHATZ; GALVIN; GAGNE, 2008).

Em aplicações de tempo real, onde se deseja um certo determinismo no tempo de resposta de algumas tarefas, o tipo de política de agendamento e priorização de acesso dos processos à CPU é relevante para o desempenho global do sistema. Por exemplo, em um SO com um escalonador preemptivo convencional, com "quanta" de tempo mínima de 10,0 [ms], em um cenário com poucos processos concorrentes e muito tempo ocioso de computação, os resultados de tempo de resposta podem até ser razoáveis. Entretanto, no caso de existir uma grande quantidade de processos e pouco tempo ocioso de computação, possivelmente o tempo de resposta médio será maior que 10,0 [ms] e/ou apresentará fortes

[11]Quanta de tempo, fatia de tempo ou *time slice* é o nome tipicamente dado ao período de tempo dado pelo escalonador para execução de um determinado processo.

variações locais (*jitter*).

Para aplicações em tempo real rígidas, costuma-se utilizar uma categoria especial de SOs, onde o escalonador dá prioridade máxima e fixa aos processos que possuem compromissos estritos de tempo de resposta. Essa categoria é denominada Real-Time Operating Systems (RTOS's), como descrito por (EMERSON; NEEMA; SZTIPANOVITS, 2006), e é representada por um vasto portfólio de produtos comerciais e de domínio público, para diversas arquiteturas e plataformas de *hardware*.

De qualquer forma, um RTOS moderno possui uma arquitetura de *software* muito complexa, que inclui: suporte completo para multiprocessamento, multitarefa, vários tipos de escalonadores encadeados, mecanismos de inversão de prioridades, suporte a memória virtual, etc. Essa complexidade reflete em seus requisitos mínimos de memória e na capacidade de processamento, que ultrapassam facilmente os recursos disponíveis em muitos dispositivos embarcados microcontrolados. Para esses cenários mais restritos existe uma vertente desses RTOS's que possuem, ao invés de um núcleo complexo, uma estrutura mais enxuta denominada *microkernel*.

2.6.3.2 Características dos sistemas executivos

Os SE, por sua vez, também são ambientes de execução de aplicações multitarefa em tempo real, mas que possuem apenas um processo ativo, codificado de forma monolítica[12] com o restante do *firmware* da aplicação.

A estrutura de um SE é muito mais enxuta, determinística e de melhor desempenho geral do que aquela dos RTOS's, incluindo os que possuem *microkernels*. Isso se deve ao fato que não há necessidade de gerenciadores complexos de recursos ou escalonadores de tarefa, uma vez que todo o agendamento e estabelecimento de prioridades é feito no momento da programação das tarefas da aplicação, de modo estático. Os SE são utilizados em aplicações que não necessitam de muita flexibilidade ou constante reprogramação, ou em sistemas que possuem grandes restrições de memória ou recursos de *hardware* insuficientes para a implementação de um *microkernel*, como é o caso de pequenos microcontroladores e DSP's. Seu desempenho em tempo real possui baixo *jitter* e latência se comparados aos melhores RTOS's.

[12]Codificado como um único programa, com todos os algoritmos construídos dentro do mesmo código-fonte, sem que possam ser lançados ou criados outros processos de forma programática.

2.7 Detalhes das tarefas de *software* para Sistemas de Potência

Os simuladores em tempo real e os sistemas embarcados possuem várias tarefas computacionais, relacionadas à aplicação em Sistemas de Potência, com um grau de complexidade ímpar, como mostrado a seguir.

2.7.1 Tarefas em simuladores em tempo real HiL

De uma forma geral, os cenários de Sistemas de Potência são interessantes do ponto de vista multidisciplinar, pois envolvem diversas áreas de conhecimento, desde circuitos elétricos, máquinas elétricas rotativas, sistemas de transmissão e distribuição, controle, processamento digital de sinais e até mesmo computação (*hardware* e *software*[13]).

Por essa razão, em geral, os problemas de Sistemas de Potência podem ser descritos e resolvidos de várias formas, como conjuntos de um ou mais métodos, tais como: regras algoritmicas e equações algébricas, matrizes de sistemas lineares, sistemas de equações diferenciais, elementos finitos, processamento digital de sinais, etc. Alguns exemplos dessas soluções podem ser vistos nos algoritmos de simulação apresentados em (WATSON; ARRIGALA, 2003), nos sistemas de equações diferenciais no espaço de estados mostrados em (KUNDUR, 1994), nos algoritmos de simulação em elementos finitos mostrados em (DEMENKO, 1996), e nas equações diferenciais mostradas por (ZANETTA JR., 2003).

Particularmente, nos simuladores em tempo real para o estudo HiL nas áreas de geração, transmissão e distribuição de energia elétrica, é de particular interesse a análise do comportamento transitório (para baixas e altas frequências) dos Sistemas de Potência.

Os dispositivos tipicamente modelados nesses estudos compreendem os geradores elétricos, linhas de transmissão, cargas, eletrônicas de potência, elementos de controle, dispositivos de proteção, dispositivos de automação, entre outros.

Nesses casos, os problemas são descritos tipicamente como conjuntos de equações diferenciais ordinárias (EDO) ou funções de transferência, como as mostradas em (ADKINS, 1964) e (KUNDUR, 1994), além de equações algébricas ou algoritmos, como aqueles mostrados em (MOHAN; UNDELAND; ROBBINS, 1995). Tais equações e regras representam matematicamente a dinâmica e o comportamento de vários dispositivos do sistema quando esses são submetidos a perturbações. A solução dessas equações permite verificar esse comportamento ao longo do tempo.

[13]Quer o engenheiro goste ou não.

2.7.1.1 Motor de simulação do cenário

Para se obter a solução desse conjunto de equações, é necessário um 'motor' de simulação (do termo usualmente utilizado em inglês *simulation engine*), onde:

- As equações podem ser descritas;
- Os parâmetros da simulação podem ser ajustados;
- Os valores iniciais podem ser calculados e/ou ajustados;
- As equações podem ser resolvidas ao longo do tempo para se obter a evolução das grandezas e dos estados do sistema mediante as perturbações implicadas;
- Os resultados podem ser registrados e armazenados para análise e interpretação.

A descrição das equações dos modelos matemáticos pode ser realizada graficamente, através de esquemas e diagramas elétricos, diagramas de blocos funcionais; ou textualmente, através de uma linguagem de programação específica ou metalinguagens, com o uso ou não de outros modelos pré-programados primitivos, como bibliotecas de componentes e funções.

Os parâmetros da simulação a serem ajustados são aqueles que adaptam as equações dos modelos para que essas representem um determinado cenário real. Alguns exemplos desses parâmetros são as constantes de tempo e coeficientes de um determinado gerador ou de uma parte da rede elétrica, parâmetros de linhas, etc. Além desses parâmetros inerentes aos modelos, ainda é necessário informar ao *engine* de simulação dados como o tempo de simulação, tolerâncias e erros máximos de convergência, parâmetros de ajuste dos algoritmos de integração numérica, variáveis de interesse para registro, etc.

Tais sistemas dinâmicos, representados por seus modelos matemáticos, fazem parte de uma categoria denominada 'Problemas de Valores Iniciais', onde todas as EDO's devem ter seus estados pré-ajustados a valores iniciais conhecidos. Em Sistemas de Potência esses valores podem ser iguais a 'zero' (condições nulas) ou de repouso [14], ou então, obtidos a partir de pré-análises do sistema, tais como aquelas realizadas nas redes elétricas em regime permanente senoidal com o uso, por exemplo, de algoritmos de cálculo de fluxo de potência.

As equações diferenciais resultantes, devido ao seus tipos e a suas complexidades, normalmente não possuem soluções analíticas. Para a solução do sistema e obtenção do comportamento dos estados ao longo do tempo, são utilizados 'resolvedores' (do inglês,

[14] No caso de Sistemas de Potência submetidos a excitações senoidais, a palavra "repouso" está também associada aos estados do sistema quando esse se encontra em regime permanente senoidal.

solvers) de EDO's que empregam métodos numéricos sofisticados para controlar erros de integração e truncamento.

2.7.1.2 Complexidade do cenário

Em aplicações em tempo real existe um compromisso de desempenho do motor de simulação para que os resultados sejam calculados no tempo correto e com a precisão desejada. Entretanto, tal desempenho pode ser comprometido pela complexidade do sistema de equações a ser resolvido. Em Sistemas de Potência, essa complexidade está relacionada principalmente à presença de:

- Grande número de entidades simuladas;
- Não-linearidades;
- Dinâmicas de diversas naturezas, e;
- Sistemas com elevados índices de rigidez (*stiffness*).

A quantidade de dispositivos simulados nos estudos resulta em um aumento proporcional na complexidade do sistema de equações. Por exemplo, em (PELLINI, 2005) é mostrado um caso com um único gerador hidroelétrico, com seus reguladores de velocidade, tensão, turbina, sistema de excitação e rede elétrica que resultam em um sistema de equações com 14 estados e cerca de 30 variáveis de interesse. Com esses mesmos modelos, um simulador multimáquina representando n geradores simultaneamente não teria menos que $14.n$ estados.

Além disso, na aplicação de modelos matemáticos mais elaborados, tais equações podem aumentar consideravelmente de ordem e passar também a incorporar termos não lineares, como por exemplo, a característica de saturação magnética nos geradores, limitadores e bandas de histerese em controladores, parâmetros 'variáveis', entre outros.

Devido à multidisciplinaridade dos Sistemas de Potência, há uma grande diversidade de dinâmicas envolvidas, e de naturezas distintas [(KUNDUR, 1994) e (WATSON; ARRIGALA, 2003)]. Por essa razão, o sistema resultante para estudo será composto por uma ampla variedade de equações e algoritmos correlacionados, das dinâmicas elétricas, eletromecânicas, mecânicas, hidráulicas, controles, etc.

Da mesma forma, essa variedade de dinâmicas resulta em uma grande dispersão nos valores das constantes de tempo envolvidas nas equações diferenciais. Em (WATSON; ARRIGALA, 2003), por exemplo, são enumerados os possíveis cenários de constantes de tempo

em simulações de Sistemas de Potência, mostrando casos com constantes da ordem de centenas de nanosegundos (para a representação de transitórios provenientes de distúrbios atmosféricos) até dezenas de milhares de segundos (para a representação de transitórios lentos de controle de vazão hídrica de reservatórios). O chamado índice ou grau de rigidez (*stiffness*) mede essa razão entre a maior e a menor constante de tempo presentes em um mesmo sistema de equações diferenciais.

Apesar de serem muito incomuns estudos de simulação em tempo real envolvendo, simultaneamente, transitórios com comportamentos temporais tão distintos quanto os extremos mostrados por (WATSON; ARRIGALA, 2003), simulações de Sistemas de Potência resultam facilmente em um grau de rigidez da ordem de 1.000 a 1.000.000, ou seja, constantes de tempo distintas separadas de três a seis décadas no tempo. Exemplos podem ser vistos em (PELLINI, 2005) e em (ANDRADE, 2007), em simulações para análise e ajuste de estabilizadores de sistemas de potência (do inglês, PSS's), e reguladores automáticos de tensão (do inglês, AVR's). Nesses e em outros casos, a rigidez do sistema pôde ser diminuída pela escolha de modelos coerentes com o objetivo do estudo de simulação. Tais modelos desprezam efeitos de alta frequência que seriam ocasionados por pequenas constantes de tempo, ou efeitos de longo período ocasionados por grandes constantes de tempo. Esses e outros cenários de simulação podem ser claramente classificados em um dos três tipos de estudos de transitórios:

- Transitórios eletromagnéticos, envolvendo constantes de tempo na faixa de 10 [μs] a 200 [μs];
- Transitórios eletromecânicos, envolvendo constantes de tempo na faixa de 200 [μs] a 1.0 [s];
- Transitórios de longa duração, com constantes de tempo acima de 1.0 [s].

Considerando os fatores colocados anteriormente, simulações em Sistemas de Potência tipicamente resultam em um grande sistema de equações diferenciais, acopladas, não lineares, a parâmetros variantes no tempo e com elevado grau de rigidez. Tais sistemas não possuem soluções analíticas e só podem ser resolvidos no domínio do tempo através de um processo iterativo de cálculo numérico [(KUNDUR, 1994) e (WATSON; ARRIGALA, 2003)].

2.7.1.3 '*Solver*' das equações algébricas e diferenciais

O algoritmo computacional responsável pelo processo iterativo de solução das equações no domínio do tempo é denominado em inglês, *solver*. [15] O *solver* resolve as equações al-

[15] A melhor tradução para o português do termo '*solver*' seria 'solucionador'.

gébricas e diferenciais considerando a grandeza 'tempo' como uma grandeza não-contínua, que avança em intervalos ou passos de tempo (do inglês, *time step*) discretos e, a *priori*, conhecidos.

A partir de um conjunto de condições iniciais dos estados, o *solver* realiza as seguintes operações em uma dada iteração:

1. Calcula os valores dos integrandos das EDO's no instante de tempo atual, utilizando equações algébricas e algoritmos que envolvem os valores anteriores dos estados e das entradas do sistema;
2. Calcula os novos valores dos estados no instante de tempo atual através de algum método de integração numérica;
3. Calcula os novos valores das variáveis de interesse (saídas) do sistema com base nos novos estados calculados e os valores das entradas do sistema;
4. Calcula qual será o próximo instante de tempo em que o *solver* deve ser novamente executado.

Existem vários tipos de *solvers* disponíveis na técnica. Cada um implementa diferentes técnicas de integração numérica, capazes de produzir resultados com maior ou menor desempenho computacional e maior ou menor precisão.

Os exemplos mais simples de *solvers* são aqueles baseados em passos de integração fixos e algoritmos de integração retangular ou trapezoidal. Por utilizarem passo fixo, o determinismo na manutenção do tempo real durante sua execução é excelente e bastante previsível.

Por outro lado, em sistemas de grande rigidez e com a presença de muitas não-linearidades, a precisão e a convergência dos modelos podem ser comprometidas caso o passo de integração não seja suficientemente pequeno. Para evitar problemas de instabilidade numérica, o passo de integração pode ser diminuído em alguns momentos. Entretanto, nessa ocasião serão necessárias mais iterações para cumprir o mesmo tempo de simulação e, caso não haja capacidade computacional, o sistema poderá se atrasar, comprometendo seu compromisso de execução em tempo real.

Solvers mais precisos envolvem complexas técnicas de integração numérica com passo variável e controle de erros, como os que utilizam o método de integração Runge-Kutta, o método *multi-step* Adams-Bashforth ou ainda, os mais recentes solvers algébrico-diferenciais da família *Differential Algebraic System Solver* (DASSL) desenvolvidos a partir do trabalho de (PETZOLD, 1982). Em geral, tais técnicas iniciam a solução das equações algébrico-diferenciais com um determinado passo de integração, calculando também em cada itera-

ção o erro acumulado no processo. Na ocorrência de uma descontinuidade, evento abrupto (como mudanças nos parâmetros das EDO‘s) ou fortes influências de não-linearidades, o erro acumulado máximo e a convergência numérica podem ser violadas. Nessa ocasião, o *solver* toma a decisão de voltar uma iteração no passado e refazer o processo de cálculo numérico, mas dessa vez com um passo de integração menor. Esse processo de diminuição do passo de tempo também é iterativo, com o *solver* reduzindo continuamente seu tamanho até que um limite inferior seja alcançado ou até que os erros voltem a ser aceitáveis. Caso as estimativas de erro permaneçam inferiores aos limites estabelecidos, o *solver* pode aumentar novamente o passo de integração gradualmente, até seu valor nominal, nas iterações subsequentes. Tal método é eficiente em termos de precisão e convergência, mas requer nitidamente maior capacidade computacional: para as rotinas mais elaboradas de cálculo numérico das integrais, para estimativa do erro local de cada estado e para cálculo do novo passo de tempo. Acima de tudo, o *solver* ocasiona um pico de processamento sempre que o erro acumulado ultrapassa os limites estabelecidos e uma nova tentativa de iteração é executada. Nessa ocasião, os resultados da iteração atual são perdidos e o sistema deve calculá-los novamente até que o erro esteja novamente aceitável.

A escolha por um ou outro *solver* é baseada na capacidade computacional da plataforma de *hardware* do simulador, nos requisitos de tempo de execução, na necessidade de maior ou menor grau de fidelidade nos resultados e da presença de não-linearidades nos modelos matemáticos (WATSON; ARRIGALA, 2003).

Por via de regra, em simulações *offline* de sistemas como os de potência, com grande complexidade, elevados índices de rigidez e não-linearidades é necessário o emprego de *solvers* mais sofisticados, de passo variável ou híbridos, para que o processo iterativo seja convergente e não resulte em instabilidades numéricas (ASTIC; BIHAIN; JEROSOLIMSKI, 1994). Entretanto, em simuladores em tempo real determinístico é comum a implementação dos *solvers* com algoritmos simples de integração trapezoidal (WATSON; ARRIGALA, 2003) ou esquemas híbridos como o mostrado por (DUFOUR; BELANGER, 2001) para obter de forma mais simples um comportamento determinístico no tempo de execução dos algoritmos e, consequentemente, a manutenção do desempenho em tempo real crítico.

2.7.2 Tarefas em sistemas eletrônicos embarcados

Da mesma forma que os simuladores, os sistemas embarcados em aplicações de Sistemas de Potência também abrangem diversas áreas de conhecimento, tais como: aquisição de dados, processamento digital de sinais, sistemas de controle discretos, lógica combinatória e sequencial para automação e algoritmos computacionais para aplicações especiais, etc.

Alguns exemplos podem ser vistos nos trabalhos de (SENGER et al., 2008) a respeito do desenvolvimento de algoritmos de proteção para um IED de proteção diferencial de linhas de transmissão, em (MUSSA, 2003) no desenvolvimento de um controle microprocessado para um conversor CA-CC, em (OLIVEIRA et al., 2007) no desenvolvimento de algoritmos para um sistema de recuperação dinâmica de tensão em DSP e em (ASSIS; REZEK; SILVA, 1997) com um dispositivo de controle digital para uma máquina CC.

Pode-se notar que a maioria desses cenários envolve o desenvolvimento e aplicação de algum algoritmo ou técnica, para uso em uma plataforma embarcada microprocessada em tempo real.

2.7.2.1 Processamento digital de sinais

Nas aplicações embarcadas é imprescindível a aquisição de dados externos através de entradas analógicas e digitais e uso de conversores analógico-digitais, circuitos de condicionamento, etc. Em seguida, os dados digitalizados precisam ser tratados matematicamente através de técnicas de processamento digital de sinais para se extrair alguma informação relevante da grandeza externa, como seu valor médio, amplitude, fase, conteúdo harmônico, etc. O desenvolvimento desses algoritmos usualmente é feito em alguma linguagem de simulação *offline* em um computador de desenvolvimento para, após testes, ser então implementada na aplicação embarcada.

2.7.2.2 Algoritmos de controle e automação tradicionais

Nas aplicações onde o sistema embarcado atua sobre uma grandeza do mundo real, através de uma saída digital ou analógica, a decisão a respeito dessa atuação é normalmente obtida do resultado do processamento de um algoritmo computacional de controle e/ou automação, de forma muito semelhante a aquela executada pelos *solvers* nos simuladores em tempo real.

No domínio do tempo discreto, as entradas e estados são processados pelo sistema embarcado através de um conjunto de equações algébricas e diferenciais e lógicas booleanas combinatórias e sequenciais. O resultado desse processamento são as saídas para o controle e a automação do processo externo. Nessas equações são utilizados elementos de controle clássico, tais como integradores, ganhos, somadores, filtros digitais, compensadores, controladores Proporcional-Integral-Derivativo (PID), etc. para tratamento das informações numéricas digitalizadas. Para tratamento de informações digitais são utilizados elementos básicos primitivos da lógica combinatória e sequencial booleana tais como portas lógicas E, OU, NOT, NAND, NOR, flip-flops, contadores, máquinas de estado, geradores de pulsos, etc.

2.7.2.3 Algoritmos específicos

Outros tipos de algoritmos também costumam ser desenvolvidos para utilização em sistemas eletrônicos embarcados microprocessados, PLCs e IED's, tais como: algoritmos com estratégias avançadas de controle como as descritas em (OLIVEIRA et al., 2007), algoritmos de inteligência artificial como o trabalho de (SANTOS, 2004) sobre algoritmos de proteção de linhas de transmissão utilizando redes neurais. Normalmente o desenvolvimento desses algoritmos é feito em uma plataforma bem diferente daquela que será utilizado no dispositivo embarcado final.

3 Estado da arte de desenvolvimento de sistemas computacionais em tempo real determinístico

A engenharia de sistemas computacionais é estudada há mais de 40 anos, tanto na área de *hardware* - com a engenharia dos sistemas eletrônicos digitais - como na área de algoritmos, com as ciências da computação e a engenharia de *software*.

Nesse capítulo são mostradas as dificuldades e os problemas encontrados no desenvolvimento dos HRTCS, principalmente na área de *software* voltado para essas aplicações em sistemas de potência.

São detalhadas algumas das ferramentas e metodologias disponíveis atualmente para tentar solucionar esses impasses, otimizar o processo e torná-lo mais simples, racional e previsível. Será mostrado que a adoção de um conjunto dessas ferramentas é a melhor solução.

3.1 Dificuldades para criação de uma aplicação

Os requisitos acirrados de desempenho e a multidisciplinaridade das soluções de HRTCS's para a área de sistemas de potência ocasionam maiores dificuldades para o *design*, implementação e testes desses equipamentos, principalmente no projeto do *software*.

Como em outras área da engenharia, o projeto geral de um HRTCS envolve as seguintes etapas:

- Especificação inicial, onde são definidos o escopo e os objetivos do projeto;
- *Design*, onde é realizado o projeto dos componentes e das partes do sistema;
- Execução, quando são implementadas as soluções projetadas;
- Verificação[1], quando são testadas e verificadas as premissas e especificações iniciais;
- Entrega.

[1]Deve-se atentar que as etapas de teste de um projeto envolvem metodologias de verificação e validação (V & V)que constituem um problema a parte para o desenvolvimento de um projeto.

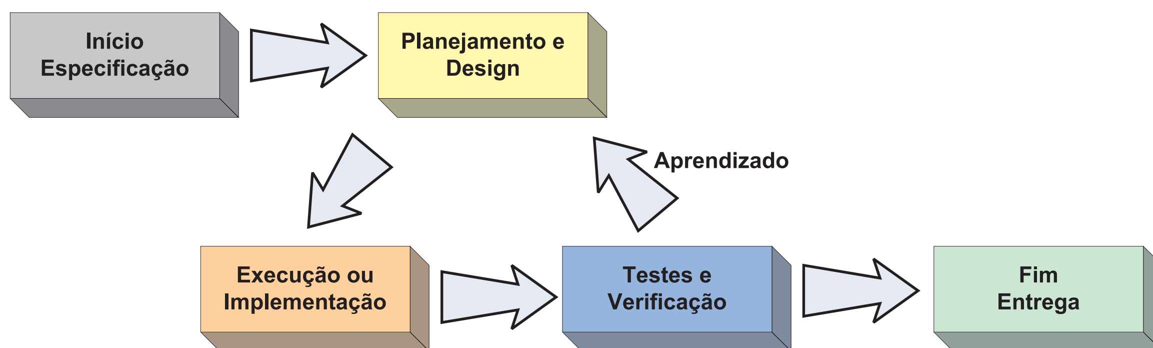

Figura 4: Etapas típicas de desenvolvimento de um projeto.

Esse processo é naturalmente iterativo, uma vez que na etapa de verificação podem ser encontrados problemas ou não-conformidades às especificações originais e, para se adequar novamente ao desejado, o projeto passa por algum ajuste ou redesenho. Esse comportamento é usual em qualquer projeto de engenharia e pode ser visto na Fig.4.

Mais especificamente nos projetos que envolvem pesquisa e desenvolvimento de novas soluções, a recorrência do ciclo *design*-implementação-teste possui várias iterações. Isso se deve ao fato que, nesses casos, são feitas inovações ou são utilizadas novas tecnologias, sem precedentes. A ausência de experiência em aplicações anteriores, faz com que o processo tenha um grau de incerteza maior e retrabalhos mais frequentes.

Na área dos dispositivos HRTCS, internamente, o projeto envolve dois sub-projetos de naturezas distintas, mas intimamente relacionadas, os já comentados: *hardware* e *software*. Nesse contexto surgem problemas que podem tornar o número de iterações de desenvolvimento explosivo, aumentando custos e tempo de mercado.

3.1.1 Problema do particionamento

Em um mundo ideal, seria interessante que após a etapa de *design*, ambos os sub-projetos pudessem ser conduzidos de forma independente e somente integrados ao final. Contudo, conforme mostrado por (NOERGAARD, 2005) e (CATSOULIS, 2005), nota-se que o desenvolvimento do *hardware* depende das escolhas (corretas ou não) feitas para o *software* e vice-versa.

Nesses projetos existe um problema que surge logo na etapa de *design* conhecido como "particionamento" (do inglês, *partitioning*) segundo (GANSSLE, 1999), que consiste na indecisão ou falta de visão clara a respeito do que será implementado em *hardware* e do que será implementado em *software* das funcionalidades previstas. Essa definição é fundamental, pois permite determinar de forma objetiva quais são as responsabilidades de um e outro mundo, além de estabelecer quais são as plataformas e as ferramentas de

desenvolvimento a serem adotadas no projeto.

Contudo, conforme mostrado por (LóPEZ-VALLEJO; LóPEZ, 2003) esse problema não possui uma solução determinística. O que é feito tradicionalmente são decisões pragmáticas, baseadas na experiência dos projetistas ou recomendações de fabricantes. Entretanto elas são sujeitas a falhas, que ocasionam retrabalhos e mais iterações de projeto.

Atualmente, com a popularização dos *hardware's* customizáveis [2], tais como ASIC's e CPLDs, os contextos de *hardware* e *software* se confundem em um único universo "programável" que permite infinitas combinações, dificultando ainda mais a decisão do particionamento. Para se racionalizar e otimizar esse processo, (SURESH et al., 2003) mostra algumas ferramentas de análise de desempenho, que permitem verificar quais partes de um *software* poderiam ser portadas para um *hardware* configurável e vice-versa. (BHATTACHARYA et al., 2008), por sua vez, apresenta uma metodologia de análise e otimização utilizando algoritmos genéticos e *Particle Swarm Optimization*. Entretanto, essas técnicas só são efetivas quando já existe um projeto inicial, ou um protótipo já realizado, e que necessita apenas de aperfeiçoamentos.

3.1.2 Programação

Uma vez que já estejam definidos minimamente os papéis do *hardware* e *software*, nota-se que o desenvolvimento do *hardware* (principalmente sua parte não-customizável) é relativamente mais rápido, com menos iterações. Seu ciclo de projeto é constituído basicamente de uma etapa de seleção de componentes (pré-fabricados) e de etapas de integração, montagem e testes. Todas essas fases são baseadas em conceitos bem definidos, tais como padrões de projeto, experiências consagradas, interfaces padronizadas e recomendações, normas e definições dos fabricantes dos componentes. Em um esforço concentrado e localizado (CATSOULIS, 2005) é possível criar uma infra-estrutura relativamente sólida, onde o restante do desenvolvimento de *software* pode ser executado. Essa infra-estrutura é também chamada de "plataforma de *hardware*", conforme (SANGIOVANNI-VINCENTELLI; MARTIN, 2001).

É no desenvolvimento da parte programável que os problemas mais graves aparecem, quando o efeito de escolhas ruins, seja na definição da plataforma ou na definição do particionamento, se juntam aos demais problemas de engenharia de *software*, como aqueles descritos por (BROOKS, 1995)[3], tais como: especificações pobres ou mutantes, má inter-

[2]Ou "sintetizáveis", conforme preferem os "hardweiros" do mundo digital.

[3]Brooks ficou famoso por sua citação - "There's no silver bullet on software engineering", quando compara os projetos de *software* a lobisomens. Inicialmente são simples e familiares, mas subitamente tornam-se monstros terríveis. Contudo, diferentemente do folclore, em *software* não existe uma bala de prata para se acabar com o problema.

pretação do particionamento durante o *design*, ausência de ferramentas capazes, ausência de recursos, erros de implementação, testes pouco específicos, etc.

Durante o *design* do *software*, nas etapas de pesquisa de novos algoritmos e técnicas computacionais, é muito comum a utilização de ferramentas de prototipagem de códigos e simuladores *offline*, tais como o MatLab, Simulink, Vissim, etc., vistos adiante com mais detalhes. Essas ferramentas são importantes e necessárias já que liberam o poder criativo do usuário, que pode contar com uma grande variedade de recursos computacionais presentes no seu computador de desenvolvimento, tais como: memória, poder de processamento, armazenamento em disco, recursos de depuração, etc. Essa disponibilidade de recursos desonera os projetistas dos problemas e das limitações presentes nas plataformas dos sistemas embarcados. O resultado é uma prototipagem rápida e eficiente do algoritmo.

Entretanto, essas ferramentas de desenvolvimento escondem os detalhes intrínsecos das plataformas finais que o projetista fica com a falsa impressão que esses códigos poderão ser facilmente adaptados para o ambiente embarcado, o que não é verdade. Na realidade, o processo irá exigir ainda mais iterações para que seja possível adaptar o código feito, para que seja executado dentro do dispositivo embarcado com o desempenho desejado.

Por todas essas razões, o desenvolvimento desse tipo de *software* assume o comportamento em espiral comentado por (WILSON; RAUCH; PAIGE, 1992) e (BOEHM, 1986) e mostrado na Fig. 1, necessitando de várias iterações até que sejam alcançados os objetivos propostos.

3.2 Soluções de metodologias de desenvolvimento

Para se racionalizar a espiral de desenvolvimento do *software* existem várias recomendações, dos quais destacamos:

- Aplicação de métodos organizacionais;
- Uso de elementos prontos;
- Favorecimento do reuso de algoritmos e técnicas;
- Uso de padrões de projeto;
- Uso de técnicas de *co-design* com o *hardware*;
- Desenvolvimento baseado em plataforma, e;
- Desenvolvimento baseado em *frameworks*.

3.2.1 Métodos organizacionais

O objetivo de tais métodos é organizar a infra-estrutura humana (engenheiros, programadores), a infra-estrutura física (computadores, ferramentas e recursos) e as especificações e metas, de forma a favorecer o desenvolvimento, a produtividade e as contribuições construtivas, diminuindo o retrabalho, os erros de programação, as inconsistências de documentação, entre outros problemas de projeto.

Alguns exemplos são as metodologias de desenvolvimento racional *Agile* (BEEDLE et al., 2001), os métodos de programação em duplas do *Extreme Programming*, uso de ferramentas *Computer Aided Software Engineering* (CASE), ferramentas de controle de versões e ambientes integrados de desenvolvimento (do inglês, *Integrated Development Environments* (IDEs)).

3.2.2 Elementos de *software* prontos

O uso de pacotes prontos de *software* é citado por (BROOKS, 1995) como fundamental para evitar os longos períodos e recorrências de desenvolvimento.

Ao invés de se recriar arquiteturas e rotinas elementares, deve-se observar se essas já existem prontas e testadas por terceiros na forma de pacotes e bibliotecas comerciais (*off-the-shelf software* ou *software* de "prateleira"). O trabalho de integração desses elementos é muito menor que seu desenvolvimento pleno, baseando-se praticamente na utilização de uma *Application Programming Interface*s (APIs) de programação.

Esse método existe de forma análoga em projetos de *hardware*, onde utilizam-se circuitos integrados de uso específico para montagem de uma arquitetura maior. Entretanto, enquanto essa prática é muito comum em *hardware*, ela é pouco aplicada no ambiente de *software* embarcado, por três razões principais:

- Existem possíveis problemas de compatibilidade, já que que muitos dos pacotes e bibliotecas produzidas por terceiros são produzidos e testados em plataformas diferentes daquela que será usada na aplicação final, o que pode gerar um grande retrabalho de adaptação;

- Existem possíveis problemas de desempenho, já que os elementos programados por terceiros podem não ter sido vislumbrados para utilização em cenários com requisitos tão exigentes, o que pode implicar na recodificação de partes do sistema, e;

- Existe a falsa impressão que os custos e o tempo de desenvolvimento aumentam com a compra de módulos pré-produzidos por terceiros.

Em todo o caso, a utilização de especificações claras, o estudo minucioso das alternativas de mercado e a utilização de linguagens de programação e plataformas de *hardware* padronizadas torna esse tipo de método viável e proveitoso.

3.2.3 Reuso de código

Um dos pilares em Engenharia de *Software* é o termo "reuso" de código, significando utilizar, por mais de uma vez, um determinado trecho de programa (função, sub-rotina, módulo) em mais de um projeto (STRINGHAM, 2009). Intuitivamente a ideia é sensata pois podem-se aplicar esforços para se desenvolver apenas as partes inéditas ou inovadoras, deixando as partes corriqueiras de um projeto para serem construídas com elementos e blocos já consagrados e utilizados com sucesso no passado (BROOKS, 1995).

Apesar de existirem comparações do reuso de código com uma espécie de folclore ou utopia devido a dificuldade em sua implantação, a verdade é que essa prática é possível ou benéfica se algumas premissas forem utilizadas na concepção e manutenção dos módulos, funções e blocos elementares que serão reutilizados (ALMEIDA et al., 2007). Algumas dessas premissas são:

- Os elementos devem ser obtidos de códigos ou funções utilizadas de forma recorrente em programas anteriores;
- Cada bloco de código deve ser unívoco ao desempenhar uma dada função (DRY[4]);
- Os códigos desses blocos devem ser pequenos e simples (KISS[5] é outra boa prática em programação, autoexplicativa);
- Os blocos devem ser parametrizáveis para se adaptarem a vários cenários;
- As interfaces e argumentos da função ou bloco devem ser simples ou então agregados em conjuntos, e;
- Os trechos de código devem ser generalizados e não devem utilizar detalhes específicos de uma arquitetura, a não ser que essa seja usada de forma frequente em vários projetos;

Algumas ferramentas modernas de programação, tais como metalinguagens e geradores automáticos de código, baseiam-se fortemente nesse princípio de reutilização de código. A ideia é criar automaticamente a estrutura do *software* com base numa descrição

[4]DRY ou *Don't Repeat Yourself* é uma prática de programação que preconiza não implementar duas ou mais vezes a mesma rotina ou funcionalidade.

[5]KISS ou "*Keep It Simple, Stupid*" ou "*Keep It Small and Simple*"

de comportamento e em funções elementares. Tais ferramentas são bastante populares na atualidade, entretanto pecam pelo uso excessivo de memória, baixo desempenho do código final e alta complexidade do código gerado.

Com base nas mesmas premissas mostradas de reúso de *software*, existe uma técnica semelhante denominada síntese comportamental, utilizada com sucesso na área de criação de *hardware* do tipo ASIC's, como será mostrado adiante.

3.2.4 Padrões de projeto

Ao invés de promover apenas o reúso de partes de programas na forma de funções, blocos e trechos de código, a ideia de padrões de projeto (do inglês, *design patterns*) possui um conceito mais amplo, significando a criação e reutilização de padrões de *design* de *software*, inclusive de forma independente da linguagem que será utilizada (LAPLANTE, 1996).

Essa ideia também é sensata, uma vez que muitas aplicações possuem arquiteturas de *software* semelhantes, como são os casos das aplicações HRTCS para sistemas de potência já citadas.

3.2.5 Técnicas de *co-design*

Desde o final da década de 80, um novo tipo de abordagem de *design* tem sido empregado com sucesso, a síntese de *hardware* digital (GAJSKI, 1988). A ideia foi concebida como solução para os altos custos de criação e desenvolvimento de circuitos integrados especializados (*full-custom*), que consumiam horas de projeto, desde o *design* de seus transistores até o *design* de elementos mais complexos como Unidades Lógicas Aritméticas, etc.

A chamada síntese de *hardware* digital funciona a partir de um conjunto de bibliotecas de "células" pré-definidas (cujos transistores e componentes fundamentais já foram desenhados e consagrados), que representam funções e elementos típicos de eletrônica digital. Com essa biblioteca, o projetista cria uma descrição funcional ou comportamental do *hardware* desejado por meio de uma linguagem de alto nível, como o VHDL (IEEE1076, 2008) ou o Verilog (IEEE1364, 2005). Essa descrição comportamental, por sua vez, é "compilada" dando origem a uma arquitetura de *hardware*. Essa arquitetura pode então ser fabricada em circuitos integrados dedicados ou colocada em lógicas programáveis do tipo PLD, FPGA ou CPLD, dando origem a um ASIC.

As ferramentas de síntese modernas são bastante poderosas, podendo gerar arquiteturas bastante elaboradas, tais como microprocessadores, DSP's e *stream processors*

representando algoritmos bastante complexos, conforme a necessidade da aplicação.

Um exemplo de *stream processor* para a área de sistemas de potência pode ser visto no trabalho de (CHEN et al., 2009), com um dispositivo FPGA representando as equações de um modelo matemático de um motor de indução para avaliação da capacidade computacional de uma FPGA. Outro trabalho é o de (USENMEZ et al., 2009), com o modelamento de um motor de indução, junto de seu acionamento e controlador de torque, dentro de uma FPGA para testes do tipo HiL.

Mais recentemente surgiu uma extrapolação da ideia de síntese de *hardware*, de forma a abranger também o *design* (ou síntese) do *software*. Esse processo é chamado de *co-design* ou "síntese de sistema". Nesse caso, o comportamento de todo o sistema - seja ele composto de lógicas digitais combinatórias, lógicas digitais sequenciais, descrições algoritmicas, fluxos de dados, etc. - é descrito em uma linguagem de alto-nível, em um processo denominado "descrição sistêmica-comportamental". Além do VHDL e Verilog, uma das linguagens mais usadas na atualidade para descrição de sistemas é o SystemC (OSCI, 2010).

O resultado de sua "compilação" é uma arquitetura conjunta de *hardware* e *software* prontas, onde as próprias ferramentas de análise e compilação se encarregaram de resolver o "problema do particionamento", colocado anteriormente, e a otimização do *design*.

Até 2002, as ferramentas de "síntese de sistemas" disponíveis comercialmente apresentavam resultados apenas satisfatórios, pois resultavam em arquiteturas muito complexas, com alto consumo de energia e baixo rendimento computacional (SANGIOVANNI-VINCENTELLI; MARTIN, 2001) e (MICHELI; ERNST; WOLF, 2002). Além disso, cada iteração do ciclo de desenvolvimento é muito lenta ao se utilizar esses métodos, já que as ferramentas utilizam algoritmos heurísticos para resolver o problema explosivo do "particionamento" entre *hardware* e *software* (MICHELI; ERNST; WOLF, 2002). Enquanto que na síntese de *hardware* é comum ouvir relatos de engenheiros depurando aplicações, onde cada tentativa de compilação e síntese levava de dezenas de minutos a horas de processamento em computadores modernos, na síntese de sistemas, existem arquiteturas que podem durar dezenas de horas a dias para serem sintetizadas.

Um fator interessante desse tipo de abordagem é que as ferramentas podem realizar também simulação e verificação da arquitetura proposta, auxiliando o usuário no processo de desenvolvimento e maturação, antes da realização da síntese propriamente dita. Conforme (MICHELI; ERNST; WOLF, 2002), a técnica de síntese de sistemas possui um grande potencial ainda por ser explorado, por exemplo com a evolução de suas ferramentas e a adoção de técnicas modernas de computação de alta performance para que o processo de "compilação" possa ser acelerado.

3.2.6 Plataformas

Existem alguns modelos de *design* que faciliam a questão de desenvolvimento do *software* e a dualidade do "problema do particionamento". Um desses modelos é o "*design* baseado em plataformas".

Segundo (SANGIOVANNI-VINCENTELLI; MARTIN, 2001), "plataforma" é uma abstração que engloba várias características de baixo-nível de um determinado contexto para constituir uma macro-visão em camadas desse contexto. Por exemplo, uma "plataforma de *hardware*" é uma forma de observação de todo o conjunto de seus componentes (circuitos, processadores, interfaces, memória, etc.) sob a perspectiva de um bloco único e monolítico, escondendo os detalhes intrínsecos de sua constituição.

Nos sistemas embarcados, o modelo de *design* baseado em plataforma de *hardware* considera esse como uma estrutura única, definida de forma pragmática, com base na especificação da aplicação e em algum superdimensionamento de recursos e capacidade. A ideia é escolher adequadamente, e de imediato, a infra-estrutura de *hardware* da aplicação e considerá-la uma caixa preta durante o desenvolvimento de seu *software*. Esse modelo traz como benefícios:

- A utilização de *hardware* maduros e já consagrados em outros cenários;
- A possibilidade de definição rápida das ferramentas e materiais de apoio para a criação do *software*;
- A possibilidade de uso de APIs, bibliotecas e o reaproveitamento de outros recursos de outros projetos que utilizam a mesma infra-estrutura;
- A disponibilidade de um suporte de usuário mais rico, já que uma plataforma usada em outras aplicações, com vários outros usuários desenvolvedores, possui uma base de conhecimento maior, e;
- Um maior conforto quanto a suporte e tempo de vida do projeto, garantido pelo fabricante da plataforma.

Logicamente, o uso de plataformas de *hardware* pré-definidas, generalistas, também apresenta pontos negativos, tais como: maiores custos para a aquisição, maiores custos em produção seriada, produtos com superdimensionamento de recursos, etc.

Na área de prototipagem de dispositivos eletrônicos e HRTCS a utilização de plataformas é um excelente ponto de partida para um projeto. Após testes iniciais e provas de conceito, se houver interesse, pode-se então fazer uma otimização "*up-down*" do *design*, ou seja, a partir do *software* já desenvolvido e plenamente funcional, podem ser repensados

os componentes e a estrutura da plataforma de *hardware*, com o intuito de racionalizar sua arquitetura e diminuir complexidades e custos.

Um parênteses deve ser feito aos HRTCS reconfiguráveis e reprogramáveis. Nesse caso, a adoção de uma plataforma de *hardware* superdimensionada - principalmente nos recursos de capacidade de processamento e memória - é fundamental para a vida útil do projeto.

Outro detalhe importante segundo (SANGIOVANNI-VINCENTELLI; MARTIN, 2001) é que o conceito de plataforma pode ser ampliado também para uma parte da infra-estrutura de *software*, com a escolha e definição adequada do Sistema Operacional/Executivo e da Camada de Abstração de *Hardware*. Nesse caso, o trabalho dos engenheiros da aplicação será o de apenas desenvolver ou adaptar o *software* (conjunto de tarefas) ao ambiente da plataforma integrada de *hardware/software*.

3.2.7 Arcabouços ou *frameworks*

A adoção da maior quantidade possível de metodologias certamente favorece a criação e a manutenção do *software* de um sistema HRTCS. Esse é o papel dos chamados arcabouços (do inglês, *frameworks*), ou seja, um conjunto de conceitos ao redor de um determinado domínio de conhecimento com o intuito de resolver ou simplificar a solução de um problema, ocultando do desenvolvedor as micro-complexidades dessa solução (HENZINGER; SIFAKIS, 2007).

Existem arcabouços dedicados para desenvolvimento de *software*, para desenvolvimento de estudos de sistemas de potência, para aplicações em matemática, física, economia, etc. O que muda de um ao outro é a abrangência de seu domínio (área de especialidade) e seu resultado final (um *software*, um *hardware*, uma simulação, uma projeção, etc.).

Segundo (SIFAKIS, 2005), em geral, as características típicas de qualquer *framework* são:

- Permitem criar sistemas complexos;
- Sistemas são criados a partir de componentes do domínio da aplicação;
- Os componentes são pequenos, compartimentados, bem definidos (com parâmetros e propriedades) e auto-sustentáveis (um componente primitivo não depende de outro para existir ou funcionar);
- Os componentes são agregados segundo uma determinada metodologia, álgebra ou semântica, bem definidas;

- Essa metodologia ou álgebra deve poder ser aplicada através de uma interface de alto-nível;
- Componentes maiores podem ser criados pela álgebra de componentes menores;
- Abstraem as complexidades do restante da arquitetura, apresentando uma visão mais alto nível e próxima do verdadeiro domínio da aplicação.

Dessa forma, o usuário de um *framework* não precisará entender ou ter contato com os detalhes mais intrínsecos das camadas inferiores. Ele pode se ater ao desenvolvimento e à solução dos problemas específicos de seu domínio de conhecimento, utilizando para isso uma interface e uma metodologia de alto nível. O resultado é uma maior produtividade e uma maior garantia de sucesso na implementação de seu trabalho.

Outro fato relevante é que seus componentes elementares, por serem menores e compartimentados, podem ser verificados e testados de forma individual e simples. Assim, na constituição de cenários mais complexos, os resultados da integração dos componentes são muito mais previsíveis e seguros.

Nesse contexto, por exemplo, um *framework* exclusivamente para estudos de Sistemas de Potência consiste em um conjunto de elementos, componentes ou classes (RIEHLE, 2000) de Sistemas de Potência, tais como linhas de transmissão, geradores, modelos de carga, etc., que podem ser utilizados em conjunto para a criação de uma aplicação, abstraindo os detalhes computacionais e numéricos dos modelos. Um exemplo de *framework* específico para estudos de Sistemas de Potência, com todas as características mostradas anteriormente, é descrito no trabalho de (LI, 2001).

3.2.8 Qual a solução para todos os problemas ?

Independentemente das premissas colocadas, a resposta para a pergunta acima é óbvia: - “Depende da aplicação.”. Notadamente nos cenários descritos dos HRTCS, onde deve-se haver prototipagem rápida de código, possibilidade de alterações de forma simples, sistemas que precisam ser reajustados ou reconfigurados frequentemente, etc., a melhor ideia é a adoção de todas as premissas citadas anteriormente, de forma simultânea. Entretanto essa é uma tarefa praticamente impossível, já que existem alguns conceitos, como *co-design* e o uso de plataformas de *hardware*, que são até contraditórios.

Deve-se notar que um *framework* de *software* é um método que agrega conceitualmente pelo menos dois dos conceitos já citados, como o reuso de código e os padrões de projeto.

Ainda, se um *framework* for utilizado com uma APIs e bibliotecas de apoio para a construção de seus componentes primitivos, agrega-se também o conceito de uso de

módulos prontos de *software*.

Extrapolando-se ainda mais, caso seja definida uma arquitetura de *hardware* - ampla o suficiente para atender a todas as aplicações desejadas - e uma arquitetura de *software* - com Sistema Operacional/Executivo e *Drivers* adequados - agrega-se mais um conceito: a adoção de uma *plataforma* para desenvolvimento.

Dessa forma, um *framework* de *software* parece ser a solução que agrega mais valores, permitindo a construção de aplicações computacionais direcionadas a uma dada área de conhecimento, sobre uma estrutura muito bem sólida e definida, de forma rápida, modular, previsível, modificável e expansível, com uma diminuição considerável no número de iterações para o desenvolvimento do projeto, reduzindo custos e tempo de mercado (SANGIOVANNI-VINCENTELLI; MARTIN, 2001).

Uma questão importante a respeito da utilização dessa técnica é sobre seu desempenho. Como o *framework* de *software* é uma camada de abstração, seu aproveitamento das capacidades do *hardware* não é o mesmo se comparado com um programa implementado diretamente em uma dada linguagem de programação compilada.

Outro ponto relevante é que apesar do *framework* otimizar o desenvolvimento de aplicações de *software*, sua própria criação (e a de seus elementos básicos) são um projeto a parte, que possui todos os problemas de desenvolvimento comentados anteriormente. Entretanto, se seu *design* for adequado, o maior esforço de desenvolvimento será feito apenas uma vez e, depois, somente portado e utilizado em outros cenários e aplicações.

3.3 Soluções disponíveis na técnica

Até o ano de 2010 podem ser observados vários trabalhos de pesquisa e produtos comerciais que abordam os problemas de desenvolvimento de sistemas HRTCS, principalmente na questão da criação, implementação, testes e manutenção de *software*.

A seguir são relacionadas as contribuições mais relevantes que poderiam ser usadas individualmente ou em conjunto para criar a plataforma completa de desenvolvimento de um HRTCS, principalmente para aquelas aplicações com as características e os requisitos necessários para os ambientes de Sistemas de Potência.

3.3.1 Diretivas para desenvolvimento de *software*

Já existem a alguns anos várias diretrizes de gerenciamento de projetos envolvendo *software* que merecem destaque, tais como as diretivas da RTCA DO-178B (RTCA, 2010) e as diretivas da ECSS (ECSS, 2010). Ambas as iniciativas são amplamente utilizadas pela

indústria aeronáutica e aeroespacial, principalmente em projetos críticos que envolvem o desenvolvimento de *software* e, algumas vezes, também de *hardware*.

Em ambos os casos, tais diretivas preconizam as etapas de desenvolvimento do *software*, desde sua especificação inicial até suas etapas de verificação e validação, com especial ênfase aos critérios que serão utilizados para a qualificação e a certificação final do *software* para as aplicações em tais ambientes embarcados.

Interessante frisar que essas diretrizes não determinam ou orientam o desenvolvedor a respeito de como o código fonte deve ser criado ou de como um projeto composto de *hardware/software* deve ser conduzido, entretanto elas propõem modelos dos produtos intermediários que devem ser gerados durante a linha de desenvolvimento de um projeto. Basicamente, tais produtos designados são documentos e modelos de documentos do processo de engenharia de *software*, com formatos e conteúdos padronizados, de acordo com a etapa em questão, tais como: documentos de especificação, documentos de análise de requisitos, documentos de detalhamento do projeto, documentos de procedimentos de teste, documentos de teste e verificação, documentos de validação e certificação, etc.

Tais metodologias são mandatórias em alguns países para que a indústria do *software* possa embarcar uma aplicação dentro de uma aeronave, como é o caso do FAA Norte-americano (FAA, 2010), que requer que um determinado produto de *software* atenda plenamente ao documento DO-178B da RTCA.

A princípio, tais diretrizes parecem burocratizar o processo de desenvolvimento de *software* embarcado. Entretanto elas foram especificamente criadas para racionalizar o processo de gerenciamento desses tipos de projetos, criando memoriais e acervos de documentos (úteis e concisos) utilizados desde a etapa de especificação, codificação, até os testes e certificação final dos produtos.

No caso dos projetos de *software* para as aplicações HRTCS descritas nesse trabalho, tais metodologias podem contribuir positivamente para o processo de desenvolvimento, principalmente na gerência do projeto e no amadurecimento de sua documentação. Entretanto, não resolvem o problema em seu âmago, a respeito de como o *software* da aplicação embarcada poderá ser codificado e recodificado por diversas vezes, mantendo sua coerência, seus resultados e desempenho.

3.3.2 Alternativas de linguagens de programação

Existe uma infinidade de linguagens de programação no mercado. Nesse trabalho foram consideradas as mais comuns para o ambiente de desenvolvimento de sistemas embarcados e simuladores em tempo real: ADA, Java, LabView “G”, linguagens da norma IEC 61131-3 e ANSI C.

ADA é uma linguagem estruturada de programação que surgiu no final na década de 1970, de semântica e sintaxe complexas, com uma descrição bastante rigorosa (e burocrática) das funções, variáveis, estruturas, etc. Ela é tradicionalmente utilizada em cenários de tempo real determinístico e sistemas críticos, principalmente em aplicações governamentais, militares e espaciais pelos norte-americanos. Fora desses ambientes não é uma linguagem muito popular. Sua manutenção é simples e o código fonte é naturalmente bastante legível e auto-documentado, não permitindo construções exóticas ou complexas, favorecendo as etapas de verificação e validação de seus projetos.

Java (JAVA, 2010) é uma linguagem de programação que surgiu na década de 1990, com suporte nativo a programação orientada a objetos. Sua principal característica é tentar ser portável a qualquer plataforma de mercado através de uma estrutura composta por uma máquina virtual padrão e um programa compilado para ela, em um formato conhecido como *byte-code*. Esse código é interpretado *byte-a-byte* pela máquina virtual, razão pela qual seu desempenho costuma ser bastante inferior das linguagens puramente compiladas, como o C. O código é independente de plataforma mas a Máquina Virtual precisa ser especialmente construída para se adequar aos recursos do *hardware*. Outro problema da linguagem advém de sua forte orientação a objetos, que não permite uma clara noção do tempo de processamento ou do caminho crítico do programa em função da criação e destruição de objetos ao longo de sua execução. Seu uso é muito popular em dispositivos móveis portáteis, como celulares, GPS's e também em aplicações ligadas ao ambiente da INTERNET. Existem máquinas virtuais JAVA portadas para aplicações em tempo real (RTSJ), mas seu desempenho para tarefas agendadas com resolução inferior a 1,0 [ms] depende da plataforma de *hardware* e *software* onde a máquina virtual está sendo executada e, em geral, é pior se comparada a outras linguagens, até mesmo interpretadas.

Tanto o JAVA quanto o ADA são linguagens que abstraem bastante a arquitetura de *hardware* do dispositivo computacional. Por essa razão, enquanto que por um lado elas são mais simples para os programadores (possuem gerenciamento de memória mais fácil, não há necessidade de ponteiros, a passagem de parâmetros é mais objetiva, etc.), elas são mais difíceis de serem aplicadas em arquiteturas novas, pois necessitam da implementação e do suporte ao *hardware* e aos *drivers* de dispositivo.

O ambiente de desenvolvimento LabView (NATIONAL, 2010) possui uma linguagem de programação denominada "G" que é baseada em um paradigma de codificação totalmente gráfico, com blocos elementares de programação, componentes e estruturas de dados, que podem ser associados para constituir um determinado programa. A linguagem tem sido desenvolvida desde meados da década de 80 e permite representar visualmente, de forma sistêmica e intuitiva, um algoritmo ou processo, inclusive com conceitos de programação paralela, de forma transparente. Seus principais pontos negativos são que a linguagem só

pode ser executada dentro do ambiente LabView ou em um *hardware* proprietário da National Instruments, e não pode ser portada para outras arquiteturas de microcontroladores, DSP's, etc., que não sejam suportadas pela National Instruments.

A norma IEC 61131-3 (IEC61131, 2003) é um esforço pela padronização das linguagens de programação para utilização em PLCs e alguns IED's de mercado. As linguagens padronizadas foram: o LADDER, o *Function Block Diagram* (FBD), o *Structured Text* (ST), o *Instruction List* (IL) e o *Sequential Function Chart* (SFC). O LADDER é muito conhecido no meio de automação, sendo uma linguagem mímica dos diagramas de contatos e bobinas feitos antigamente com contatores e relés. O FBD é uma linguagem gráfica de descrição de fluxos de dados analógicos e digitais, muito intuitiva, como aquelas utilizadas pelo MatLab Simulink, Vissim e LabView citados adiante. A linguagem ST é uma linguagem textual estruturada, muito semelhante em sintaxe à linguagem PASCAL. O IL é uma linguagem de execução de blocos funcionais de baixo nível, descrita textualmente, muito semelhante a um *Assembly*, com operadores matemáticos, lógicos e uma pilha de argumentos. Por fim, a SFC é uma linguagem gráfica de descrição de máquinas de estados e processo sequenciais.

A norma IEC 61131 defende que uma aplicação deve ser descrita com uma ou mais linguagens, simultaneamente, cada uma aplicada a um domínio do problema. Um exemplo de aplicação, acompanhado de uma excelente revisão das linguagens da norma é feita por (FAUSTINO, 2005).

Tais linguagens são muito interessantes para aplicação em um problema de desenvolvimento de *software*, mas entretanto, a norma IEC 61131 apenas descreve como essas linguagens devem se comportar e não como executá-las em suas plataformas de *hardware* (FAUSTINO, 2005). Uma das poucas iniciativas em se portar essas linguagens para o universo de HRTCS é o FORTE e 4DIAC (4DIAC, 2010), ferramentas baseadas também na IEC 61499 de sistemas de controle distribuído, e o ISAGraf citado a seguir.

A linguagem ANSI C (KERNIGHAN; RITCHIE, 1978) é uma linguagem estruturada criada no princípio da década de 1970, para desenvolver o sistema operacional Unix em vários tipos de arquiteturas de computadores disponíveis. Por essa razão ela já nasceu com recursos que permitem o acesso a vários aspectos intrínsecos do *hardware*, tais como ponteiros de memória e a capacidade de enxerto de códigos em linguagem de máquina (*Assembly*). Entretanto, ainda assim mantém sua característica de linguagem de alto-nível, sendo a linguagem mais usada em sistemas embarcados, desde minúsculos microcontroladores até grandes processadores de múltiplos núcleos, e a linguagem mais popular da atualidade segundo (TIOBE..., 2010).

Devido ao seu potencial de acesso ao *hardware*, em algumas circunstâncias a linguagem C não apresenta uma boa portabilidade entre arquiteturas distintas, requerendo bastante

esforço dos programadores para realizar adaptações. Entretanto, o uso de rotinas e bibliotecas de abstração de *hardware* favorece esse trabalho. A linguagem possui uma grande quantidade de compiladores e ferramentas disponíveis no mercado, que permitem a criação de códigos compilados para uma infinidade de arquiteturas. O desempenho obtido com seus programas depende do compilador, mas em geral, é muito próximo daquele que seria obtido programando-se diretamente em código de máquina (*Assembly*). Segundo a comunidade de programadores e desenvolvedores do grupo "comp.lang.c" (USENET, 2010), um programa compilado em C chega a ser no máximo, duas ou três vezes mais lento que o respectivo código feito puramente em *Assembly*, enquanto que as demais linguagens chegam a ser de três a dez vezes mais lentas (principalmente as interpretadas).

Todas essas linguagens de programação são apenas as ferramentas de construção para uma dada arquitetura. Essas linguagens, por si sós, não resolvem os problemas descritos anteriormente, mas suas escolhas impactam no desenvolvimento global da aplicação. O ideal é agregar a uma dessas linguagens os outros valores já comentados.

3.3.3 Ambientes integrados de desenvolvimento

Existe no mercado uma série de ambientes de desenvolvimento (IDEs). Esses ambientes consistem em um conjunto de ferramentas, bibliotecas e *frameworks* para modelagem, prototipagem e simulação de sistemas. Os exemplos mais tradicionais são: o Matlab (MATHWORKS, 2010), o LabView (NATIONAL, 2010), o VisSim (VISSIM, 2010), o SciLab/SciCOS (INRIA, 2010) e o Modelica (MODELICA, 2010). Todos são ambientes generalizados, consagrados para aplicações científicas e comerciais, sendo amplamente utilizados no meio acadêmico, industrial e aeroespacial.

Desses ambientes, o MatLab, LabView e VisSim são ambientes pagos enquanto que o SciLab/SciCOS é uma ferramenta de código aberto. O Modelica possui implementações tanto de código aberto - como o OpenModelica da (OPENMODELICA, 2010) - quanto comerciais - como o Dymola da (DYNASIM, 2010).

Todos os ambientes possuem um conjunto de ferramentas para programação e *design* das aplicações, que possuem linguagens de sintaxe, semântica e metodologias próprias. O LabView, particularmente, apresenta apenas a linguagem visual "G". Nos demais ambientes, os códigos podem ser implementados em linguagem textual e/ou através de *frameworks* gráficos, que permitem a criação de algoritmos e fluxos de dados também de forma gráfica, como diagramas de blocos por exemplo.

Todos os ambientes possuem vastas bibliotecas para aplicações de controle, processamento digital de sinais, automação, programação estruturada, lógica booleana combinatória e sequencial, etc. O destaque vai para o MatLab, que possui uma rica coleção

de recursos externos, chamados *toolboxes*, que, entretanto, são pagos à parte da licença principal de uso do ambiente. O MatLab é um dos poucos que possui também um arcabouço (um *framework*) completo de modelos matemáticos consagrados de componentes de sistemas de potência, no chamado *PowerSystem Toolbox*.

É importante ressaltar que na descrição de sistemas dinâmicos através dessas linguagens e *frameworks*, cada ambiente deixa implícito o *solver* que será utilizado como método de solução das equações algébrico-diferenciais. Quando o usuário utiliza o ambiente para uma simulação *offline*, normalmente é utilizado o método mais preciso disponível, por exemplo, um *solver* algébrico-diferencial de passo variável iterativo. Entretanto isso não ocorre quando tais ambientes são utilizados para simulação em tempo real, como será visto adiante.

Todos os produtos, de uma forma ou outra, podem ser utilizados em aplicações de simulação em tempo real HiL. Entretanto, para obtenção do desempenho necessário, tais sistemas requerem a utilização de dispositivos de *hardware* especiais e dedicados, tais como placas de aquisição, processadores de alto desempenho, etc., com custos adicionais. Esses dispositivos de *hardware* são comentados adiante.

Em geral, todos esses ambientes possuem geradores automáticos de código fonte, capazes de converter o cenário montado pelo usuário (com todos os recursos dos *frameworks* e do ambiente) em uma listagem em linguagem ANSI C, para permitir a compilação e a utilização em sistemas embarcados. Entretanto tais ferramentas ainda são: muito incipientes, geram programas longos, com grande consumo e movimentação de memória, com estruturas de dados e de programa complexos, pouco otimizados, não fazem uso eficiente de paradigmas como processamento paralelo e distribuído, possuem manutenção muito complexa e são muito difíceis de serem analisados posteriormente por programadores. Além disso, os códigos gerados dependem de bibliotecas de funções e compiladores especiais para a arquitetura alvo onde serão utilizados.

É importante ressaltar que esses geradores de código são capazes de gerar apenas as rotinas de aquisição, processamento e síntese de dados da aplicação embarcada. A parte de *software* do sistema operacional, seus *drivers* de dispositivo e a também a plataforma de *hardware* são providas por fabricantes externos.

Outro fato importante é que quando as aplicações são executadas em tempo real em *hardware* dedicado ou então, quando os cenários são convertidos para linguagem C e compilados para uma aplicação embarcada, muitas vezes o ambiente substitui os algoritmos de integração numérica e os *solvers* por versões mais simples, como a integração numérica trapezoidal de passo fixo. Dessa forma, o desempenho em tempo real é mais determinístico e garantido, em detrimento de resultados comparáveis (em precisão e convergência) com aqueles obtidos em simulações e testes *offline*.

Atualmente, são muito populares no mercado os *kits* de desenvolvimento (pequenas plataformas de *hardware*), contendo processadores, DSP's ou ASIC's, que utilizam a estrutura desses ambientes como *front-end* para a criação de suas aplicações. Nesses *kits*, por exemplo, um engenheiro consegue criar rapidamente, em laboratório, uma malha de controle de um sistema dinâmico e testá-lo em uma aplicação em tempo real com bons resultados. Entretanto, essas abordagens só servem como provas de conceito, uma vez que as plataformas desses *kits* não possuem a robustez necessária para aplicações em cenários reais, possuem um custo relativamente elevado para produção seriada do seu *hardware* e possuem custos de licenciamento e uso de seu *software*, requerendo até mesmo o pagamento de *royalties* para serem "embarcados" e produzidos em série. Além disso, os códigos gerados são difíceis de serem portados para outras arquiteturas e principalmente, não utilizam todos os recursos ou o desempenho disponíveis na plataforma.

Um outro tipo de ambiente de desenvolvimento para aplicações embarcadas existe no mundo industrial há mais de 15 anos, chamado ISAGraf (ICSTRIPLEX, 2010). A ideia do ISAGraf é permitir a criação de aplicações com requisitos de tempo real determinístico para serem executados em uma máquina virtual própria. Essa máquina virtual é vendida para uma ampla gama de plataformas, inclusive sistemas embarcados, mas principalmente, PLCs de aplicações industriais. Os pontos positivos do ISAGraf são seu ambiente gráfico de desenvolvimento, a aderência às normas IEC 61131-3 (IEC61131, 2003) e IEC 61499 (IEC61499, 2005), comentadas anteriormente. O principal ponto negativo é seu desempenho relativamente baixo, com aplicações com resolução temporal máxima da ordem de 1,0 a 5,0 [ms].

3.3.4 Soluções de plataformas de *hardware*

Algumas soluções presentes no mercado para HRTCS são baseadas em plataformas de *hardware* de uso geral, associadas ou não a algum ambiente de desenvolvimento ou linguagem de programação.

Essas plataformas são dispositivos e placas de circuito com arquiteturas ou barramentos padronizados que permitem a constituição de configurações contendo aquisição, processamento e síntese de dados, segundo os requisitos dos HRTCS.

Alguns dos produtos que podem ser citados são: os PLCs de inúmeros fabricantes, alguns tipos de IED's, alguns sistemas dedicados de aquisição e controle, os *hardware* de simulação da dSpace (DSPACE, 2010), os *hardware* de simulação e controle CompactRIO da National (NATIONAL, 2010) e outras plataformas baseadas em barramentos industriais, tais como o VME/VPX da (VITA, 2010) e o PC-104/PCI-104 (PC104CONS, 2010).

Os PLC's existentes no mercado são plataformas de *hardware* interessantes para apli-

cações HRTCS's, principalmente pela robustez de seu *design* e pela grande variedade de configurações e fabricantes disponíveis. Entretanto, como não existe um padrão de arquitetura de *hardware*, os produtos e acessórios não são compatíveis entre si. Tais produtos podem ser programáveis segundo as linguagens da IEC 61131-3 (IEC61131, 2003) comentadas anteriormente. Entretanto, apesar da sintaxe e semânticas serem padronizadas, os arquivos gerados para armazenamento dos programas ainda não são compatíveis entre PLCs diferentes. Os maiores problemas para o uso dos PLCs como plataformas para os HRTCS's são os seus custos elevados e seus desempenhos satisfatórios, com resolução temporal não muito menor que 1,0 a 2,0 [ms] por varredura (*scan*).

Na atualidade, alguns fabricantes de IED's tem voltados seus esforços para o desenvolvimento de plataformas para HRTCS voltados para ambientes de subestações. Esse é o caso do dispositivo SEL-3351 da fabricante americana SEL (SEL, 2010), baseado em um computador de arquitetura Intel "x86" com 2,0 [GFLOPS] de capacidade de processamento. Entretanto esse produto ainda não possui cartões de entrada e saída para as finalidades de aquisição e síntese de dados analógicos e digitais externos.

Outros dispositivos que devem ser mencionados são produtos de aquisição de dados como os das empresas brasileiras (AQX, 2010) e (LYNX, 2010), que podem ser customizados para aplicações de HRTCS. Entretanto tais soluções não possuem um ambiente de desenvolvimento de *software* simples. O dispositivo montado por (PELLINI, 2005) como simulador de um gerador hidroelétrico era baseada em um produto da Lynx Tecnologia e possuiu todos os problemas de desenvolvimento de *software* imagináveis.

A empresa alemã dSpace se especializou em criar plataformas de *hardware* para uso com o ambiente de desenvolvimento MatLab em aplicações em tempo real determinístico. O principal objetivo de suas plataformas não é o uso embarcado, mas sim em ambientes de desenvolvimento em laboratório, principalmente no setor de simulação tipo HiL para as áreas automotivas, aeroespacial e industrial. Seus produtos, apesar dos custos elevados, possuem um dos melhores desempenhos no mercado, com execução de algoritmos em tempos inferiores a 10,0 [μs] graças a placas com processadores dedicados de arquitetura PowerPC.

A Opal-RT (OPAL, 2010) é uma empresa canadense cujo portfólio de produtos se iniciou com estruturas de *hardware* dedicadas para simulação em tempo real de modelos desenvolvidos no MatLab/Simulink. Suas soluções utilizam os recursos de geração automática de código em linguagem C do MatLab e um *software* proprietário de adaptação e compilação. Essa metodologia permite que os modelos sejam executados em computadores de arquitetura Intel "x86" com desempenho em tempo real determinístico e com tempos de execução inferiores a 50,0 [μs], utilizando recursos de computação paralela simétrica SMP. Como a dSpace, suas principais áreas de aplicação são como ferramentas

de desenvolvimento em laboratório e plataformas para simulações HiL.

A National Instruments possui um grande portfólio de dispositivos e acessórios para aquisição, processamento e síntese de dados em tempo real. O destaque fica por conta dos dispositivos da linha CompactRIO, que são verdadeiros PLCs de alto desempenho, configuráveis em termos de interfaces e capacidades de processamento, com versões robustas para uso em aplicações industriais. Todas as ferramentas de desenvolvimento do CompactRIO são baseadas na linguagem LabView e sua extensão para execução em tempo real LabViewRT. As configurações do CompactRIO possuem um custo relativamente elevado, pouco superiores às das placas da dSpace, mas possuem ótimos desempenhos, da ordem de 5,0 a 50,0 [μs] para processamento de dados e controle, graças a placas com processadores dedicados de arquitetura PowerPC e placas de apoio com lógicas programáveis tipo FPGAs.

Outras alternativas são sistemas construídos sobre barramentos industriais, tais como o VMEBus (VITA, 2010) e o PCI-104 (PC104CONS, 2010). Cada um desses barramentos são padronizados, possuem uma grande variedade de fornecedores de CPU's e periféricos, e podem constituir plataformas de *hardware* bastante poderosas para aplicações HRTCS. O único inconveniente é que tais esquemas não possuem ambientes de desenvolvimento ou *frameworks* padronizados e abertos, sendo quase sempre programados em uma linguagem básica de alto-nível utilizando-se plataformas de *software* comerciais, como as comentadas a seguir.

3.3.5 Sistemas operacionais e executivos

Existem várias soluções no mercado para os papéis de Sistema Operacional ou Executivo em tempo real e para o conjunto de *drivers* de dispositivo (HAL). Entre esses, podemos citar os pacotes de sistemas operacionais: QNX, VXWorks, ECos, FreeRTOS, RTLinux e RTAI/Comedi. Outro produto importante é o pacote do sistema executivo RTEMS.

Os sistemas operacionais em tempo real QNX (QNX, 2010) e VXWorks (VXWORKS, 2010) são produtos comerciais de grande sucesso no mercado de embarcados e sistemas críticos, possuindo aplicações nos mais diversos setores, tais como médico, automotivo, aeroespacial, aeronáutico, naval e em aplicações de IED's para automação e proteção para sistemas de potência. São pacotes completos de sistemas operacionais, com custo elevado, grande disponibilidade de *drivers*, compatíveis com quase todas as arquiteturas de processadores modernos, principalmente as baseadas nas CPU's PowerPC e Intel "x86". Ambos possuem ambientes de apoio ao *design* e desenvolvimento da aplicação, com alguns *frameworks* especializados produzidos por terceiros. O desempenho e a robustez desses

sistemas são ímpares. Ambos consomem recursos de memória do *hardware* entre 1,0 a 30 [MB] de memória RAM e entre 2,0 a 80 [MB] de memória ROM. Possuem recursos de multiprocessamento simétrico e, no caso do QNX, suporte a multiprocessamento distribuído de forma transparente para a aplicação e também suporte a execução de processos em CPU's dedicadas. O pacote de *software* QNX Neutrino para plataformas Intel "x86" é gratuito para aplicações de pesquisa e desenvolvimento, sem fins lucrativos. Aqueles projetos de produtos embarcados, tanto com o QNX ou com o VXWorks, devem pagar *royalties* aos seus respectivos fabricantes.

O ECos (ECOS, 2010) e o FreeRTOS (FREERTOS, 2010) são sistemas operacionais de tempo real, de código aberto, muito utilizados no ambiente de sistemas embarcados com microcontroladores e pequenos processadores. Não possuem *frameworks* ou ambientes dedicados de desenvolvimento, mas são baseados em programação em linguagem C ANSI e C++. Possuem suporte a uma grande variedade de plataformas de *hardware* e são bastante enxutos, podendo ser portados a outras arquiteturas. Não suportam multiprocessamento simétrico. Em algumas plataformas, o *kernel* desses sistemas consome pouco mais de 2,0 [kB] de memória RAM e cerca de 30 [KB] de memória ROM. Não possuem um conjunto muito extenso de *drivers* de dispositivo. O desempenho obtido com os sistemas é excelente graças a simplicidade de suas arquiteturas.

O RTLinux (RTLINUXFREE, 2010) e o RTAI (RTAI, 2006) são sistemas operacionais em tempo real baseados no *kernel* Linux. Ambos possuem modificações (do inglês, *patches*) que tornam o escalonamento de tarefas do Linux convencional não-preemptivo apenas para as tarefas em tempo real. Ambos são de código aberto, apesar de possuírem implementações comerciais vendidas por terceiros. Possuem amplo suporte à maioria dos processadores e arquiteturas modernas, inclusive aos recursos de processamento paralelo SMP do Linux convencional. Seu consumo de recursos é semelhante ao do VXWorks e do QNX. Não possuem ambiente de desenvolvimento e são programados em linguagem C. Podem ser utilizados com uma série de bibliotecas e *frameworks* de terceiros. Através da biblioteca COMEDI (HESS; ABBOTT, 2006), para suporte a interfaces de aquisição e síntese de dados, o RTAI, em conjunto com o ambiente de desenvolvimento e prototipagem SciLab/SciCOS, pode ser utilizado em projetos de simulação em tempo real HiL e algumas aplicações de controle, com ótimo desempenho. Apesar de ainda não serem soluções tão robustas quanto seus concorrentes comerciais, o RTAI e o RTLinux são citados pelo consórcio europeu OCERA (OCERA, 2010) como a melhor plataforma aberta para sistemas embarcados de tempo real (RIPOLL et al., 2002).

O RTEMS (RTEMS, 2010) é um Sistema Executivo desenvolvido desde o final da década de 1980. No início, tratava-se de um ambiente executivo de tarefas, com requisitos de processamento em tempo real determinístico, para apoio a sistemas computacionais

aero-embarcados em aplicações militares. Mais tarde, sua especificação tornou-se mais ampla e de código aberto. É um sistema muito utilizado no meio militar, principalmente nos Estados Unidos, Europa e Asia, suportando as linguagens de programação ADA e C em seu ambiente de desenvolvimento. Hoje, trata-se de um sistema operacional completo, com suporte a vários processadores (SMP), multitarefa, mas para execução de apenas um processo. Pode ser usado com uma grande variedade de arquiteturas de CPU's, inclusive pequenos microcontroladores e sistemas com requisitos estritos de memória. Seu desempenho é superior ao obtido com os Sistemas Operacionais em Tempo Real comentados anteriormente, dado sua arquitetura extremamente enxuta.

3.3.6 Ambientes dedicados para estudos de sistemas de potência

O ATP/EMTP (ATP/EMTP, 2010) é uma ferramenta de simulação e estudos de sistemas de potência, tradicional e consagrada, capaz de executar simulações *offline* de uma grande variedade de cenários com boa precisão. A ferramenta possui alguns recursos de programação através de uma linguagem de descrição de uso geral chamada MODELS, permitindo a codificação de algoritmos específicos e outros modelos. Entretanto, a arquitetura do simulador não foi projetada para aplicações HiL e não há como portar facilmente seu *solver* ou seus modelos matemáticos para outra plataforma, para que seja obtido o desempenho desejado.

Uma das implementações do EMTP para aplicações em tempo real é o HyperSim (IREQ, 2010), que utiliza uma estrutura de *High Performance Computing* (HPC) com supercomputadores para resolver os sistemas algébrico-diferencias com o desempenho desejado. Não há detalhes adicionais sobre sua estrutura de *software* ou sobre seu *framework* de *design* das simulações de sistemas de potência.

O *Real Time Digital Simulator* (RTDS, 2010) é um dos simuladores em tempo real mais conhecidos para aplicações em sistemas de potência do tipo HiL. Seu *engine* de simulação possui uma ampla biblioteca de componentes e elementos, além de recursos para programação que podem ser utilizados para prototipagem de algoritmos e modelos. O simulador é baseado em uma arquitetura de *hardware* com processamento paralelo simétrico e distribuído, cujas plataformas computacionais são placas com processadores PowerPC conectadas em uma rede de alta velocidade. O desempenho dessa arquitetura permite simulações com passos de tempo inferiores a 10,0 [μs]. Não há detalhes adicionais a respeito da linguagem de programação que pode ser usada ou da capacidade computacional disponível, por exemplo, para a implementação de um modelo computacional completo de um IED.

A empresa canadense Opal-RT (OPAL, 2010) desenvolveu recentemente um *solver*

para as equações algébrico-diferenciais criadas pelo ambiente de desenvolvimento do MatLab Simulink. Esse *solver* permite a simulação de grandes sistemas modelados com o *framework* do *PowerSystem Toolbox*. Chamado de *Artemis*, o *solver* permite a execução dos modelos e algoritmos, com passo de integração variável, mantendo o desempenho em tempo real determinístico, utilizando como plataforma computacional um *cluster* de computadores comuns baseados na arquitetura Intel "x86" e uma rede alta velocidade.

Todos esses ambientes dedicados mostrados anteriormente são importantes para os estudos de sistemas de potência. Além disso, servem de ferramentas de apoio ao engenheiro de desenvolvimento de um HRTCS, na execução, por exemplo, de testes do tipo HiL. Entretanto, quando utilizados como plataformas de prototipagem e desenvolvimento de algoritmos, seu papel é secundário, servindo apenas como prova de conceito de um determinado método, sem que esteja garantida sua portabilidade, sem dificuldades ou adaptações, a uma plataforma externa menor, mais robusta e dedicada àquela aplicação.

3.3.7 Arcabouços já desenvolvidos

O uso de arcabouços é comum em várias áreas da técnica. Como comentado anteriormente, dentro de vários ambientes de desenvolvimento, linguagens de programação, bibliotecas, etc. existem arcabouços de *software* para as mais diversas aplicações, como o *PowerSystem Toolbox* do MatLab (MATHWORKS, 2010) e as bibliotecas de controle e instrumentos virtuais presentes no LabView (NATIONAL, 2010).

Na área de sistemas embarcados e sistemas de potência existem trabalhos importantes que envolvem esses arcabouços que merecem destaque.

(FRöHLICH, 2001), em seu trabalho, propõe uma nova estratégia para a construção de sistemas com requisitos e desempenho em tempo real, como os HRTCS, através da aplicação de sistemas operacionais orientados para a aplicação, construídos por um arranjo de componentes reutilizáveis. Um dos objetivos de seu trabalho são as arquiteturas de sistemas embarcados.

Esse método de *design* "top-down" faz com que tanto à plataforma de *hardware* quanto à de *software* estejam bastante customizadas às necessidades da aplicação, favorecendo seu desenvolvimento e desempenho final. Para isso, a aplicação deve ser analisada e separada em seus domínios e, em cada um, deve ser aplicado um arcabouço de *software* específico, na forma de bibliotecas de componentes. A esse processo de *design* foi dado o nome de *Application Oriented System Design* ou AOSD.

Foi elaborada uma plataforma de *software* de um sistema operacional, com as premis-

sas de seu trabalho, chamado EPOS (*Embedded parallel Operating System*). Esse sistema foi usado como base para testes do método de *design* desenvolvido.

A respeito do *framework* desenvolvido, sua estrutura é baseada em orientação a objetos. O sistema descrito é compilado por ferramentas específicas e resulta em um código fonte comum. Esse é então compilado para a arquitetura destino onde é executado o EPOS. Os resultados obtidos são excelentes do ponto de vista de desenvolvimento de *software* de sistemas embarcados.

O trabalho de (LI, 2001) mostra um *framework* chamado "PowerFrame" criado especificamente para estudos de Sistemas de Potência, sem os compromissos de execução em tempo real, mas com requisitos estritos de precisão, convergência numérica e utilização de recursos de processamento paralelo distribuído.

Seu *framework* foi criado em uma arquitetura de quatro camadas com orientação a objetos. O *design* em camadas favorece o reuso e futuras extensões de componentes para criação de casos mais complexos. A camada mais elementar representa os componentes de Sistemas de Potência que podem ser agregados para formar um caso de estudo qualquer. A camada seguinte é responsável por hierarquizar os componentes e separar os domínios do problema em cenários de análise, tais como resolução de circuitos elétricos, fluxo de potência, cálculo de parâmetros, etc. A camada seguinte faz a execução dos algoritmos de resolução dos sistemas de equações. A última camada é responsável por fazer a comunicação entre outros resolvedores de sistemas presentes em outras plataformas computacionais, no caso de processamento paralelo distribuído. (LI, 2001) usa seu trabalho para resolver quatro problemas diferentes de sistemas de potência, demonstrando que o enfoque de *design* através de *frameworks* é eficiente.

O trabalho de (PICIOROAGA, 2004) mostra um *middleware* chamado "OSA+" desenvolvido para aplicações embarcadas em tempo real. O alvo de seu sistema são as pequenas aplicações embarcadas com pouca memória e baixa capacidade de processamento, utilizadas em ambientes de processamento distribuído. Seu *middleware* consiste em um *framework* orientado a objetos para criação e agendamento de tarefas que é executado sobre um sistema executivo mínimo tipo *microkernel*. São utilizados blocos específicos de comunicação para implementar a passagem de dados e objetos entre nós. Quando usado em um sistema de processamento paralelo distribuído heterogêneo, as comunicações são feitas entre os nós utilizando a arquitetura CORBA. Tanto o *framework* quanto o *middleware* são compilados para linguagem de máquina para obtenção de maiores desempenhos.

Em (NETO et al., 2010) é descrito um *framework*, denominado "MARTe", que consiste em um conjunto de módulos de aplicação genéricos, denominados GAM's (do inglês, *Generic Application Module*), executados por um escalonador em tempo real sobre um sistema operacional capaz. Os usuários do *framework* podem criar os GAM's conforme a necessidade, utilizando um conjunto pré-definido de componentes interligados em série ou paralelo. A ideia é que os GAM's possam ser desenvolvidos, testados e depurados em ambientes de desenvolvimento sem os requisitos de tempo real, para então serem utilizados na aplicação verdadeira. O *framework* é feito em linguagem C++ e amplamente orientado a objetos, sem interfaces de mais alto nível. Os usuários criam uma listagem em C++ (utilizando os componentes e objetos do *framework*) que é então compilada e descarregada na plataforma de escalonamento e execução. Sua primeira aplicação foi para o controle de estabilidade do plasma em aplicações experimentais de fusão nuclear do laboratório JET-EFDA na Inglaterra, onde todo o sistema de estabilização roda em tempo real, com agendamento de 50,0 [μs]. O *jitter* do agendamento de execução das tarefas, utilizando como plataforma de *hardware* um Intel Quad Core e como plataforma de *software* o Sistema Operacional Linux com o *patch* RTAI e COMEDI, foi inferior a 1,0 [μs]. Infelizmente nesse teste não é mostrada a complexidade do sistema de controle em termos de estados e variáveis de entrada e saída, ou sequer como o sistema faz uso das múltiplas unidades de processamento disponíveis na plataforma. O *framework* foi concebido para ser portado a vários tipos de plataforma, mas principalmente aquelas de alto desempenho e grande disponibilidade de recursos de memória.

O trabalho de (RIEHLE, 2000) aponta os vários benefícios de produtividade com o uso de *frameworks* apropriados para a área de domínio de cada aplicação. Em sua obra o autor apresenta uma estratégia de modelagem para criação do ambiente do *framework* baseada nos papéis dos componentes em um determinado cenário além da modelagem tradicional por classes orientadas a objetos. Sua tese explora os motivos pelo qual um *framework*, ao invés de ajudar em uma determinada aplicação, acaba por se tornar mais complexo que a mesma. Além disso, ele promove os critérios que devem ser utilizados para se fazer o *design* dessa arquitetura, sem que esses problemas ocorram.

O trabalho de (GUO; EDWARDS; BOROJEVIC, 2008) é um conjunto de funções e subrotinas especialmente desenhadas para controle e eletrônica de potência, com a finalidade de constituir uma biblioteca de componentes de *software* reutilizável e reconfigurável. A

abordagem utilizada para criar o *framework* utiliza todas as premissas de análise de domínio, ao identificar os elementos reutilizáveis e torná-los compartimentados. O resultado é uma biblioteca composta por unidades elementares denominadas ECO's (*Elementar Control Objects*) onde são codificadas equações no domínio do tempo discreto ou algoritmos procedurais. O usuário utiliza a biblioteca, criada em linguagem C, para constituir sua aplicação de fluxo de dados, sem preocupações quanto ao desempenho, etc., somente se focando no objetivo do sistema. O resultado é usado em um pequeno sistema executivo em tempo real determinístico denominado "DARK".

A ideia desse trabalho é comparar esse tipo de abordagem com o aquele tradicionalmente feito em eletrônica de potência, onde muitos elementos tem de ser implementados em baixíssimo nível para se obter a performance necessária gerando grandes problemas no desenvolvimento do *software*. A conclusão do trabalho afirma que o ambiente criado é muito melhor para o desenvolvimento de aplicações, às custas de um pior desempenho se comparado com as técnicas tradicionais. Entretanto, como frisado pelo autor, com o surgimento de processadores mais poderosos e com menos restrições de memória, esse tipo de enfoque com *frameworks* certamente irá prevalecer.

O trabalho de (DAVARE et al., 2007) defende a metodologia de *design* simultâneo de *hardware* e *software* baseados em plataformas, com a utilização do *framework* "Metropolis" e seu sucessor "Metro II", para apoio ao desenvolvimento de aplicações, principalmente as embarcadas. Nessas ferramentas, é utilizada uma metalinguagem, especialmente desenvolvida, para descrição do dispositivo em termos de suas funcionalidades e comportamentos. A metalinguagem se apoia em uma biblioteca de componentes, incluindo núcleos de *hardware* externos especialmente desenvolvidos para utilização em FPGAs. O processo de análise e síntese é feito através da interpretação da metalinguagem, incluindo o particionamento entre *hardware* e *software*, a determinação dos micro-componentes alocados para a solução, sua disposição física, os fluxos de dados de algoritmos e o escalonamento das tarefas, resultando em uma arquitetura completa de um processador. Não são fornecidos dados de desempenho ou da complexidade dos esquemas testados.

O trabalho de (COSTA et al., 2007) apresenta um *middleware* para sistemas embarcados, heterogêneos, reconfiguráveis, conectados em rede, com o propósito de facilitar o desenvolvimento de *software* desses sistemas. Como em outros trabalhos, a abordagem fez uma divisão da estrutura da aplicação em duas camadas, um *middleware kernel* e um *framework* orientado a objetos de componentes reutilizáveis. Essa estrutura, denominada

"RUNES" foi utilizada em redes de sensores e atuadores em uma dada infraestrutura de transportes numa rodovia. O usuário elabora a aplicação de forma independente da arquitetura através do *framework*. O *middleware* se encarrega então de tornar a aplicação desenhada funcional, fazendo a comunicação com nós vizinhos e também, reconfigurando a mesma caso ocorra a falha em algum componente de sua arquitetura.

4 Arcabouço para aplicações em tempo real em sistemas de potência

Como mostrado nos capítulos anteriores, o desenvolvimento de *software* para a área de HRTCS é complexo e demanda muitas iterações, desde suas etapas iniciais de *design* até sua aceitação final. A aplicação de um nível de abstração, através de um arcabouço, composto por componentes específicos do domínio da aplicação, compartimentados, parametrizáveis, reutilizáveis, parece ser uma possível solução. Outros trabalhos desse tipo, em áreas correlatas, apresentam resultados expressivos.

Alguns produtos de mercado possuem ambientes de apoio com *frameworks* e plataformas de desenvolvimento, alguns inclusive, direcionados às áreas de sistemas embarcados, controle, automação e sistemas de potência. Entretanto, esses ambientes apresentam várias restrições que impedem sua ampla utilização como uma ferramenta definitiva para o desenvolvimento dos HRTCS's.

Esse capítulo detalha a criação de um arcabouço, especializado para essa área, que tenta preencher essas e outras lacunas do processo de desenvolvimento de *software* desses sistemas.

4.1 Características

O arcabouço mostrado nesse trabalho, consiste em uma ferramenta de apoio ao projeto e desenvolvimento de aplicações para HRTCS's. Suas principais características são:

- Separar o domínio da aplicação (aplicações em tempo real em sistemas de potência) do restante da plataforma de *hardware* (eletrônica, interfaces de entrada, saída e comunicação) e *software* (sistemas operacionais, *drivers*, escalonadores, etc.);
- Permitir que o engenheiro possa desenvolver a aplicação independentemente de outros detalhes mais intrínsecos das plataformas;
- Permitir a montagem da aplicação através de uma linguagem de alto nível, de forma

a constituir o conjunto necessário de tarefas de controle, automação, lógicas combinatórias e sequenciais e outros algoritmos;

- Constituir a aplicação com base em uma biblioteca de componentes menores, compartimentados, parametrizáveis, reutilizáveis e independentes das plataformas;
- Prototipar, testar e depurar a aplicação, sem os requisitos de desempenho, em quaisquer plataformas, antes de sua implantação em cenários reais;
- Permitir a modificação dos algoritmos e parâmetros da aplicação com facilidade;
- Permitir a inclusão de novos componentes à biblioteca;
- Possuir portabilidade a outras plataformas, mantendo a compatibilidade e previsibilidade dos resultados;
- Permitir obter elevados graus de desempenho e determinismo em tempo real;
- Resultar em baixa ocupação de memória RAM e ROM, favorecendo sua utilização com dispositivos microcontrolados;
- Poder ser utilizada em sistemas com grande desempenho computacional e grande disponibilidade de recursos;
- Poder utilizar recursos de *hardware* e *software* específicos de algumas arquiteturas, se necessário;
- Fazer uso de recursos de comunicação para implementação de processamento distribuído;
- Ser um ambiente completamente aberto e compartilhado com o público para receber futuras contribuições.

4.2 Cenário de utilização

Os cenários vislumbrados para utilização desse arcabouço são aqueles onde existem claramente:

- Uma plataforma de *hardware* já definida, contendo um núcleo microprocessado, memórias, entradas, saídas e canais de comunicação;
- Uma plataforma básica de *software* já definida, contendo um Sistema Operacional ou Executivo e um HAL;

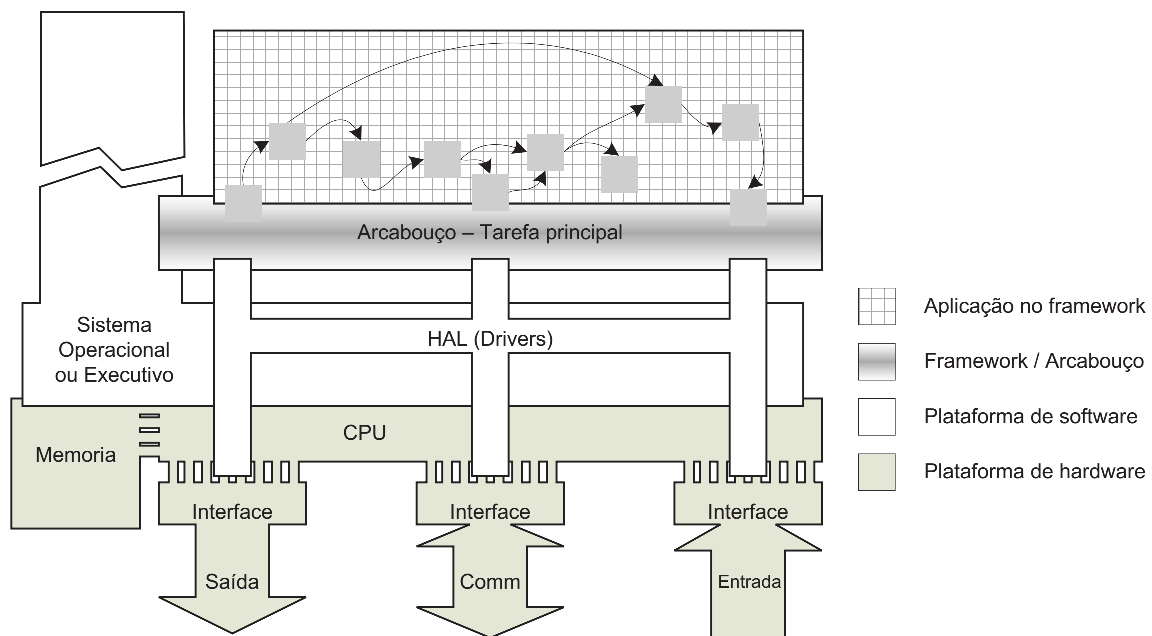

Figura 5: Plataforma de *hardware*, *software* e a parte embarcada do arcabouço em HRTCS

- Tarefas de *software* a serem desenvolvidas responsáveis pelo processamento de um fluxo de dados, ou seja, coletar os dados de entradas, processá-los e apresentá-los nas saídas do HRTCS.

O arcabouço é o conjunto de ferramentas de *software* que permite auxiliar o desenvolvimento dessas tarefas de fluxo de dados. Ele é composto por vários programas, rotinas e metodologias. Algumas são usadas no ambiente de desenvolvimento da aplicação, enquanto que outras são usadas diretamente no HRTCS. Por exemplo, uma visão de um HRTCS, contendo suas plataformas e a parte do arcabouço responsável por executar uma aplicação desenvolvida, pode ser vista na Fig.5.

4.3 Valores agregados de outras áreas

Na criação do *framework*, inicialmente foram buscados problemas semelhantes e outros precedentes na técnica a fim de se definir, com mais detalhes, como seria estabelecida a estrutura desse *framework*, como preconizado por (RIEHLE, 2000).

4.3.1 Síntese de *hardware* de alto nível

Notou-se que o paradigma de desenvolvimento com a associação de componentes reutilizáveis já existe em outras áreas. Um exemplo é a síntese comportamental de *hardware*

de alto nível, usado para a criação de ASIC's a partir de dispositivos do tipo FPGA, como comentado anteriormente e mostrado em detalhes por (GAJSKI et al., 1992).

Nesse tipo de abordagem, o engenheiro realiza uma descrição do comportamento do circuito digital, utilizando linguagens como o *VHDL* e *Verilog*, segundo as necessidades da aplicação. A partir de uma especificação, é realizada a descrição das relações combinatórias e sequenciais entre as entradas e saídas digitais de um circuito integrado hipotético. Essa descrição gera um arquivo, que é submetido a um processo de síntese, dando origem a uma arquitetura de *hardware*. Essa arquitetura pode então ser produzida de forma seriada (pelas *silicon foundries*) ou então transferida para um dispositivo generalizado programável, como uma FPGA, que passa então a funcionar com aquele comportamento desejado.

4.3.1.1 Detalhes da síntese de *hardware*

O processo de síntese de *hardware* para ASIC's é complexo, sendo composto de várias etapas desde a descrição comportamental até a criação da arquitetura final de *hardware*. Cada etapa é realizada dentro de um conjunto de programas externos. Tais etapas (GAJSKI et al., 1992) são itemizadas e descritas de forma simplificada na Fig. 6 e comentadas a seguir.

- Processamento léxico;
- Otimização algorítmica;
- Análise do fluxo de dados;
- Aplicação de restrições de arquitetura;
- Processamento das bibliotecas de componentes;
- Alocação de recursos;
- Escalonamento e agendamento de execução;
- Mapeamento de registradores;
- Processamento das saídas;

O processamento léxico faz a leitura do arquivo de descrição comportamental e verifica a sintaxe e a semântica da linguagem e suas construções. Em caso de sucesso, o arquivo de entrada é convertido em um formato intermediário, livre de comentários e informações desnecessárias.

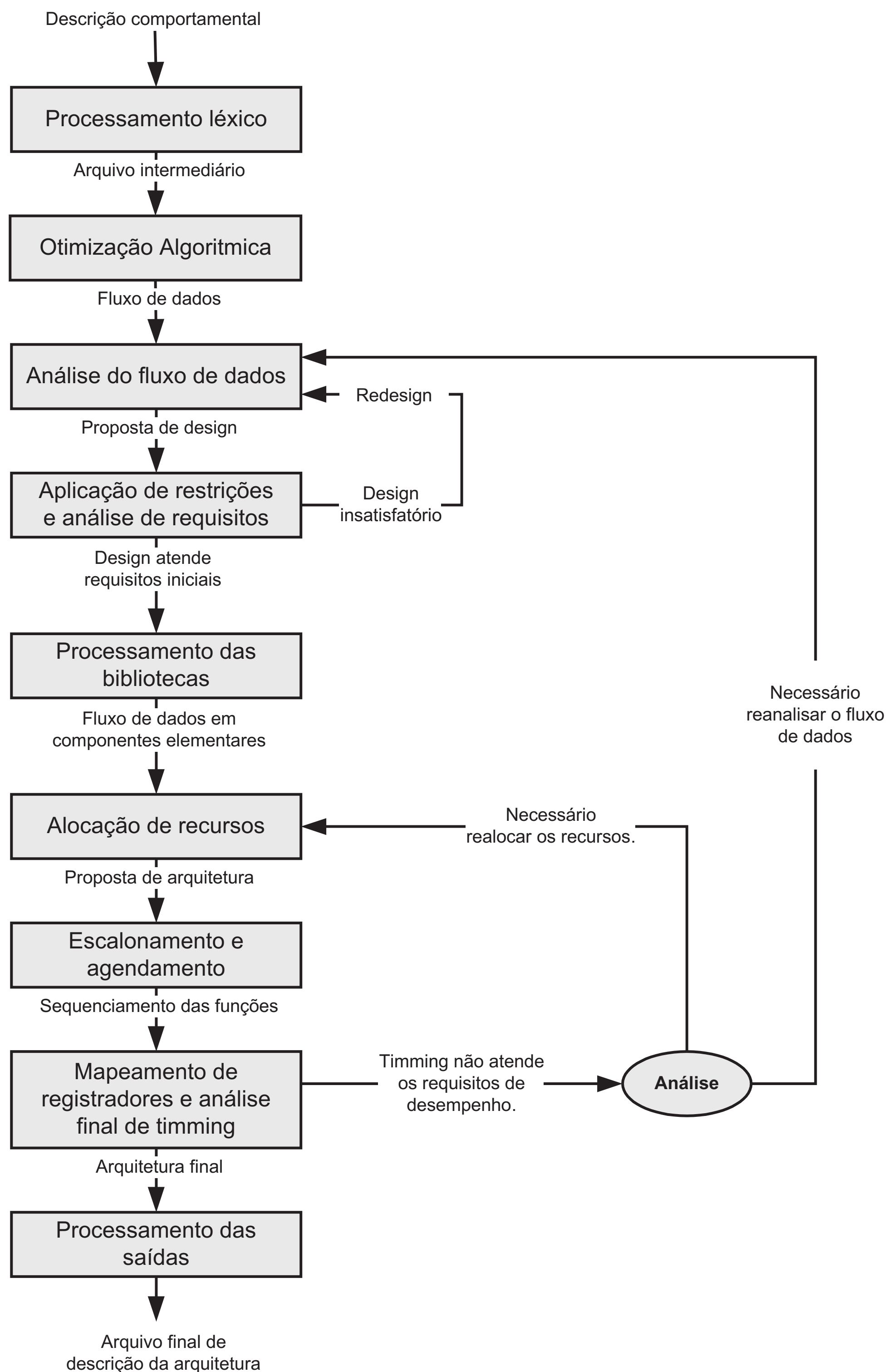

Figura 6: Processo simplificado de síntese de *hardware* a partir de uma descrição comportamental de um circuito.

A etapa de otimização algorítmica inicia a interpretação do arquivo intermediário, a fim de localizar trechos inconsistentes e violações de conceitos, tais como: componentes que estão fora do fluxo de dados entre as entradas e saídas, componentes associados de forma errada (saída ligada com saída), etc.

A análise do fluxo de dados verifica a consistência dos dados entre as entradas e as saídas, analisando as larguras de barramentos, tipos de dados, etc.. A seguir, é realizado um *design* preliminar do *hardware* e do caminho de dados e controles entre as entradas e saídas. Nesse ponto também já é possível analisar as concorrências e potenciais paralelismos de processos dentro do fluxo de dados.

A etapa de aplicação de restrições analisa se o *design* criado até o momento ainda é tangível em termos: de componentes utilizados, de limitações temporais (se o *timming* será adequado), de consumo de energia, etc. Caso o *design* seja insatisfatório, o processo anterior de análise será executado novamente, de forma iterativa, para propor um novo *design*. Esse processo é realizado continuamente até que seja encontrada uma solução ou até que ela se prove impossível.

Uma vez que o projeto proposto seja realizável, as bibliotecas de macro-componentes são lidas e incorporadas ao *design* existente, gerando uma arquitetura contendo apenas os componentes elementares da tecnologia.

Em seguida é feita a alocação dos recursos utilizados no *design* aos recursos presentes no circuito integrado generalista (FPGA). Nesse momento, por exemplo, uma determinada lógica de um "somador digital", presente no código comportamental, será associado a um determinado conjunto de componentes presentes em um dado setor da pastilha.

Adiante, no escalonamento, é feito um esquema de controle dos blocos funcionais alocados e a adição de registradores intermediários. O objetivo é controlar o momento em que os blocos funcionais e os registradores serão acionados ao longo do tempo (*clock*), para que os dados fluam (sejam processados) conforme o comportamento esperado.

Na etapa de mapeamento de registradores são estabelecidas as ligações entre os componentes e a verificação dos requisitos de tempo do fluxo de dados. Caso exista alguma violação das premissas iniciais ou das restrições impostas, o processo é reiniciado, com outras premissas, a partir de passos anteriores, até que uma solução de arquitetura seja encontrada.

A etapa de processamento da saída realiza a junção das arquiteturas resultantes do fluxo de dados, fluxo de controles e outras máquinas de estado resultantes em um único arquivo. Esse arquivo contém todas as informações para tornar um determinado *chip* programável generalizado, tipo FPGA, no circuito ASIC desejado.

Esse tipo de síntese apresenta um espaço de soluções explosivo. Por essa razão, como exemplificado em algumas etapas, os processos de síntese são iterativos, utilizando uma determinada heurística para encontrar a melhor solução de arquitetura. O processo pode levar de minutos a horas para ser completado.

4.3.1.2 Inspirações para o *framework*

Essencialmente, as funcionalidades existentes no processo de síntese de *hardware* também podem ser aplicadas ao *framework* desse trabalho.

Analisando cada etapa anterior é possível notar que também no *framework*:

- As relações entre componentes elementares podem ser descritas através de uma linguagem, metalinguagem ou metodologia de alto nível;
- A linguagem deve ser compilada e analisada quanto a sua semântica e quanto à violação das regras para constituição de seus algoritmos e lógicas;
- É importante que exista uma biblioteca interna de componentes primitivos, implementados para execução nas CPU's com o melhor desempenho possível;
- É importante uma biblioteca de macrocomponentes, elaborados pelos usuários através da associação dos componentes menores;
- Durante o processo de interpretação e análise, os macrocomponentes podem ser substituídos por seus conjuntos de componentes primitivos;
- Os fluxos de dados devem ser analisados, uma vez que as entradas e as saídas podem envolver informações de naturezas diferentes (valores booleanos, valores numéricos) e que devem estar associadas aos componentes de forma consistente;
- Algumas plataformas onde a aplicação será implantada possuem restrições de memória e tempo de execução que podem fazer com que as especificações originais sejam violadas;
- Algumas arquiteturas possuem mais de um núcleo de processamento ou outros recursos especiais. Esses podem ser usados para otimizar os algoritmos;
- Uma vez definido todo o fluxo de informações entre as entradas e as saídas, os componentes e suas estruturas de dados internos podem ser alocados estaticamente na memória da plataforma;
- Os componentes devem ser executados na ordem correta, definida pelo fluxo de dados e pelas regras de álgebra dos componentes;

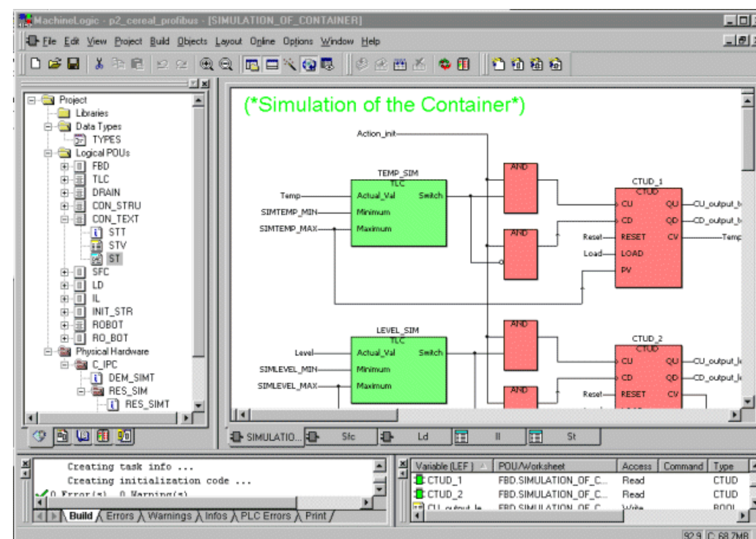

Figura 7: Tela do *software* MachineLogic com recursos de programação da linguagem “FBD”.

- Todo o escopo de dados e instruções devem ser armazenados em um arquivo para posteriormente serem transmitidos e interpretados pela plataforma de *hardware*.

Devido a essas semelhanças, as ferramentas desenvolvidas foram diretamente inspiradas naquelas existentes para a síntese de *hardware*.

4.3.2 Normas IEC 61131 e IEC 61499

As linguagens da norma IEC 61131-3 (IEC61131, 2003) são amplamente utilizadas na programação de PLCs e IED’s de automação e controle (FAUSTINO, 2005). Das várias linguagens, principalmente as linguagens de descrição gráficas FBD e SFC mostram-se intuitivas para representar os fluxos de dados entre as entradas e as saídas desses sistemas.

Um exemplo de interface gráfica utilizando os recursos da IEC 61131-3 para diagramas de bloco é mostrado na Fig.7, do pacote de *software* “MachineLogic Development” da CTC (PARKER-HANNIFIN, 2010), subsidiária da empresa Parker Hannifin. Esse tipo de linguagem gráfica também pode ser vista aplicada com sucesso nos ambientes de desenvolvimento do MatLab Simulink (MATHWORKS, 2010), VisSim (VISSIM, 2010) e SciLab SciCOS (INRIA, 2010).

Existem várias vantagens para o uso desse tipo de enfoque. Primeiramente, essas linguagens exigem uma visão sistêmica da aplicação, ou seja, uma divisão lógica, em módulos bem definidos, de todo o domínio do problema. Isso favorece seu entendimento e permite o projeto de cada módulo de forma compartimentada. Esses argumentos são corroborados pela norma IEC 61499 (IEC61499, 2005) de desenvolvimento de sistemas

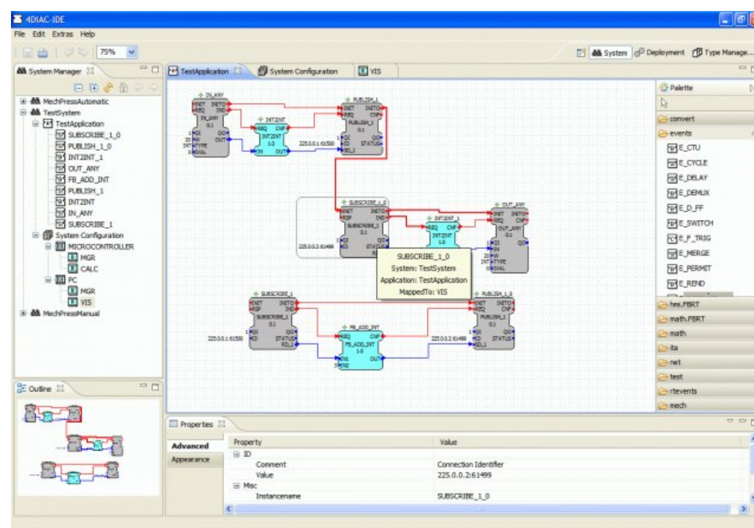

Figura 8: Tela do *software* 4DIAC IDE para *design* de aplicações de automação e controle segundo a IEC 61499.

distribuídos, onde as soluções de controle e automação também são representadas por blocos, conectados entre si, segundo o fluxo de dados e eventos do sistema, como pode ser visto na Fig.8.

Dados esses aspectos, decidiu-se utilizar esse tipo de visão sistêmica e sua representação gráfica como premissas básicas para o *framework* desse trabalho. Dessa forma, a descrição dos fluxos de dados será feita através de diagramas de blocos funcionais, interligados segundo um conjunto de regras. A descrição desses blocos e de suas ligações serão feitas através de uma metalinguagem, armazenada, a princípio, na forma de arquivos texto. No futuro poderá ser desenvolvida uma interface gráfica para manipulação dessas informações, como feito no (4DIAC, 2010).

4.3.3 Forma de processamento da metalinguagem

O fluxo de dados descrito através da metalinguagem de programação é um formato muito abstrato para ser utilizado diretamente por microprocessadores ou CPU's. Como qualquer linguagem de alto nível, ele precisa ser convertido ou traduzido para outro formato, semanticamente equivalente, que seja passível de ser interpretado e executado por uma plataforma computacional.

Como em outras linguagens modernas, esse processo de conversão do código, desde seu nível mais alto até o mais baixo e elementar, emprega uma série de ferramentas intermediárias, tais como: pré-processadores, compiladores, "linkeditores", interpretadores, etc.

Inspiradas nessas linguagens, o *framework* desse trabalho também utiliza ferramentas semelhantes.

4.3.4 Portabilidade

Existem várias linguagens disponíveis na técnica, que possuem, como premissa básica, a garantia de portabilidade transparente para outras plataformas, tais como as linguagens Java e Python.

Para atingir esse objetivo, essas linguagens adotam uma estrutura composta por uma máquina virtual para execução de seus programas. Essa máquina virtual emula todos os detalhes de uma plataforma real, de uma forma simplificada e padronizada, por exemplo: o gerenciamento de memória, o compartilhamento de recursos, etc.

Dessa forma, todas as incompatibilidades entre *hardware* diferentes, permanecem ocultas, permitindo que um mesmo código seja executado em qualquer outro local onde aquela máquina virtual tenha sido implementada.

Conceitualmente, a questão da portabilidade da aplicação não é plenamente resolvida com o uso de máquinas virtuais. Na realidade, ela só é transferida de lugar, já que, ao invés do usuário se preocupar em portar a aplicação entre plataformas, o usuário passa a se preocupar em portar a máquina virtual.

Apesar de parecerem problemas semelhantes, deve-se notar que o trabalho de adaptação da máquina virtual é feito uma vez, somente quando ocorre uma mudança na plataforma. Quando ocorre uma mudança no *software* da aplicação, não é necessário modificar nada nessa estrutura. Da outra forma, sem a máquina virtual, toda a vez que a plataforma é modificada ou toda a vez que a aplicação é alterada, ocorrem retrabalhos nas outras partes do sistema.

Nesse trabalho, não há necessidade de se implementar uma máquina virtual completa, uma vez que muitos detalhes já são definidos para as tarefas da aplicação, como por exemplo: as interfaces de entrada e saída, o espaço de memória, etc. Nesse caso, é implementado somente um núcleo operacional, denominado "interpretador", com algumas otimizações com o intuito de favorecer seu desempenho.

4.3.5 Linguagem de implementação das ferramentas

As ferramentas que constituem o *framework* (compiladores, interpretadores, etc.) precisam ser escritos em uma dada linguagem de programação de alto nível. Essa linguagem de implementação deve possuir os seguintes requisitos principais:

- Possuir compiladores para as diversas plataformas de interesse (preferencialmente gratuitos)[1];
- Permitir criar programas com ótimo desempenho, e;
- Possuir recursos para acesso direto a detalhes intrínsecos do *hardware* e do Sistema Operacional/Executivo e HAL.

Como mostrado no Capítulo 3, a linguagem que satisfaz plenamente os requisitos acima é a linguagem C, padrão ANSI, sendo utilizada como base para construção de todas as ferramentas desse trabalho.

4.3.6 Experiência de outros trabalhos

Todas as informações pesquisadas e mostradas nos Capítulos 2 e 3 foram utilizadas como experiência para o desenvolvimento desse arcabouço. Entretanto, alguns pontos encontrados nesses trabalhos foram fundamentais para definir a direção desse desenvolvimento. Esses pontos são:

- O uso de plataformas de *hardware* e *software* apontado por (SANGIOVANNI-VINCENTELLI; MARTIN, 2001);
- A estratégia de *design* "top-down" mostrada por (FRöHLICH, 2001) para uma definição mais racional das funcionalidades do sistema operacional e da plataforma de *hardware*;
- A carência e a necessidade de *frameworks* para a área de Sistemas de Potência, mostrado por (LI, 2001);
- A facilidade de *design* e os resultados do uso de *frameworks*, como mostrado por (NETO et al., 2010);
- As vantagens de desenvolvimento com *frameworks versus* seus impactos em desempenho, como mostrado por (GUO; EDWARDS; BOROJEVIC, 2008);

4.4 Desenvolvimento

A obra de (ALMEIDA et al., 2007) cita uma série de metodologias para o reúso de códigos, conforme os preceitos da Engenharia de *Software*, incluindo os conceitos que

[1] Uma vez que todo o restante desse trabalho se apoia em soluções de domínio público, software livre ou software gratuito.

definem e caracterizam os *frameworks* de apoio. Segundo esses conceitos, o arcabouço desenvolvido nesse trabalho, denominado "TB" (de *TransBlock*), pode ser categorizado como sendo um *framework*:

- De *software*, ou seja, de apoio a programação;
- Do tipo vertical e horizontal, ou seja, com elementos específicos do domínio da aplicação e elementos que também podem ser encontrados em outros domínios, e;
- Especialista, ou seja, aplicado especificamente, nessa obra, aos cenários com desempenho em tempo real do domínio de Sistemas de Potência.

Esse *framework* é constituído por:

- Uma biblioteca de blocos *template* (modelo) para as aplicações em Sistemas de Potência;
- Uma metalinguagem de descrição do fluxo de dados;
- Uma ferramenta de pré-processamento;
- Uma ferramenta de compilação, e;
- Uma ferramenta de interpretação em tempo de execução;

Basicamente, o usuário tem uma ideia de algoritmo para uma determinada aplicação embarcada. Essa ideia é estruturada na forma de um diagrama de blocos e descrita através da metalinguagem do *framework*. Com o apoio das ferramentas de pré-processamento e compilação, o usuário pode gerar um código-objeto e testá-lo em quaisquer plataformas onde foi previamente portado e implementado o interpretador. Os resultados obtidos independem da plataforma, assim como o processo de desenvolvimento e depuração. Dessa forma o usuário pode escolher desenvolver seu *software* o máximo do tempo em um ambiente confortável, como seu *Desktop*, e apenas realizar o teste, na plataforma final desejada, quando o algoritmo já estiver completamente maduro e livre de problemas. Esse esquema de funcionamento pode ser visto na Fig.9.

Cada um desses componentes é explicado com mais detalhes a seguir.

4.4.1 Biblioteca de blocos *template*

O arcabouço é constituído por uma coleção de blocos funcionais, implementados internamente, formando uma biblioteca de recursos para utilização pelo usuário.

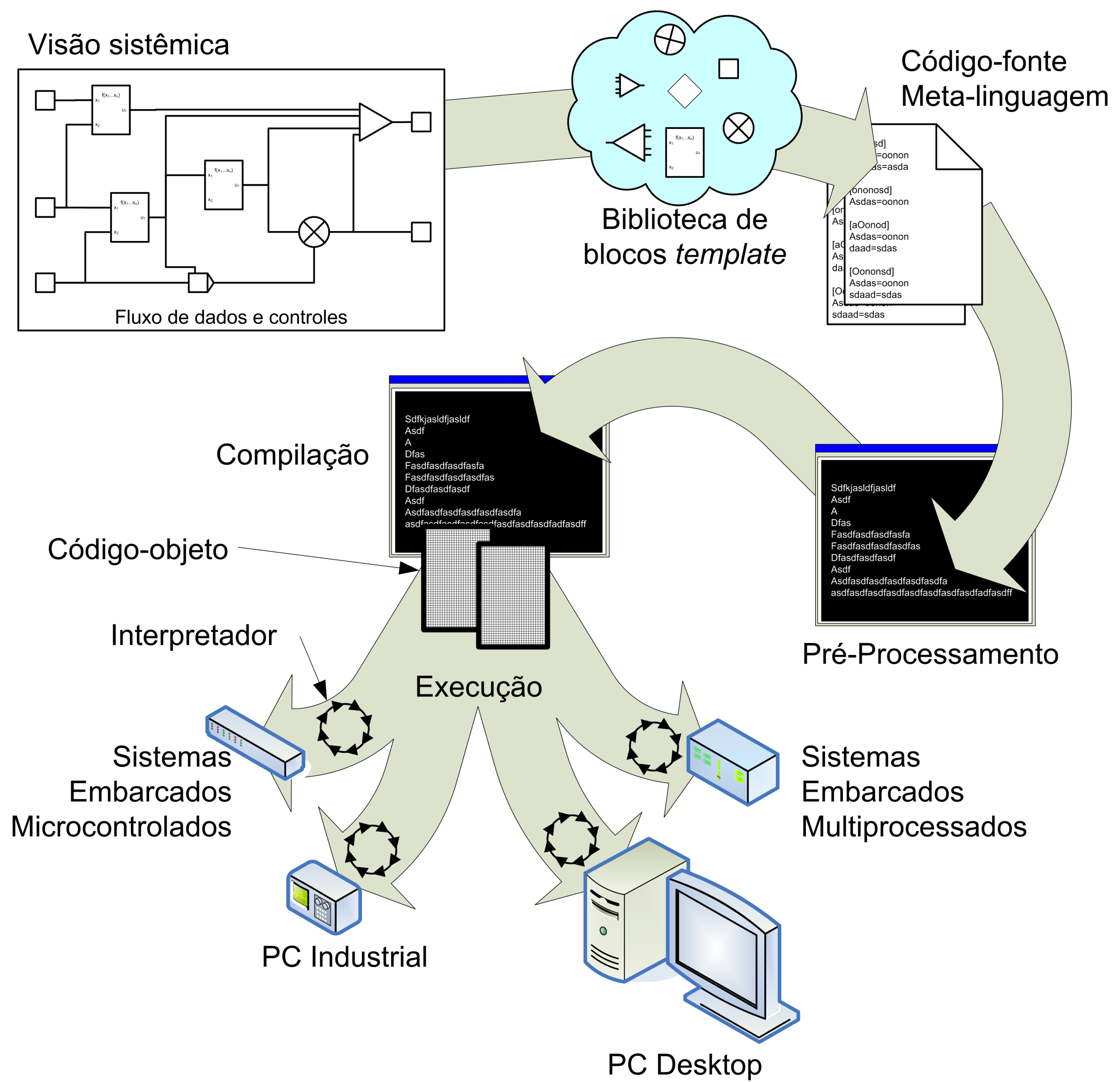

Figura 9: Funcionamento básico do *framework* "TB", de apoio ao desenvolvimento de um HRTCS para Sistemas de Potência.

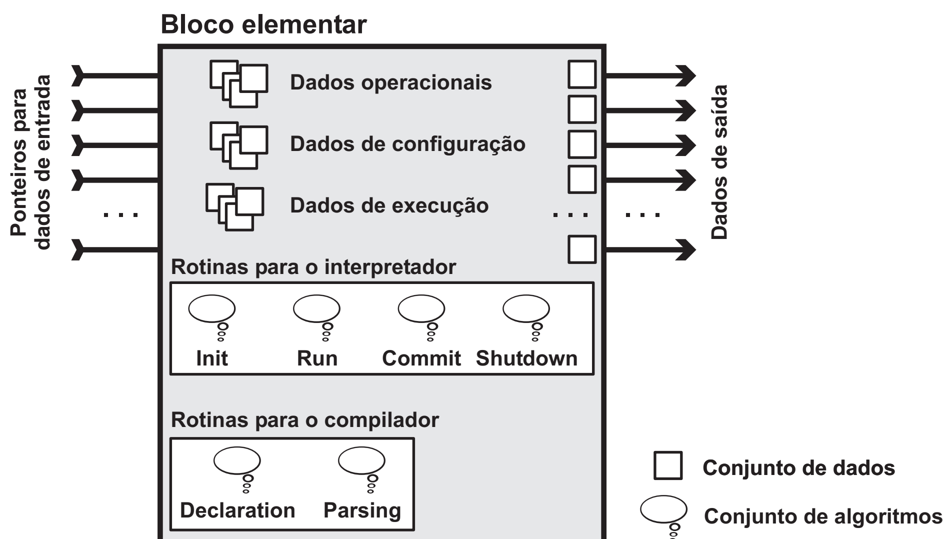

Figura 10: Bloco elementar, generalizado, do “TB”.

A funcionalidade desempenhada por cada bloco *template* pode ser primitiva ou complexa. Blocos primitivos implementam somente operações algébricas e lógicas simples, tais como: soma, subtração, lógica E, lógica OU, etc. Blocos complexos implementam operações e algoritmos mais elaborados, tais como: integração numérica, controlador PID, filtro-digital, limitador de gradiente, banda de histerese, etc. A quantidade de recursos dessa biblioteca é determinada pelo domínio da aplicação. A quantidade de blocos implementados dentro do interpretador depende das capacidades da plataforma onde esse está sendo executado, como será visto adiante.

4.4.2 Estrutura de um bloco *template*

A estrutura de cada um desses blocos é padronizada, sendo composta por entradas, saídas, parâmetros e sub-rotinas, agrupados em um conjunto de dados e um conjunto de algoritmos, como pode ser visto na Fig.10.

Pode-se notar na Fig.10, que cada bloco pode possuir um número qualquer de entradas, saídas e dados internos, conforme a necessidade.

Cabe ressaltar que alguns blocos podem não possuir entradas ou saída explícitas, como são os casos das interface de sinais com o mundo exterior, geradores de sinais e constantes, blocos de anotação de grandezas e oscilografia, etc.

Detalhes específicos de cada tipo e suas funcionalidades serão mostrados adiante.

4.4.2.1 Conjunto de dados

Como mostrado na Fig. 10, o conjunto de informações que constitui um bloco é composto de:

- Dados de saída;
- Dados de entrada;
- Dados de configuração;
- Dados de execução (*runtime*), e;
- Dados operacionais.

Os dados de saída são posições fixas de memória dentro do conjunto de dados. Os dados de entrada, por outro lado, são representados por ponteiros, que armazenam os endereços onde as informações devem ser buscadas na memória. Esse tipo de abordagem possui duas vantagens. Primeiramente, já que, em uma dada instância, ponteiros só apontam para um único lugar[2], uma entrada de um bloco só poderá se referir a uma única saída. Em segundo lugar, duas saídas nunca poderão ser ligadas entre si. Assim, são evitadas possíveis incongruências na criação da aplicação.

Os dados de configuração são variáveis utilizadas pelo bloco para armazenar sua parametrização. Essas informações são obtidas no momento da execução do compilador, que faz a leitura do arquivo de entrada e captura os ajustes designados pelo usuário.

Os dados de execução (*runtime*) são variáveis usadas internamente pelos algoritmos de processamento, durante a etapa de execução e interpretação do arcabouço.

Finalmente, os dados operacionais são variáveis utilizadas pelo compilador e interpretador durante a leitura do arquivo de entrada e durante a preparação de sua execução. Esses dados armazenam a identificação de cada bloco, o tamanho de sua área de memória, seu tipo, número de entradas e saídas, etc.

Todos os tipos de variáveis que conformam esse conjunto foram racionalizados para apenas quatro tipos possíveis, a saber:

- Variáveis analógicas;
- Variáveis digitais;
- Ponteiros para variáveis analógicas;

[2] Ainda bem!

- Ponteiros para variáveis digitais;

A natureza e a precisão desses tipos de variáveis podem ser ajustados no momento de criação do *framework*, de acordo com a disponibilidade da arquitetura e de acordo com a precisão e o desempenho desejados para o sistema.

Por exemplo, para aproveitar os recursos de uma plataforma com microprocessador de 32 bits, padrão Intel "x86", as variáveis analógicas do arcabouço podem ser feitas com natureza de ponto flutuante e precisão dupla (64 bits), e as variáveis digitais com natureza inteira e precisão dupla (32 bits). Entretanto, no caso de um DSP de 16 bits, padrão Analog Devices "ADSP21XX" (ANALOG, 2010), seu desempenho seria melhor aproveitado se as variáveis analógicas fossem feitas com natureza de ponto fixo e precisão simples (16 bits), e suas variáveis digitais com natureza inteira e precisão simples (16 bits).

De qualquer forma, seria possível utilizar um *framework* desenhado para a arquitetura "x86", até mesmo em uma plataforma mais limitada, como é o caso de um microcontrolador de 8 bits, sem capacidade de ponto flutuante, por exemplo. Nesse caso, o sistema irá funcionar, mas com uma expressiva penalidade no desempenho computacional, além de um grande aumento no tamanho do programa interpretador, uma vez que nesse, as operações de 64 bits terão de ser feitas como uma cadeia de várias operações de 8 bits.

Deve-se atentar que uma mudança na natureza e precisão das variáveis, também ocasiona mudanças nas aritméticas dos algoritmos internos de cada bloco, mostrados a seguir.

4.4.2.2 Conjunto de algoritmos

Todo bloco *template* possui rotinas básicas, criadas e desenvolvidas no momento de sua gênese. Algumas dessas rotinas são usadas no momento da compilação, enquanto que outras são usadas no momento da interpretação do código-objeto. Tais rotinas, como podem ser vistas na Fig. 10, são:

- Rotinas do compilador:
  - Rotina de declaração do bloco (*Declaration*), e;
  - Rotina de leitura de configurações (*Parsing*);
- Rotinas do interpretador:
  - Rotina de inicialização (*Init*);
  - Rotina de execução (*Run*);

- Rotina de pós-execução (*Commit*), e;
- Rotina de finalização (*Shutdown*).

A rotina de declaração instrui o próprio compilador a reconhecer aquele bloco como um dos blocos *template* de sua biblioteca. Essa rotina cadastra no compilador as informações dos dados operacionais do bloco, tais como: sua identificação interna, o número de entradas, número de saídas e, principalmente, se o bloco se trata de um elemento puramente algébrico (ou combinatório) ou então, de um elemento com ao menos um estado interno (bloco sequencial, com memória). O motivo dessa diferenciação é mostrado adiante.

A rotina de *Parsing* é executada no momento da compilação, durante a leitura do arquivo de entrada. Essa rotina lê as configurações e parâmetros ajustados pelo usuário para um determinado bloco, armazenando as informações na memória do compilador.

Cada bloco utilizado pelo usuário possui um rotina de inicialização (*Init*). Essa rotina é executada pelo interpretador antes da primeira iteração do sistema. O objetivo é preparar os dados internos de cada elemento, adequadamente, por exemplo, ajustando as condições iniciais de um bloco integrador numérico, zerando os *buffers* de memória em um bloco de filtragem digital, etc. A ordem de execução dessas rotinas de inicialização, ou seu tempo de execução, não são relevantes para o funcionamento ou desempenho do sistema.

A rotina de execução (*Run*) de cada bloco é executada pelo interpretador durante todo o período de funcionamento do *arcabouço*. Essa rotina é o coração de cada componente, sendo responsável, em cada iteração, por processar as informações das entradas, os dados internos e de configuração, para constituir os novos valores das saída. A decisão sobre a ordem de disparo dessas rotinas é extremamente relevante para o correto funcionamento do sistema. O tempo de execução dessas rotinas também é relevante para o desempenho global do sistema.

A rotina de pós-execução de cada bloco (*Commit*) também é executada pelo interpretador continuamente. Essa rotina não possui ordem correta para ser executada entre as demais. A única condição é que elas sejam executadas ao final de todas as rotinas *Run*. Essa rotina foi criada por duas razões:

- Para gerenciar semáforos, no caso de execução do arcabouço em ambientes de processamento paralelo simétrico, com mais de uma CPU;
- Para sincronizar a operação de atualização das saídas de todos os blocos sequenciais presentes no sistema, para que todos atualizem seus valores APÓS todos os cálculos algébricos terem sido efetuados.

Esses mecanismos serão explicados adiante.

A rotina de finalização (*Shutdown*) de cada bloco é executada no momento do término das atividades do arcabouço. Contudo, em sistemas embarcados, essa rotina nunca é executada, tendo em vista que, em condições normais, esses sistemas permanecem sob funcionamento ininterrupto.

Deve-se ressaltar, que o interpretador pode ser usado também fora de um ambiente embarcado, como nos casos de desenvolvimento, testes e simulações dos algoritmos. Nesse caso, a simulação pode ser executada e interrompida pelo usuário a qualquer momento, ser executada passo a passo, etc. Alguns blocos *template* são dedicados para essa operação de depuração, como é o caso do bloco de oscilografia e do bloco de registro de eventos digitais. Esses blocos armazenam dados durante a execução da simulação. Ao término, suas rotinas de finalização (*shutdown*) são executadas, e as informações coletadas são gravadas em disco ou apresentadas na forma de gráficos e tabelas para o usuário.

É importante ressaltar que, das funções comentadas acima, as duas rotinas do compilador e a rotina de inicialização para o interpretador são mandatórias para implementação em qualquer tipo de bloco. As demais, só são implementadas conforme a necessidade. Por exemplo, para um bloco "somador", não há a necessidade de implementação de uma rotina de pós-execução ou de uma rotina de finalização.

Um exemplo de como essas rotinas são criadas em linguagem ANSI C é mostrado no Apêndice A.

4.4.3 Metalinguagem de descrição

O usuário do arcabouço deve desenhar sua aplicação utilizando os blocos *template* disponibilizados, em um esquema idêntico ao preconizado para a linguagem FBD da IEC 61131-3 (IEC61131, 2003). Esse processo pode ser feito de forma visual, através de uma interface gráfica de alto nível (que não foi contemplada nesse trabalho) ou através de uma descrição textual.

4.4.3.1 Formato do arquivo

O formato e a sintaxe utilizadas no arquivo de descrição possui uma estrutura muito semelhante a dos arquivos padrão `[.INI]`, utilizados no sistema operacional Microsoft Windows (MICROSOFT, 2010), ou dos arquivos formato `[.RC]` e `[.CONF]`, utilizados nos sistemas operacionais padrão Unix.

De uma forma geral, o arquivo possui várias seções, para cada bloco *template* utilizado no algoritmo, com a seguinte sintaxe:

```
[<nome_do_bloco_template>]
```

Em seguida à declaração desse nome de seção, existem parâmetros específicos para cada bloco, tais como: identificação alfanumérica (`Name=<nome_fantasia_do_bloco>`), parâmetros atribuídos a valores (`Gain=<valor>`), etc. Os nomes das entradas, saídas e dos parâmetros internos dos blocos são definidos pelo programador no momento da criação do *template*.

O arquivo também possui comentários de linha e bloco, inseridos pelo usuário por meio dos caracteres "#" e ";", para auxiliar no processo de descrição e documentação. Um exemplo desse tipo de arquivo pode ser visto no Apêndice B.

4.4.3.2 Descrição do algoritmo da aplicação

Como ilustrado na Fig.9, a partir de uma visão sistêmica do algoritmo, o usuário deve dispor os blocos funcionais necessários para estabelecer o fluxo de dados entre as entradas e as saídas do sistema de acordo com a disponibilidade da biblioteca de blocos *template* e de outros macro-blocos.

Parâmetros internos

Cada bloco é parametrizado com suas respectivas propriedades de ganho, constantes de tempo, coeficientes, etc. Deve-se atentar que tais parâmetros devem estar dimensionalmente coerentes às unidades utilizadas para representar os dados nas entradas e saídas. Por exemplo, a aquisição de um determinado valor de tensão do mundo exterior, feita por um conversor A/D, será representada no ambiente do arcabouço, não em sua unidade de engenharia real, mas sim condicionada, por um fator de conversão, devido ao processo de quantificação do conversor A/D. O usuário desenvolvedor do algoritmo precisa estar ciente desses fatores de conversão, para ajustar a forma de processamento dos dados.

Relacionamentos

O relacionamento entre os blocos é feito através da atribuição das saídas de um elemento às entradas de outro. Assim, segundo o formato do arquivo mostrado anteriormente, para que um bloco "B" tenha sua entrada ligada à saída de um bloco "A", dentro da seção que descreve "B", deve haver uma citação dessa ligação no seguinte formato:

```
<entrada_do_bloco_B> = <nome_fantasia_do_bloco_A>.<nome_da_saída_do_bloco_A>
```

Por exemplo:

```
1 [BLOCO_TEMPLATE_EXEMPLO]
  Name    = Bloco_B
3 Input_0 = Bloco_A.IntegratorOutput
```

Isso pode ser visto com mais detalhes no Apêndice B.

Regras de constituição

Para que o fluxo de dados criado pelo usuário seja possível de ser executado conforme um sistema de controle descrito na forma de um diagrama de blocos (OGATA, 1998) , existem algumas premissas elementares:

- Dois blocos não podem ter suas saídas ligadas entre si;
- Uma entrada de um bloco não pode ser deixada desconectada;
- Não podem ocorrer *loops* algébricos envolvendo qualquer conjunto de blocos, ou seja, dado o grafo de conectividade que representa os blocos e suas ligações entre entradas e saída, nessa grafo não pode haver nenhuma malha que apresente em seu enlace apenas blocos puramente algébricos.

As violações dessas regras são checadas durante as etapas de pré-processamento e compilação da metalinguagem.

4.4.3.3 Recursos adicionais

Essa metalinguagem foi elaborada com extensões que também permitem:

- Incluir arquivos externos dentro do arquivo principal, como o mecanismo de "`#include`" da linguagem ANSI C, para agregar trechos de blocos de uso frequente, por exemplo;
- Definir símbolos globais que podem ser associados a propriedades e parâmetros dos blocos, por exemplo, `DEF_G = 9.8`, e;
- Definir macro-blocos, ou seja, o agrupamento de uma estrutura de vários blocos *template*, que se repetem por muitas vezes dentro do fluxo de dados, em um bloco maior, reutilizável.

4.4.4 Pré-processador

Segundo (NOERGAARD, 2005), o uso de um pré-processador é fundamental pois permite a montagem de construções complexas e sistemáticas no código-fonte, de forma automática.

O papel do pré-processador, segundo (KERNIGHAN; RITCHIE, 1978), é ler o código-fonte original, criando um novo código-fonte, reestruturado, com a adição e substituição de elementos, para posterior envio ao compilador. Nesse processo, alguns erros comuns de digitação, sintaxe e formatação dos dados podem ser percebidos pela ferramenta e sinalizados ao usuário.

Nesse trabalho, o funcionamento do pré-processador obedece aos seguintes passos:

1. Leitura e interpretação do arquivo texto de entrada;
2. Leitura e inclusão de outros arquivos indicados pelo usuário;
3. Identificação dos blocos *template* utilizados pelo usuário;
4. Identificação e aprendizado dos macro-blocos utilizados pelo usuário;
5. Identificação e aprendizado de símbolos e definições feitas pelo usuário ao longo de todo o código-fonte;
6. Escrita de um novo arquivo de entrada, sem comentários, contendo apenas o código-fonte dos blocos e as expansões de macro-códigos e símbolos utilizados.

O mecanismo de expansão de macro-blocos e símbolos consiste em localizar no código-fonte original as citações a esses elementos e processá-las. No caso de símbolos, é elaborado um dicionário interno com seus nomes e valores. No caso de macro-blocos, o pré-processador busca em disco o arquivo que define sua estrutura e assimila sua forma de construção como um conjunto de blocos *template*. Ao final desse processo de aprendizado, o pré-processador escreve um novo arquivo de código-fonte a partir do original, substituindo as citações dos símbolos pelos seus respectivos valores definidos, e expandindo as citações dos macro-blocos por suas estruturas de construção com blocos *template*. O processo é iterativo, pois existem alguns macro-blocos e símbolos que citam outros macro-blocos e símbolos, de forma aninhada. O pré-processador também detecta se existe recursividade infinita nesse processo de expansão, interrompendo o trabalho e exibindo os erros ao usuário.

O arquivo final é composto apenas de blocos da biblioteca interna, com seus parâmetros e associações de entradas à saídas, livre de comentários e símbolos.

4.4.5 Compilador

O compilador é uma peça fundamental do *framework*. É o trabalho dessa ferramenta que permite que a metalinguagem em alto nível, inteligível pelos usuários, seja convertida para um formato elementar, de baixo nível, capaz de ser interpretado pela plataforma computacional.

No início do desenvolvimento desse trabalho, para obter maior performance do código final, em detrimento à sua portabilidade, cogitou-se criar um compilador para realizar uma conversão total, ou seja, traduzir a linguagem de blocos do *framework* diretamente para uma linguagem intermediária em ANSI C. Essa linguagem intermediária poderia então ser compilada e "linkeditada" por outras ferramentas externas, para obtenção de um código de máquina puro. Nessa abordagem, o compilador é mais simples, consistindo basicamente de um pré-processador um pouco mais elaborado.

Esse tipo de solução é aquela utilizada em ambientes como o MatLab (MATHWORKS, 2010) e LabView (NATIONAL, 2010), para uso em plataformas proprietárias de *hardware*. Apesar de seus bons resultados, tais métodos dependem de outras ferramentas de compilação externas e bibliotecas, que muitas vezes não podem ser redistribuídas com um produto final. Além disso, nestes casos, quando surgem problemas durante os testes da aplicação, é comum ser levantada a questão se, o problema está originalmente no *design* do algoritmo original, ou na tradução desse para a linguagem intermediária. Cabe informar que esse código intermediário produzido é extremamente complicado de ser entendido e depurado.

A outra alternativa são os compiladores para pseudocódigo, como é o caso do compilador da linguagem Java (JAVA, 2010). Além das vantagens de portabilidade já citadas, não são necessárias ferramentas de terceiros ou linguagens intermediárias para seu funcionamento. O único impasse é que o projeto dessa ferramenta apresenta uma complexidade maior, semelhante aquela encontrada em um compilador de uma linguagem completa, como o ANSI C. Esse esquema ainda implica no desenvolvimento de uma máquina virtual ou interpretador, como já explicado.

Nesse trabalho, o compilador criado agrega as funções de conversão do código fonte para linguagem de baixo-nível e também as funções de "linkedição" e montagem do código-objeto final.

Desde a leitura do arquivo de entrada, até a escrita do código-objeto, são executadas as seguintes etapas, ilustradas na Fig.11, explicadas com mais detalhes adiante:

1. Leitura, processamento léxico do arquivo fonte gerado pelo pré-processador e interpretação (*parsing*) dos parâmetros dos blocos utilizados;

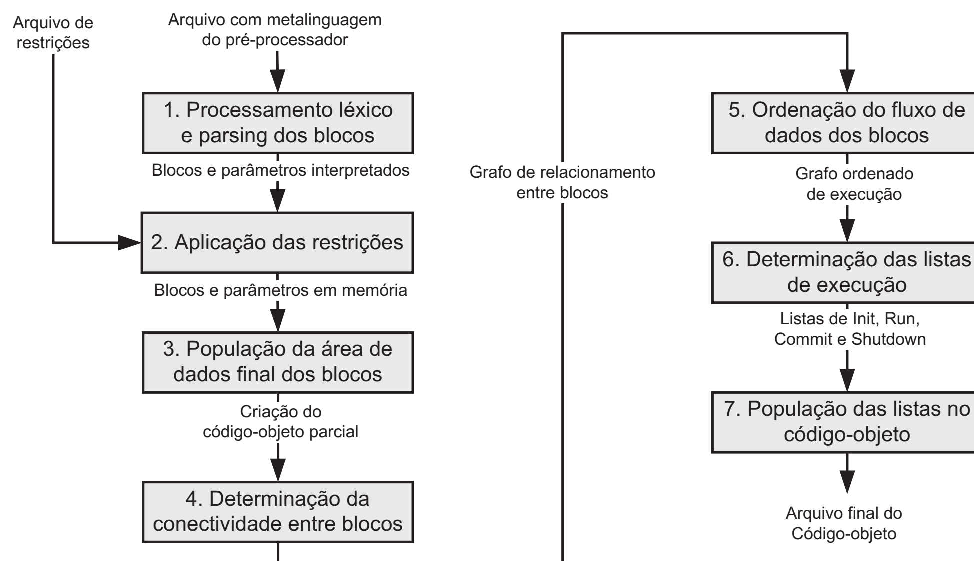

Figura 11: Funcionamento básico do compilador do "TB".

2. Leitura e aplicação do arquivo de restrições de *hardware*;

3. População da área de dados final com os blocos utilizados;

4. Determinação da conectividade entre os blocos;

5. Ordenação do fluxo de dados;

6. Determinação das listas de execução;

7. População da área de instruções e geração do código-objeto.

Para fornecer mais detalhes a respeito do funcionamento do compilador, o sistema dinâmico apresentado no Apêndice B foi compilado segundo essa metodologia. Além do arquivo de código-objeto final, o resultado visual dessa compilação é mostrado no Apêndice C.

As etapas de compilação citadas anteriormente para esse exemplo de sistema são descritas a seguir e, quando pertinente, são feitas referências aos resultados apresentados no Apêndice C.

Na etapa de leitura do arquivo gerado pelo pré-processador, o compilador faz a interpretação de todos os blocos *template* utilizados pelo usuário em seu algoritmo, armazenando-os internamente segundo uma numeração contínua, como mostrado na Fig.12. A saída visual desse processo pode ser observada no Apêndice C entre as linhas 4 a 10 da listagem. A numeração adotada para o blocos é mostrada logo abaixo nesse Apêndice, entre as linhas 28 a 39.

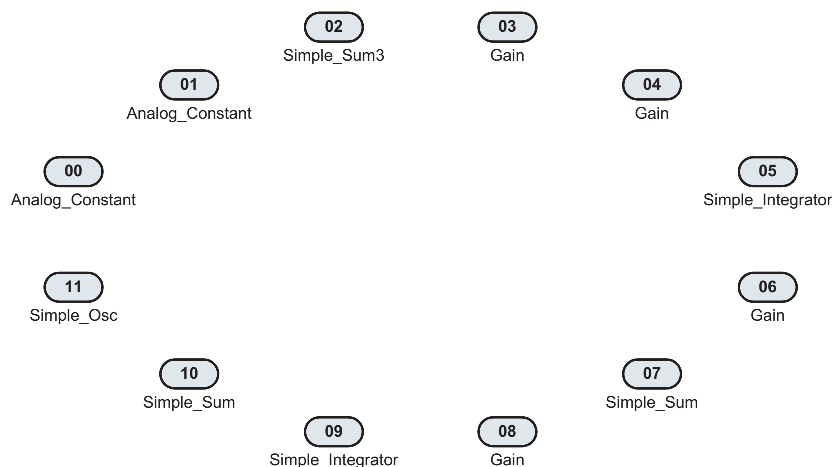

Figura 12: Blocos do sistema de teste do Apêndice C, numerados pelo processo de compilação.

Se um determinado bloco existir em sua biblioteca interna, o compilador dispara a respectiva rotina de *parsing*. Essa rotina captura todos os parâmetros ajustados pelo usuário e, em caso de qualquer inconsistência ou erro de sintaxe, retorna uma mensagem de erro. A saída desse processo pode ser vista no Apêndice C, entre as linhas 11 a 23. Quaisquer blocos não encontrados na biblioteca interna resultam em erro no processo de compilação.

Em seguida, é lido o arquivo de restrições de *hardware*. Esse arquivo, instrui o compilador a respeito da quantidade de memória disponível na plataforma alvo para armazenar o código-objeto, além do número de CPU's disponíveis para execução.

O compilador inicia então a montagem da área de memória dos blocos. Cada bloco lido, e seus parâmetros, são concatenados nessa região de memória, que mais tarde formará o código-objeto final. Deve-se ressaltar, que nessa etapa, todos os ponteiros das entradas dos blocos já são direcionados às posições de memória das saídas, segundo a orientação do usuário descrita no código-fonte.

Ao fazer os apontamentos entre entradas e saídas, simultaneamente, é montada uma tabela de relacionamento direto na forma de uma matriz. Essa matriz é uma expressão do grafo de conectividade entre os nós (blocos, no caso), presentes ao longo do fluxo de dados descrito pelo usuário, como o grafo mostrado na Fig.13. Nessa figura, as setas indicam o fluxo de dados de saídas em direção a entradas de blocos. No caso da representação matricial, os valores de cada célula indicam as dependências de entradas com relação às saídas.

Um exemplo dessa matriz é mostrado no Apêndice C, entre as linhas 42 e 54. Nessa matriz, as colunas indicam as saídas de cada bloco, e as linhas, as entradas. Um número

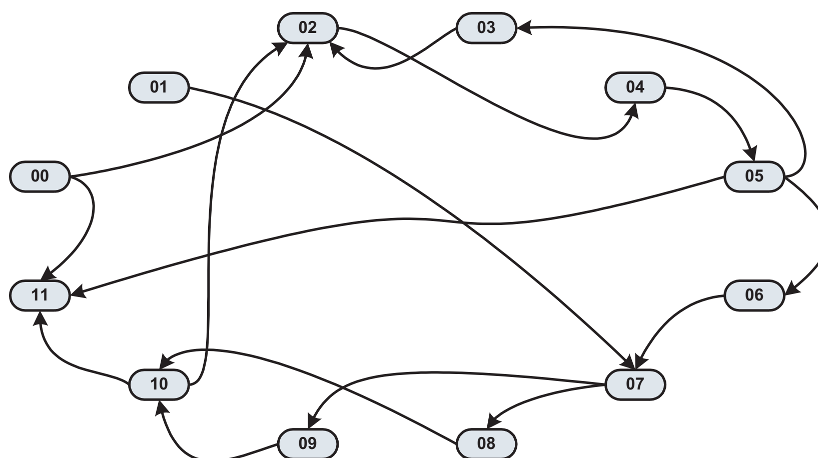

Figura 13: Grafo de conectividade dos blocos do sistema de teste do Apêndice C.

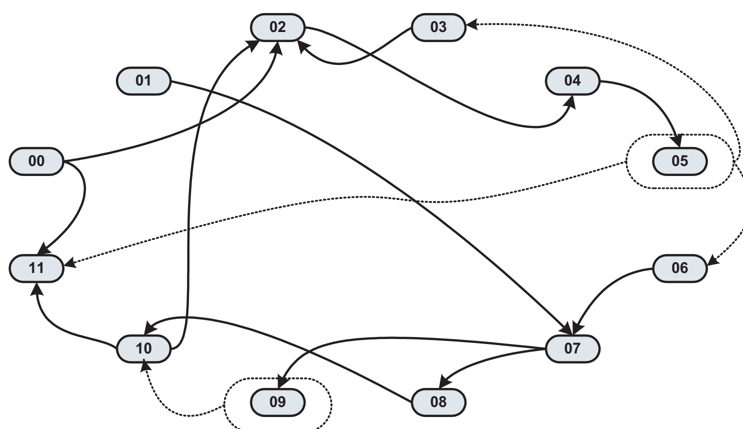

Figura 14: Destaque para os relacionamentos de dependência dos blocos 3, 6, 10 e 11 para com blocos sequenciais 5 e 9, no caso, que representam os integradores do sistema do ApêndiceC.

`01` em uma dada célula indica a ligação de uma entrada com uma saída. Pela exploração da matriz, já é possível determinar se existem duas saídas ligadas a uma única entrada, por exemplo.

Como dito anteriormente, cada bloco do arcabouço possui uma sinalização específica a respeito de seu caráter algébrico ou sequencial. Os blocos sequênciais, ou seja, aqueles que possuem memórias internas, quebram a relação algébrica direta de sua própria saída em função de sua entrada. Nesse caso, sua coluna na matriz de relacionamento deve ser preenchida por zeros, ou seja, ninguém depende da atualização do valor de sua saída. Isso pode ser visto na listagem do Apêndice C, entre as linhas 86 e 98, em uma nova matriz de relacionamento onde as colunas dos blocos 5 e 9 foram zeradas. Visualmente o processo pode ser observado na Fig.14, onde os relacionamentos em questão são destacados, e na Fig.15, onde as dependências para com os blocos 5 e 9 são removidas.

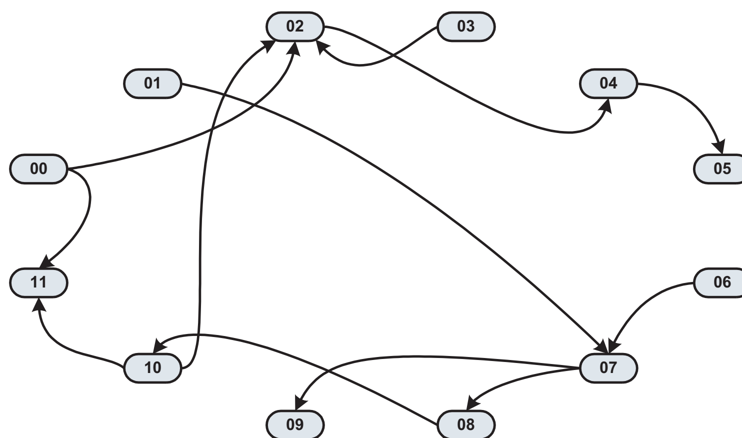

Figura 15: Grafo refinado do sistema de teste do Apêndice C após quebra de relacionamento dos blocos sequenciais, 5 e 9.

Logicamente, a saída desses blocos sequenciais será calculada, mas ela somente será atualizada no ciclo de execução das rotinas de *Commit* comentadas anteriormente. Esse tipo de abordagem foi inspirado nas sínteses de *hardware* comportamental (GAJSKI et al., 1992), onde existem lógicas de transferência de dados entre registradores (*Register Transfer Logic* (RTL)) e estruturas de *pipeline* em um fluxo de dados, que apresentam um paradigma semelhante.

Por meio das informações da matriz de relacionamento e de algoritmos de varredura, como os mostrados em (GAJSKI et al., 1992), é possível identificar a presença de *loops* algébricos dentro da estrutura de blocos. Nesse caso, uma informação de erro é mostrada ao usuário para que se faça uma correção na topologia de blocos ou um re-modelamento no fluxo de dados.

Outra função da matriz é determinar a ordem de execução dos blocos (GAJSKI et al., 1992). Para que a álgebra representada apresente resultados consistentes, conforme um sistema de controle, durante sua execução computacional existe uma ordem de processamento das entradas para as saídas, obedecendo às dependências entre os blocos, de forma que não sejam violadas as equações algébricas e diferenciais do sistema (OGATA, 1998). Essa ordem não tem uma solução única e depende de uma heurística para ser resolvida. Nesse trabalho, a primeira sequência "factível" faz a heurística parar seu processamento, independentemente se a solução é a melhor ou a pior, ou a mais otimizada para processamento paralelo, conforme o caso.

No método implementado, a princípio, o compilador cria uma lista, e estabelece uma ordem arbitrária, com base na sequência original dos blocos, como mostrado no Apêndice C, nas linhas 56 e 57. Nessas linhas do Apêndice, deve-se atentar que os blocos com números mais a esquerda (início da lista) serão executados antes dos demais.

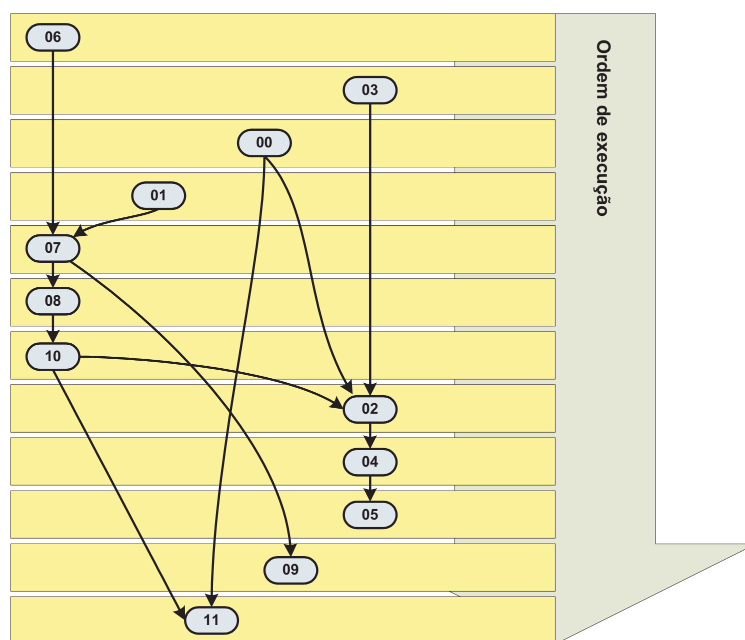

Figura 16: Ordem geral de execução dos blocos baseado no grafo de conectividade refinado.

Em seguida, é feita uma ordenação, do final da lista ao início, com base nos dados da matriz de relacionamento. A ideia é buscar os extremos do grafo de conectividade, ou seja, os elementos que não dependem dos dados de outros são colocados no início da fila de processamento. Em seguida são colocados os demais dependentes, mantendo a precedência da matriz. O resultado é um grafo ordenado, como o mostrado na Fig. 16.

Uma vez que nem todos os blocos possuem rotinas de execução, pos-execução e finalização, o compilador faz um refinamento da lista geral de execução e cria quatro novas listas, cada uma dedicada a um tipo de rotina dos blocos, as listas de: Inicialização (*Init*), Execução (*Run*), Pós-Execução (*Commit*) e Finalização (*Shutdown*). Essas listas podem ser vistas no Apêndice C, nas linhas 103, 104, 105 e 106, respectivamente.

O processo de compilação é, então, finalizado, com a escrita dessas filas de execução dentro da região de memória do código-objeto. Essa região de memória é escrita em disco, e pode ser então transferida para um dispositivo embarcado, ou qualquer outra plataforma onde é executado um interpretador. Deve-se ressaltar que todas as informações necessárias para o funcionamento do *framework* estão contidas nessa área de dados.

4.4.6 Interpretador

O interpretador é o elemento mais simples do arcabouço, justamente para alcançar o melhor desempenho possível durante a execução do fluxo de dados.

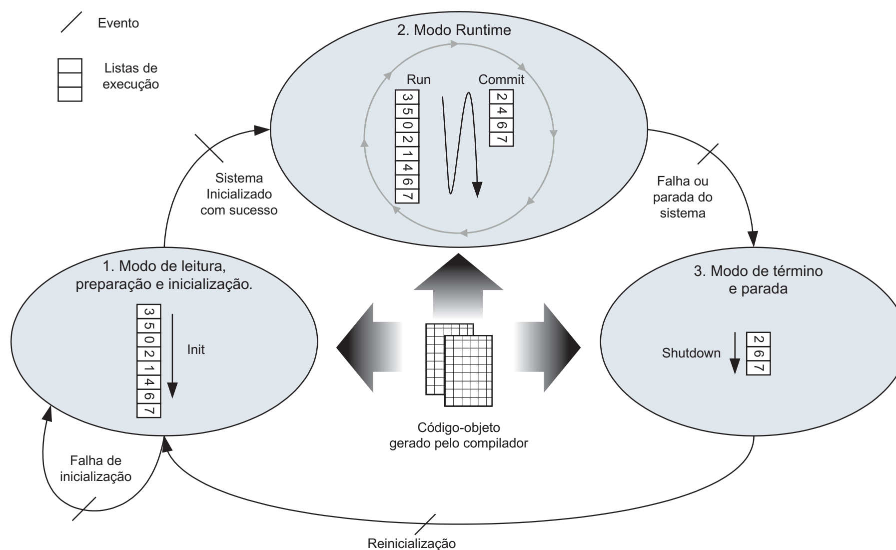

Figura 17: Exemplo de máquina de estados para o funcionamento do interpretador "TB".

Sua estrutura de *software* agrega as rotinas de inicialização, execução, pós-execução e finalização, de cada um dos blocos *template* da biblioteca. Por essa razão, dependendo da quantidade de blocos *template* presentes no arcabouço, a ocupação de memória de programa do Interpretador, na plataforma de execução, pode ser maior ou menor. Em pequenos sistemas microcontrolados, com pouca memória de programa, talvez seja interessante implantar o arcabouço com um quantidade mais enxuta de blocos elementares.

Um exemplo de implementação do interpretador em uma dada plataforma é mostrada na Fig.17. Nessa, foi feita uma máquina de estados com três estados principais, a saber:

1. Modo de leitura, preparação e inicialização;

2. Modo *Runtime*, e;

3. Modo de término e parada;

No modo de leitura, preparação e inicialização, o interpretador permanece aguardando a chegada do código-objeto, por exemplo, através de um canal de comunicação. Uma vez que esse código está disponível, o interpretador faz sua leitura e verifica se o mesmo é compatível com seu conjunto de blocos *template*. Caso seja, ele posiciona uma conjunto de ponteiros sobre o código fonte, e inicia a execução das listas de processamento. A primeira lista executada é a de inicialização dos blocos da aplicação (*Init*).

Em seguida, o sistema evolui para o modo de operação contínua (*runtime*). Nesse modo, o interpretador permanece aguardando algum sinal de sincronismo externo, que

sinalize seu momento de execução. Tipicamente esse sinal é uma interrupção de *hardware* de um conversor analógico digital, uma interrupção de um *timer* interno configurável, ou, por exemplo, um evento de comunicação, como a chegada de um pacote de dados por um determinada interface.

Nesse momento, o interpretador processa as listas de execução (*Run*) e pós-execução (*Commit*), na ordem estipulada pelo compilador. O processo de execução dessas listas precisa ser rápido o suficiente, antes que ocorra outro evento de sincronismo. Esse processo é executado continuamente em um dispositivo HRTCS.

Caso ocorra alguma falha grave de *hardware* ou, no caso de utilização em um computador *desktop*, caso o usuário encerre a execução do interpretador, o sistema irá evoluir para o modo de operação 3, quando será executada a lista (*Shutdown*) que ordena as finalizações dos blocos pertinentes.

4.4.7 Outros recursos e observações

4.4.7.1 Forma de implantação do arcabouço

Para que se inicie com o uso do arcabouço desenvolvido nesse trabalho, basta seguir as seguintes recomendações:

- Escolher uma arquitetura alvo, de preferência compatível com a arquitetura do computador onde será feito o desenvolvimento (vide item a seguir sobre *Endianess*);
- Adaptar os arquivos de compilação e códigos fonte em C do *framework* para incluírem apenas os blocos funcionais desejados para o ambiente de desenvolvimento. Deve-se atentar que o hábito de agregar muitos blocos pode tornar o interpretador difícil de ser portado a outras arquiteturas com maiores restrições de memória;
- Compilar as ferramentas do pré-processador e compilador "TB";
- Adaptar e compilar a ferramenta do interpretador para as plataformas alvo desejadas. No caso de uso direto no computador pessoal de desenvolvimento, já existe uma versão do interpretador que permite apenas a execução em modo *offline*, como um simulador, sem requisitos de desempenho em tempo real;
- No caso de uso em plataformas embarcadas, além da adaptação do interpretador, deve-se criar ou aproveitar algum mecanismo de comunicação com o *hardware*, para que seja possível transferir para esse, o bloco de memória do código-objeto necessário para seu funcionamento. Esse recurso só é necessário durante o desenvolvimento dos algoritmos. Uma vez que esses estejam prontos, é possível alterar a forma de

carregamento do código-objeto, para que o interpretador já se inicie com o código incorporado.

Deve-se atentar que quaisquer mudanças feitas no conjunto de blocos suportado pelo compilador, deve ser feita também nas demais ferramentas (interpretador - principalmente, e pré-processador).

4.4.7.2 Alinhamento de memória e orientação

Para que os códigos-objetos do compilador possam ser lidos e interpretados em outras plataformas sem problemas, deve-se checar as características de alinhamento e orientação (*Endianess*) das palavras binárias na memória da arquitetura. Por exemplo, sabe-se que a arquitetura Intel "x86" (32 bits) possui alinhamento de memória por palavra de 32 bits, com orientação tipo *little-endian*. Qualquer outra arquitetura, que apresente as mesmas características, pode ler e interpretar o mesmo código-objeto formado pelo compilador criado para o "x86". A arquitetura ARM é uma dessas arquiteturas compatíveis.

Para que o ambiente proposto do *framework* seja compatível com o maior número possível de arquiteturas, recomenda-se a utilização de dispositivos microprocessados que apresentem as mesmas característica de largura de dados na arquitetura de suas CPU's, por exemplo, todas as arquiteturas com 32 bits, ou 16 bits, etc. Dessa forma, evita-se uma enorme confusão com *frameworks* compilados para inúmeras plataformas diferentes.

4.4.7.3 Múltiplos fluxos de dados assíncronos

O arcabouço desenvolvido dá suporte para a criação de um único fluxo de dados, executado segundo algum mecanismo de sincronização da plataforma. Entretanto, em muitos sistemas, é interessante existir mais de um fluxo de dados, rodando em taxas de varredura completamente diferentes (assíncronas). Por exemplo: um relé de proteção com recursos de medição, poderia ter dois fluxos de dados, um executado com taxas maiores, para aquisição e processamento digital de sinais, e outro executado com periodicidade menor, para execução de filtragens de média, análise de conteúdo harmônico, etc.

Nesses casos, sugere-se construir não uma aplicação com os dois fluxos de dados internos, mas sim dois fluxos de dados independentes. Na plataforma final, deve-se então implementar dois interpretadores e escaloná-los com as periodicidades desejadas. Isso é particularmente mais simples, em plataformas que possuem um sistema operacional, com recursos de tempo real, capazes de gerenciar múltiplas tarefas de alta prioridade.

Esse método de execução de múltiplos interpretadores é uma forma de se realizar processamento paralelo. Entretanto, caso seja necessária a troca de informações entre os

processos (fluxos de dados), devem ser feitos blocos de comunicação especializados para essa tarefa, gerenciando, inclusive o acesso por meio de semáforos, por exemplo.

4.4.7.4 Desempenho teórico

Uma das maiores discussões a respeito de desempenho, quando se trata de sistemas computacionais (independentemente de ser ou não em tempo real) é o confronto "*Clock* versus Aritmética versus *Worst Case Execution Time* (WCET)" (MACNEIL, 2004).

Dado um algoritmo com um determinado caminho crítico, dada uma aritmética com uma certa precisão, e dada uma determinada velocidade de *clock* da CPU, porque que os números de desempenho reais, em termos de operações por unidade de tempo, nunca não fazem sentido ? A realidade diz que o resultado nunca é preciso, variando de duas a dez vezes o esperado. Isso é devido a uma série de fatores, dentre eles: a qualidade das ferramentas de compilação para código de baixo nível, e as características da arquitetura.

O interpretador do *framework* é a parte mais relevante de todo o sistema, pois é dele que depende o desempenho quando o sistema está em execução. As decisões por utilizar um bloco compacto e monolítico de memória para armazenar o código-objeto, de usar listas de execução para ordenar as chamadas às rotinas dos blocos, de utilizar ponteiros pré-ajustados, de racionalizar os escopos de dados, de evitar movimentações de memória, etc. foram todas tomadas com o objetivo de favorecer, única e exclusivamente, o desempenho do interpretador.

Essas medidas não foram pragmáticas, mas sim fundamentadas em inúmeras experiências e relatos de programadores e engenheiros de sistemas embarcados por toda a Internet, tais como as encontradas no grupo de discussão "comp.arch.embedded" (USENET, 2010) e nas listas de discussão presentes em (EMBEDDEDRELATED, 2010). Conforme mostrado adiante, será possível notar que o resultado final é melhor que várias outras linguagens interpretadas de mercado, como o Java e o Python, mas que, de qualquer forma, os números de desempenho nunca se aproximam do desempenho teórico esperado.

4.4.7.5 Granularidade dos blocos

Como será visto na seção de resultados adiante, o desempenho do sistema fica limitado pela granularidade utilizada para descrever o fluxo de dados do algoritmo.

A chamada "granularidade fina" de código consiste na utilização de milhares de blocos *template*, bastante primitivos, como por exemplo, simples operações de adição, multiplicação, lógica booleana, etc. Como explicado, os conjuntos de algoritmos desses blocos serão ordenados em longas filas de execução e enviados ao interpretador. O papel do

interpretador é processar essas filas ao longo do tempo, isso significa fazer uma série de chamadas às sub-rotinas dos blocos. Essas chamadas a funções consomem um certo tempo computacional, mas, em geral, são relativamente rápidas. Entretanto, como as próprias funções são elementares, pequenas em termos de algoritmo, pode se perder muito mais tempo nas suas chamadas, do que no processamento efetivo dos dados.

Por essa razão, nesse *framework* é sempre importante estudar a granularidade dos processos montados. Se o fluxo de dados demanda um "exército" de pequenos blocos, certamente será melhor construir pessoalmente um novo bloco, dedicado a toda essa funcionalidade, em linguagem compilada em C, por exemplo, do que perder o precioso tempo computacional com chamadas para execução de pequenos processamentos. Desenhar o fluxo de dados com blocos maiores, de maior valor agregado em termos de algoritmo, permite aumentar a granularidade do processo e aproveitar melhor a capacidade computacional da plataforma.

5 Resultados e discussão

O principal objetivo desse trabalho é diminuir os problemas de desenvolvimento de *software* na área de sistemas embarcados e aplicações com estritos requisitos de desempenho, principalmente na área de Sistemas de Potência. Entretanto, "quantificar" a melhoria que esse *framework* traz a um determinado projeto ou aplicação é uma tarefa muito subjetiva a curto prazo.

Como mostrado nos trabalhos de (SIFAKIS, 2005) e (RIEHLE, 2000), somente o uso continuado e os resultados a longo prazo podem realmente atestar o mérito da metodologia, ao comparar as diferenças de tempo e de custo do desenvolvimento antes e depois de sua implantação.

De qualquer forma, uma validação pode ser feita através de vários testes comparativos entre a metodologia proposta e as tecnologias convencionais de produção de *software*, além da aplicação em um problema real.

Nesse sentido foram realizadas duas configurações do arcabouço "TB": uma configuração de blocos elementares para simulação de sistemas dinâmicos, e uma configuração de blocos mais específicos para um cenário real de aplicação em um relé de sobrecorrente.

Com tais configurações, foram realizados os seguintes testes e implementações, descritos adiante com mais detalhes:

- Testes de validação:
  - Testes de consistência numérica;
  - Testes de desempenho computacional;
  - Testes de ocupação de memória;
  - Testes de portabilidade;
  - Testes de granularidade;
- Teste em aplicação real:
  - Implementação, e testes de desempenho de um relé digital de sobrecorrente, com aquisição de dados via barramento de processo da norma IEC 61850.

5.1 Testes de validação

Nessa etapa, foram utilizadas algumas plataformas computacionais para execução em tempo real e execução *offline*, utilizando *hardware*, sistemas operacionais e métodos diferentes para resolução de um mesmo sistema dinâmico, apresentado no Apêndice B. O sistema simples descrito nesse Apêndice já foi executado no passado como parte do trabalho de (PELLINI, 2005), mas envolvendo DPS's no seu processamento em tempo real e programação em Assembly.

Nos testes desse trabalho foram utilizadas duas arquiteturas de *hardware*: computadores IBM PC tradicionais de arquitetura Intel "x86" e microcontroladores de arquitetura ARM9 ARM966E-S. Ambas são plataformas de 32 bits, com *endianess*[1] do tipo *little-endian*, de arquitetura *Harward* modificada. Essas características comuns permitem o emprego do arquivo de código-objeto gerado pelo *framework* TB, e de seu interpretador, em ambas as plataformas, com praticamente nenhuma adaptação.

Em termos de *software* básico, nos PC's utilizou-se um sistema operacional convencional (Microsoft Windows XP) e um sistema operacional com requisitos de tempo real (Linux Ubuntu + RTAI). No microcontrolador foi elaborado um sistema executivo dedicado, com requisitos de processamento em tempo real. Essa escolha heterogênea de sistemas operacionais/executivos tem como objetivo avaliar as diferenças de resultados entre o ambiente de desenvolvimento do engenheiro (computadores comuns com sistemas operacionais comuns) e os ambientes de aplicação de um HRTCS.

Para simulação do caso teste, foi montada uma configuração de blocos no *framework* "TB", conforme mostrado no Apêndice B. Para fins de comparação, o mesmo caso teste foi elaborado no *software* de resolução de sistemas dinâmicos em diagramas de bloco Vissim, versão 3.0, da (VISSIM, 2010), bem como em uma versão de *solver* codificada diretamente em C ANSI e compilada. Esses dois outros simuladores foram criados para execução no PC, apenas.

A preferência pelo uso do *software* Vissim foi devida a problemas e dificuldades encontradas na utilização do MatLab/Simulink, principalmente nos quesitos de precisão numérica, de configuração do método e passo de integração e do seu menor desempenho. A respeito da precisão, notou-se diferenças no resultado de um mesmo caso de simulação, entre versões diferentes do MatLab/Simulink. Mais tarde, foi descoberto que o problema de precisão (abaixo da 7ª casa decimal) era devido ao fato de que algumas versões do Simulink utilizam, em seus blocos, "precisão arbitrária" ao invés de precisão dupla em

[1] *Endianess* é a forma de orientação de palavras na memória do computador. As ferramentas do *framework* desenvolvido dependem dessa característica para a correta geração do código-objeto que será interpretado. No caso, foram buscadas arquiteturas com o mesmo *endianess* para que não fosse necessário recodificar ou alterar tais ferramentas.

ponto flutuante. Um problema semelhante também foi detectado no *software* Vissim, e será comentado adiante.

Foram feitas várias combinações de *software*, *hardware* e sistemas operacionais, chegando a cinco configurações, das quais três utilizam o *framework* desse trabalho, com o mesmo pre-processador e compilador, mas com interpretadores pouco adaptados para cada arquitetura. As configurações principais utilizadas foram:

(a) **PCXP_TB:** Interpretador TB, executado em um computador *laptop*, com processador AMD Turion 64 X2, 1.6 [GHz], 2,0 [GB] de memória RAM, executando o sistema operacional Microsoft Windows XP SP3. O interpretador roda como uma aplicação de usuário, completamente *offline*;

(b) **PCXP_VS:** Algoritmo de simulação do sistema dinâmico programado no *software* Vissim, executado em um computador *laptop*, com processador AMD Turion 64 X2, 1.6 [GHz], 2,0 [GB] de memória RAM, executando o sistema operacional Microsoft Windows XP SP3. O algoritmo roda como uma aplicação de usuário, completamente *offline*;

(c) **PCXP_C:** Algoritmo de simulação do sistema dinâmico programado diretamente em C, executado em um computador *laptop*, com processador AMD Turion 64 X2, 1.6 [GHz], 2,0 [GB] de memória RAM, executando o sistema operacional Microsoft Windows XP SP3. O algoritmo roda como uma aplicação de usuário, completamente *offline*;

(d) **PCRT_TB:** Interpretador TB, executado em tempo real, com um computador *desktop* com processador AMD Athlon XP, 1.6GHz, 512 MB de memória RAM, executando o sistema operacional Linux Ubuntu 8.04, com kernel Linux 2.6.23, customizado com o *patch* para tempo real RTAI 3.6. O interpretador "TB" funciona em tempo real, em um *loop* temporizado com período de 100,0 [μs];

(e) **MURT_TB:** Interpretador "TB" rodando em um kit de desenvolvimento da Raisonance / ST Microelectronics para o microcontrolador STR912, de arquitetura ARM9, *clock* de 96 MHz, 90 [kB] de memória RAM e 512 [kB] de memória FLASH, com sistema executivo especialmente criado para a plataforma. O interpretador "TB" é executado em tempo real dentro de uma interrupção de *timer*, com período de 100,0 [μs].

Em alguns testes, outros arranjos foram realizados para permitir análises mais detalhadas.

O conjunto de blocos *template* utilizados no *framework* para execução do sistema teste descrito no Apêndice B consiste em: bloco de constante analógica, bloco de constante digital, bloco de somatória para duas entradas, bloco de somatória para três entradas com ganhos ajustáveis por entrada, integrador numérico com algoritmo de integração trapezoidal, bloco de multiplicação de sinal por constante e bloco de registro de oscilografia. Todos os blocos foram constituídos para utilizarem aritmética de ponto flutuante de dupla precisão (tipo *double* de variável na linguagem ANSI C).

Foi utilizado um passo fixo de integração de 100,0 [μs], com método de integração trapezoidal, durante um intervalo total de simulação de 200,0 [s]. Os dados foram coletados a cada 100 passos de integração e os testes foram avaliados com precisão até a décima quinta casa decimal.

5.1.1 Consistência numérica

O objetivo desse teste é verificar se os algoritmos modelados através do arcabouço realmente apresentam os resultados esperados, ou seja, se a ordenação da execução dos blocos e a correta álgebra dos elementos estão coerentes com os resultados obtidos através de outros métodos de simulação.

Partindo de condições iniciais nulas, as cinco configurações foram submetidas a um degrau unitário, em t=0,0 [s], na referência de seu controlador de velocidade (variável “WREF” mostrada na Fig.30 do Apêndice B).

Os resultados do teste para as configurações (a), (d) e (e), que executam o *framework* “TB”, foram idênticos até a 15ª casa decimal. Deve-se ressaltar que o microcontrolador ARM não possui o suporte a processamento aritmético em ponto flutuante por *hardware* como na plataforma PC. Nesse caso, para o microcontrolador, foi utilizada uma biblioteca de emulação *soft-float* do compilador “GNU GCC”, capaz de produzir os mesmos resultados do PC.

Analisando a saída do *framework* “TB” com o resultado das configurações com o Vissim (b) e com o algoritmo em ANSI C (c), chega-se aos resultados mostrados na Fig.18.

Na figura, as três curvas de cada configuração se confundem, mostrando resultados praticamente idênticos. Considerando o resultado do Vissim como referência, pode-se fazer uma análise do erro para os demais simuladores, como mostrado na Fig.19.

Na Fig.19, as duas linhas mostradas estão sobrepostas e são idênticas, ou seja, o resultado simulado do código em C e do *framework* são iguais. Entretanto, nota-se que ambos apresentam um erro absoluto com relação ao Vissim da ordem de quase 10^{-4}. Após uma análise mais aprofundada, notaram-se vários problemas no Vissim, semelhantes

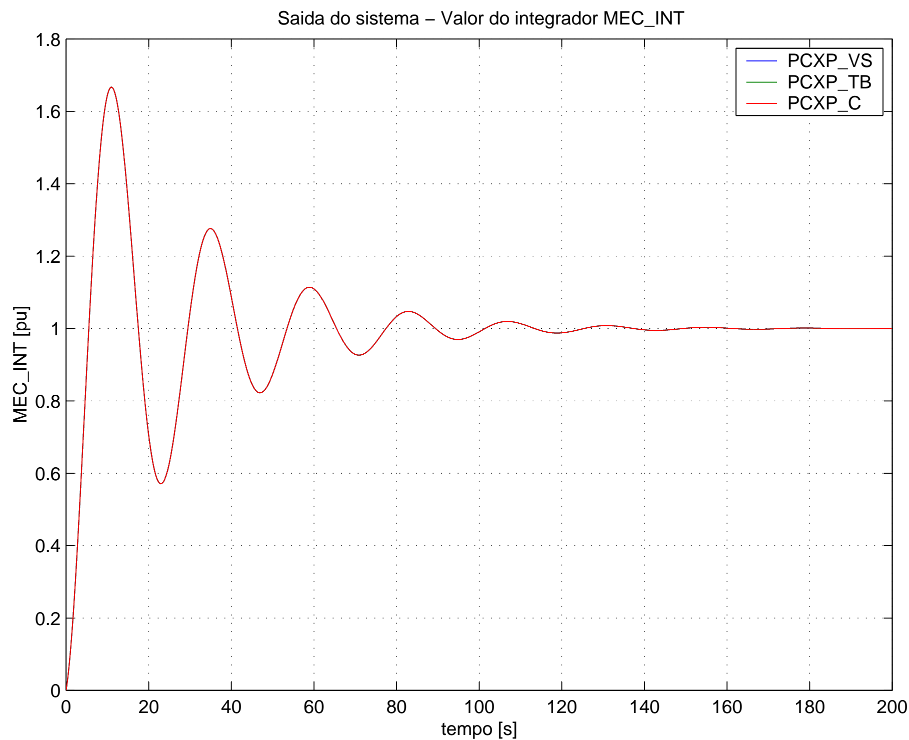

Figura 18: Saída da simulação de um dos *frameworks* "TB", do Vissim e do algoritmo em C puro para o caso teste. As curvas são muito semelhantes e estão sobrepostas.

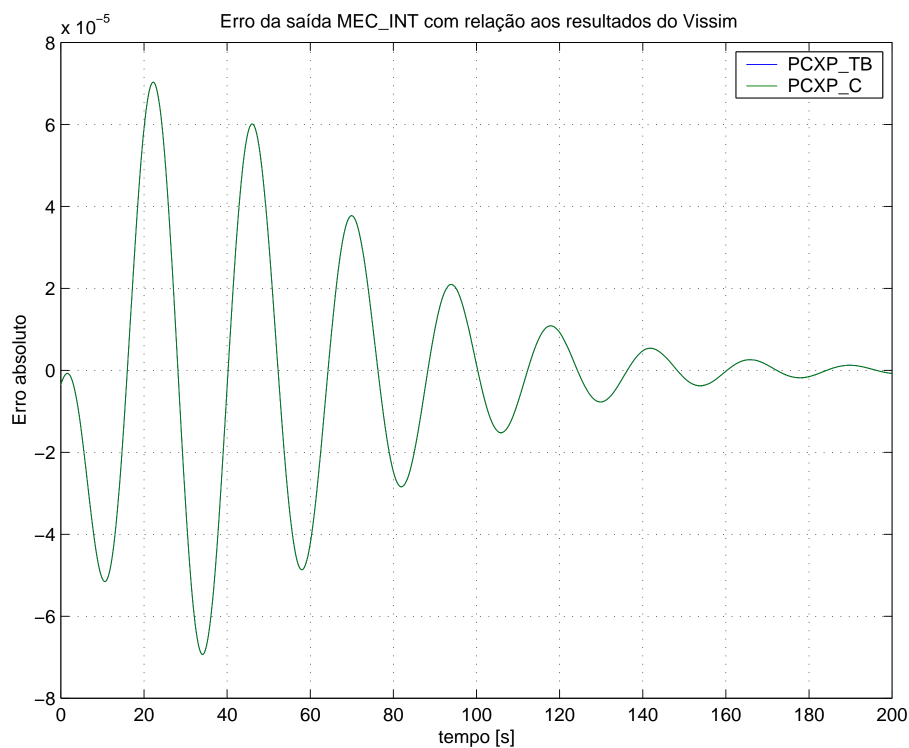

Figura 19: Erro absoluto da saída do *framework* e do algoritmo puro em ANSI C, com relação ao resultado do Vissim, para o mesmo caso teste.

aqueles observados também no MatLab/Simulink.

Notou-se que o Vissim 3.0 apresenta um erro de falta de precisão nos valores dos coeficientes dos blocos de ganho, mostrados na Fig.30 do Apêndice B, pois utiliza precisão simples para representá-los. Foi observado também que o método de coleta dos dados para a elaboração dos gráficos é diferente entre os ambientes de simulação. No Vissim, a coleta dos dados é feita após a integração numérica, enquanto que, no caso do *framework* e do código em C, essa coleta é feita antes da atualização das saídas dos blocos de integração numérica (antes das rotinas de *Commit* dos integradores). Por final, foi verificado que o Vissim, na sua primeira iteração de cálculo, executa uma integração retangular ao invés de uma integração trapezoidal, o que é conceitualmente mais correto, já que não há valores conhecidos das entradas dos integradores antes do início da simulação. No caso do *framework* "TB", a primeira iteração executa uma integração trapezoidal, considerando o valor anterior do integrando igual a zero.

A respeito da validade da álgebra realizada no *framework*, convém dizer que o algoritmo implementado em código ANSI C puro (citado ao final do Apêndice B em B.3) é baseado no método de descrição em espaço de estados mencionado em pelo menos dois livros de referência para a área: (OGATA, 1998) e (WATSON; ARRIGALA, 2003). Segundo esses autores, a descrição desse sistema de teste, no domínio do tempo, resulta no sistema de equações diferenciais ordinárias de primeira ordem, cuja forma matricial é mostrada na Eq. (5.1).

$$\begin{aligned} \begin{bmatrix} \dot{X}_1 \\ \dot{X}_2 \end{bmatrix} &= \begin{bmatrix} Coef_0 & Coef_1 \\ Coef_2 & Coef_3 \end{bmatrix} . \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} + \begin{bmatrix} Coef_4 \\ Coef_5 \end{bmatrix} . \begin{bmatrix} U \end{bmatrix} \\ \begin{bmatrix} Y \end{bmatrix} &= \begin{bmatrix} Coef_6 & Coef_7 \end{bmatrix} . \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} + \begin{bmatrix} Coef_8 \end{bmatrix} . \begin{bmatrix} U \end{bmatrix} \end{aligned} \tag{5.1}$$

Onde, no caso do sistema utilizado, mostrado na Fig. 30 do Apêndice B:

- X_1 e X_2 são os estados do sistema, ou seja, os valores instantâneos dos integradores MEC_INT e W_CONT_ITERM, respectivamente;
- $Coef_0$, $Coef_1$, $Coef_2$ e $Coef_3$ são os coeficientes da matriz de estados e dependem dos ganhos das malhas do sistema, conforme:

$$\begin{aligned} Coef_0 &= K12 \,.\, W_CONT_KP \,.\, WFBK - K12 \,.\, DGAIN \\ Coef_1 &= K12 \\ Coef_2 &= WFBK \\ Coef_3 &= 0 \end{aligned} \tag{5.2}$$

- Da mesma forma, $Coef_4$ e $Coef_5$ são os coeficientes da matriz de entrada, dados respectivamente por:

$$\begin{aligned} Coef_4 &= K12 \ . \ W_CONT_KP \\ Coef_5 &= 1 \end{aligned} \tag{5.3}$$

- U é a variável de entrada livre do sistema, no caso, o valor de referência $WREF$

- Y é a variável de saída do sistema, determinada como uma combinação linear dos estados e da entrada;

- $Coef_6$ e $Coef_7$ são os coeficientes da matriz de saída que, para tornar a saída Y igual ao valor do estado MEC_INT, devem ser dados por:

$$\begin{aligned} Coef_6 &= 1 \\ Coef_7 &= 0 \end{aligned} \tag{5.4}$$

- $Coef_8$ é o coeficiente da matriz de passagem direta (do inglês *feedthrough*) do sistema, no caso, igual a zero.

Esse sistema, simulado em C e no MatLab (sem auxílio do Simulink) como um sistema discreto equivalente, nas condições de precisão, algoritmo de integração (síntese do sistema em tempo discreto através de transformada bilinear, método "Tustin" no MatLab) e passo de tempo já comentados, apresentou resultados numéricos idênticos ao do *framework* para a saída MEC_INT, corroborando a validade *framework* "TB" no esquema de ordenação e execução de seus blocos e em sua álgebra do fluxo de dados.

5.1.2 Desempenho computacional

Às cinco configurações anteriores foi acrescentada mais uma, composta pela seguinte configuração:

(f) PCRT_C: Algoritmo de simulação do sistema dinâmico programado diretamente em C, executado em tempo real, com um computador *desktop* com processador AMD Athlon XP, 1.6GHz, 512 MB de memória RAM, executando o sistema operacional Linux Ubuntu 8.04, com kernel Linux 2.6.23, customizado com o *patch* para tempo real RTAI 3.6. O código é executado em um *loop* temporizado com período de 100,0 $[\mu s]$.

Os seis cenários foram analisados quanto aos seus desempenhos durante a execução da simulação teste. O objetivo principal é verificar se a metodologia adotada de programação

do *framework* é melhor ou pior, em termos de velocidade, do que os demais métodos de programação e implementação. Também será verificada a diferença de desempenho do *framework* quando esse é executado nos computadores IBM PC ou nos microcontroladores ARM.

Deve-se notar que o universo da arquitetura Intel “x86” possui uma grande disponibilidade de recursos de *hardware*, tais como: *clock* elevado, memórias maiores e maior capacidade de processamento e coprocessamento aritmético em ponto-flutuante. Por outro lado, o universo do microcontrolador ARM possui sérias restrições nesses itens, o que deve resultar em alguma diferença de desempenho entre as duas arquiteturas.

Em todas as configurações de teste, mesmo naquelas que apresentam compromissos de execução em tempo real, foi medido o tempo médio de processamento, efetivamente gasto, em cada iteração, na resolução do sistema dinâmico. Foi descontado o tempo ocioso entre as iterações no caso das configurações (d), (e) e (f) que rodam em tempo real.

No caso da execução nos computadores IBM PC, nas configurações (a), (c), (d) e (f), foi possível utilizar um contador interno específico da arquitetura “x86”, denominado “TSC”. Esse registrador possui a resolução temporal de um ciclo de *clock* da CPU e permite medições com grande resolução e confiabilidade, tanto do tempo de cada iteração, como do tempo total de simulação, em termos do número de ciclos de computação da CPU. A conversão do valor lido do registrador “TSC” para as unidades de tempo do Sistema Internacional foi feita com base no valor nominal do relógio desses computadores.

No caso da configuração (b) com o Vissim, o tempo de cada iteração foi estimado com base no tempo total da simulação, medido pelo relógio interno do computador IBM PC. Nesse caso, eventuais derivas e incertezas (entre 20 a 200 [ppm]), devido a qualidade dos osciladores e cristais internos desse relógio, podem ter afetado a precisão dessas medições.

No caso da configuração (e) com o microcontrolador ARM9, o desempenho foi verificado através do monitoramento, com um osciloscópio, de sinais elétricos acionados pelo microcontrolador, durante a execução do código do *framework*. Esse método permite estimar o tempo médio entre o início de uma determinada atividade e seu término. Nesse caso, os cristais e osciladores utilizados como base de tempo para esse equipamento possuem ótima estabilidade e baixa deriva (inferior a 10 [ppm]), permitindo boa precisão.

Os resultados são mostrados na Tabela 1. Na tabela, o desvio padrão das medidas permite estimar o *jitter* na manutenção do desempenho do tempo real.

Tabela 1: Desempenho da simulação teste. Tempos em [μs].

	(a) PCXP_TB	(b) PCXP_VS	(c) PCXP_C	(d) PCRT_TB	(e) MURT_TB	(f) PCRT_C
Passo de tempo de simulação	$100,0$	$100,0$	$100,0$	$100,0$	$100,0$	$100,0$
Tempo médio por iteração	$1,15$	$2,10$ *	$0,36$	$0,46$	$45,3$	$0,09$
Desvio padrão	$0,25$	– *	$0,19$	$0,03$	$0,20$	$0,03$

A obtenção precisa dos parâmetros marcados com "*" na Tabela 1 não foi possível para o Vissim, uma vez que não pode-se ler o registrador de temporização TSC nesse programa. Nesse caso, o tempo médio por iteração foi estimado com base no número total de iterações e o tempo total de simulação. Entretanto, não é possível separar e isolar precisamente esse tempo de processamento de outras tarefas do programa Vissim, tais como o gerenciamento de sua interface gráfica, interação com o usuário, etc.

Os passos de tempo de simulação foram citados na tabela para que se possa estimar o ocupação do processador em cada caso. Ou seja, para se obter essa estimativa, basta efetuar o cálculo mostrado em 5.5.

$$POC = \frac{itertime}{tstep} \;.\; 100,0 \; [\%] \tag{5.5}$$

Onde:

POC é a ocupação percentual ou carregamento do processador;

$itertime$ é o tempo médio por iteração da tabela anterior, e;

$tstep$ é o passo de tempo de simulação.

Apesar de não estarem sendo executados em um sistema operacional em tempo real, analisando-se os resultados de (a) e (c) do tempo médio por iteração e de (d) e (f), pode-se notar a degradação de desempenho mediante o uso da linguagem interpretada desse trabalho *versus* a linguagem C compilada. Essa degradação era esperada, uma vez que o processo de cálculo foi dividido em blocos, chamados individualmente e em sequencia, para executarem o processamento desejado.

Nesse caso, o *overhead*, ou o incremento de carga computacional introduzido pelo uso do interpretador "TB" na plataforma Windows XP foi de cerca de 790 [ηs], e é proporcional à quantidade de blocos utilizada. Como no caso teste existem 11 blocos, essa sobrecarga computacional é equivalente a 72 [$\eta s/bloco$] ou cerca de 116 ciclos de processamento por bloco, considerando-se o *clock* de 1,6 [GHz] do processador utilizado. Na plataforma com sistema operacional em tempo real, esse *overhead* total foi de cerca de 370 [ηs], ou seja, 34 ciclos de processamento por bloco.

Deve-se atentar que esse *overhead* serve apenas como uma ordem de grandeza do custo computacional do *framework*, e é uma estimativa grosseira por bloco, uma vez que o que está sendo analisado é o tempo médio de toda a iteração dividido pelo número total de blocos, e não o tempo especificamente gasto no processamento de cada um.

As diferenças nas medidas de tempo entre as plataformas baseadas no Windows XP (a) e (c) e no SO em tempo real RTAI (d) e (f) são ocasionadas pelas mudanças de contexto, entre as tarefas, feitas pelo escalonador do Windows, que não é adequado para execução em tempo real. Esse escalonador, interrompe o processamento dos programas de teste usados em (a) e (c), durante a execução de uma iteração, conforme as regras de preempção do sistema operacional, reassumindo o processamento mais tarde. Isso ocasiona um aumento significativo do tempo computacional de algumas iterações e impacta negativamente nas medidas de média e desvio padrão. Nos testes efetuados no Windows, algumas iterações tiveram tempo de processamento, com pior caso, de mais de $12,0$ [ms].

Analisando-se os resultados do *framework* executado em tempo real nas configurações (d) e (e) nota-se uma grande diferença de desempenho, com o PC atingindo uma velocidade quase 100 vezes maior que a do microcontrolador. Esse fato é advindo da utilização das bibliotecas de processamento aritmético em ponto flutuante por *software* no microcontrolador ARM e de sua arquitetura interna simplificada, com cache muito inferior ao do PC, menos estágios de pipeline e, sobretudo, *clock* muito inferior. Tomando como base as relações entre os *clocks*, temos:

$$\frac{clock_{PC}}{clock_{ARM}} = \frac{1600.0[MHz]}{96.0[MHz]} = 16,7 \tag{5.6}$$

Além do PC ser 16,7 vezes mais veloz que o microcontrolador, o fato do ARM9 utilizar rotinas aritméticas de ponto flutuante por *software*, resulta em uma penalidade adicional entre 4 a 50 ciclos de processamento por instrução de aritmética em ponto flutuante, dependendo da operação executada. Para obtenção de maior desempenho nesse microcontrolador, as rotinas de cálculo matemático no *framework* (integradores, blocos de ganho, etc.) devem ser recodificados em aritmética inteira ou de ponto fixo, mais apropriadas aos recursos da arquitetura ARM9.

Em linhas gerais, o desempenho do *framework* depende da quantidade de blocos utilizada. Quanto maior esse número, maior será o *overhead* de processamento devido as chamadas às rotinas de execução do blocos e menor será seu desempenho com relação a uma linguagem puramente compilada. Dessa forma, recomenda-se a programação interna dos blocos com algoritmos mais densos e elaborados ao invés de operações pequenas e atômicas, tais como simples adições, multiplicações, etc. Essa "densidade algorítmica" é a granularidade comentada anteriormente. Testes de granularidade são mostrados adiante.

De qualquer forma, para a prototipagem de algoritmos e fluxos de dados, a utilização do *framework*, mesmo com um conjunto de blocos de operações simples, ainda é recomendada. Em detrimento de um maior desempenho, o arcabouço favorece o desenvolvimento sistemático, modular e compartimentado dos algoritmos da aplicação. Conforme esses algoritmos são testados e verificados, todo o conjunto de blocos pode ser recodificado na forma de blocos maiores ou de um bloco único, em linguagem compilada, para obter o desempenho máximo.

No caso de utilização com um bloco único, contendo todo o algoritmo de fluxo de dados, o *framework* aparentemente perde o sentido de sua aplicação. Entretanto, deve-se ressaltar que o uso do arcabouço resolve, além dos problemas operacionais de inicialização e execução, também os problemas de parametrização de um HRTCS, uma vez que ao se transferir o código-objeto para o dispositivo, está se transferindo também toda a configuração interna de seus elementos, variáveis e ajustes de forma automática. Isso poderá ser visto com mais detalhes na descrição do estudo de caso com o relé de proteção adiante.

5.1.3 Ocupação de memória

Um fator relevante para qualquer interpretador, *engine* ou *runtime* de um determinado *framework* ou linguagem de programação, é sua ocupação de memória RAM e ROM, principalmente em sua aplicação em sistemas embarcados microcontrolados.

No caso do *framework* "TB", seu interpretador, para a configuração de teste (e) utilizando o microcontrolador ARM, resultou nos seguintes números mostrados na Tabela 2.

Tabela 2: Ocupação de memória. Valores em [bytes] e [%].

	Tamanho em RAM		**Tamanho em ROM**	
	[bytes]	**[%]**	**[bytes]**	**[%]**
Parte operacional de inicialização	4	$0,7$	1576	$27,2$

... Continua na página seguinte

Tabela 2 – continuação

	Tamanho em RAM		**Tamanho em ROM**	
	[bytes]	**[%]**	**[bytes]**	**[%]**
Parte operacional de execução e pós-execução	4	0,7	968	16,7
Parte operacional de finalização	4	0,7	200	3,5
Biblioteca de blocos	560	97,9	3048	52,6,0
Total	572	100,0	5792	100,0

A ocupação de memória ROM no *framework* desenvolvido é praticamente proporcional ao número de blocos presentes na biblioteca *template*. Por exemplo, no caso de teste, existem sete blocos implementados: integrador simples, constante analógica, constante digital, ganho, somador simples, somador triplo com ganho e oscilógrafo. Desses, todos possuem rotinas de inicialização, ocupando 27,2 [%] de memória ROM. Entretanto, os demais blocos podem possuir ou não as rotinas de execução (*run*), pós-execução *commit* e finalização (*shutdown*), conforme a necessidade.

Quanto à ocupação de memória RAM, nota-se o baixo requisito de memória para a parte operacional do *framework* e um maior consumo na memória designada para os blocos. Essa memória é proporcional à quantidade de blocos instanciados da biblioteca *template*. No caso teste, são utilizados dois blocos integradores, dois somadores simples, um somador triplo com ganho e quatro blocos de ganho. Assim, por exemplo, a memória RAM necessária para a utilização de um integrador é contabilizada duas vezes, e quatro vezes para o caso dos blocos de ganho.

Como já comentado, deve-se considerar que o *framework* utilizado no microcontrolador ARM fez uso de aritmética de ponto flutuante de dupla precisão por *software*. Isso ocasiona pelo menos o dobro de ocupação de memória RAM e ROM no dispositivo, dependendo da quantidade de operações de ponto flutuante presentes nos algoritmos.

Números mais detalhados e realistas de ocupação de memória por blocos são fornecidos adiante nesse trabalho, na descrição do caso de desenvolvimento do relé de proteção com recursos da norma IEC 61850 de valores amostrados (SV).

5.1.4 Portabilidade a outras plataformas

As ferramentas de pré-processamento e compilação do *framework* “TB”, utilizado no ambiente de testes, foram criadas para a plataforma IBM PC, para execução sob o sistema operacional Windows. Os interpretadores, por outro lado, foram feitos em várias versões, adequadas ao *hardware* e ao sistema operacional/executivo que seria utilizado para execução (PC com Windows (a), PC com RTAI (d) e ARM com executivo (e)).

No caso do compilador, como esse é responsável pela geração do código-objeto final, existe uma forte dependência dessa ferramenta com os detalhes intrínsecos da arquitetura onde o código-objeto será executado. As principais dependências com a arquitetura de execução são:

- Tipos usados na representação dos dados;
- *Endianess* e alinhamento da estrutura de memória, e;
- Recursos exclusivos do *hardware*.

No caso teste, o *framework* utiliza variáveis analógicas representadas em ponto flutuante de precisão dupla (tipo *double*, de 64 bits, da linguagem ANSI C), e variáveis inteiras e booleanas representadas como inteiros de precisão dupla (tipo *long*, de 32 bits, da linguagem ANSI C). O arcabouço foi criado para dispositivos com organização de memória do tipo *little-endian* e alinhamento de endereços em 32 bits, que é o padrão adotado nas arquiteturas “x86” e ARM9. Os blocos desenvolvidos também não utilizam nenhum detalhe específico do *hardware* dessas arquiteturas. Nesse caso, o código-objeto gerado pelo compilador é totalmente compatível tanto com os interpretadores para a plataforma PC como os para a plataforma ARM.

Entretanto, caso seja necessário executar o arcabouço em outras arquiteturas, que apresentem alguma diferença nos itens citados anteriormente, será necessário também adaptar o compilador. Esse seria o caso, por exemplo, do emprego do arcabouço em um DSP, como o Analog Devices SHARC 21XXX, que utiliza ponto flutuante em *hardware* com precisão simples ao invés de dupla, orientação de memória tipo *big-endian*, alinhamento de endereços em 16 bits, e que possui recursos internos de duplo núcleo de processamento. Outro exemplo seria a arquitetura PowerPC, que é muito semelhante à Intel “x86”, exceto por sua orientação de memória ser do tipo *big-endian*.

Nos casos de teste citados, só houve a necessidade de se portar e adaptar os interpretadores para cada arquitetura de execução. A máquina de estados descrita no capítulo anterior (para gerenciamento das filas e rotinas de inicialização, execução, pós-execução e finalização) foi alocada em uma interrupção de *hardware* ou em uma tarefa periódica,

conforme a disponibilidade da plataforma. Entretanto, as rotinas internas dos blocos e suas estruturas de dados permaneceram inalteradas. As configurações e dados dos blocos, suas associações, etc. são armazenadas em um mesmo local, monolítico, favorecendo a obtenção dos mesmos resultados, independentemente do dispositivo empregado para sua execução.

Isso é bem diferente do método tradicional de implantação de *software* nesses dispositivos, onde um especialista é requerido para se ajustar as estruturas de dados entre plataformas, para se reparametrizar os algoritmos conforme a necessidade, até mesmo para se interpretar os resultados obtidos entre dispositivos diferentes. Todas as intervenções são feitas diretamente no código-fonte do dispositivo, permitindo o "surgimento" de novos *bugs*, requerendo novas compilações, alterações em documentação e principalmente, gastando mais tempo de desenvolvimento.

5.1.5 Aumento da granularidade dos blocos

O caso utilizado para teste foi elaborado com blocos pequenos e elementares de forma proposital, para se evidenciar que um processo de granularidade muito fina (com muitas operações atômicas e singulares) usando o *framework* terá um resultado muito ruim em termos de desempenho, comparado a um código compilado convencional, devido ao *overhead* das chamadas às rotinas internas dos blocos. Como citado anteriormente, devem-se buscar blocos com conteúdo interno mais denso, com maior número de operações possível.

O caso teste foi recodificado, de forma a englobar os blocos 7, 8, 9 e 10, mostrados na Fig.30 do Apêndice B, em um único bloco. Estes blocos, juntos, desempenham o papel de um controlador proporcional-integral (PI) simplificado, que será criado como um bloco elementar da biblioteca *template* e instanciado para um novo teste de desempenho.

Esse controlador não é um bloco de controlador PI completo, já que esse não possui rotinas especiais de saturação da variável de comando de saída e controles de *anti-windup* do termo integral. Esse bloco somente agrega as álgebras dos demais blocos substituídos na forma de uma única rotina monolítica.

Esse novo esquema é mostrado na Fig.20.

O novo bloco de "controlador PI" foi testado novamente nas configurações dos *frameworks* em tempo real (d) e (e), para serem comparados contra seus resultados anteriores. Os resultados são mostrados na Tabela 3.

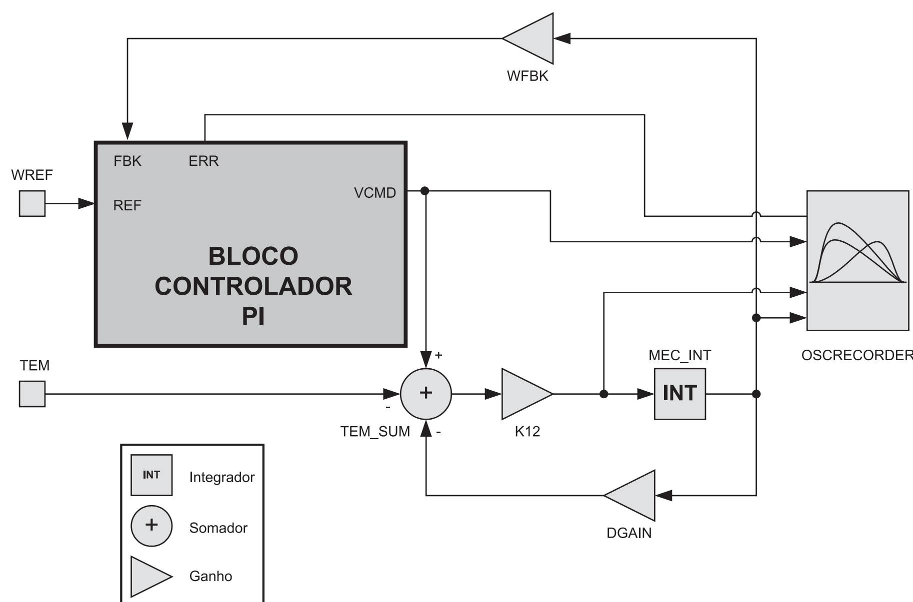

Figura 20: Diagrama de blocos modificado do sistema de teste para análise de granularidade.

Tabela 3: Desempenho da simulação teste com bloco de maior granularidade. Tempos em $[\mu s]$.

	Caso teste original		Caso com controlador PI	
	(d)	**(e)**	**(d)**	**(e)**
	PCRT_TB	**MURT_TB**	**PCRT_TB**	**MURT_TB**
Passo de tempo de simulação	$100,0$	$100,0$	$100,0$	$100,0$
Tempo médio por iteração	$0,46$	$45,3$	$0,37$	$39,1$
Desvio padrão	$0,03$	$0,20$	$0,03$	$0,18$

Os resultados mostram uma melhoria no desempenho global com relação aos resultados anteriores, pela substituição da chamada a quatro blocos individuais pela chamada às rotinas de um único bloco.

A maior dificuldade para o estabelecimento da melhor granularidade de um determinado fluxo de dados é o conhecimento aprofundado dos requisitos operacionais do algoritmos no domínio da aplicação. Isso é, determinar quais as regiões do código que resultam em vantagens em sua modularização e instanciamento, e quais as que devem ser mantidas monolíticas.

5.2 Aplicação em relé digital com IEC 61850

Como ilustração dos méritos e vantagens de utilização do arcabouço desenvolvido nesse trabalho, foi criada uma aplicação em Sistemas de Potência com o desenvolvimento de um relé digital de proteção contra sobrecorrentes, com as funções de:

- Proteção ANSI 51 de sobrecorrente temporizada, monofásica, de tempo definido, e *reset* instantâneo;
- Proteção ANSI 50BF, por falha de disjuntor, e;
- Seletividade lógica.

O dispositivo criado utiliza os paradigmas mais recentes da área de proteção de sistemas elétricos, ao empregar os conceitos da norma IEC 61850, para implementar um esquema de envio e recebimento de mensagens em um barramento de comunicações baseado em rede Ethernet. Mais informações a respeito dessa norma e desse esquema de mensagens são apresentadas adiante.

O dispositivo foi desenvolvido e testado com o arcabouço desse trabalho em uma plataforma IBM PC *desktop*, como a mostrada anteriormente na configuração (a), e depois portado para interpretadores, tanto para a plataforma IBM PC em tempo real da configuração (d), quanto para um microcontrolador da configuração (e).

5.2.1 Detalhes da norma IEC 61850

Como o domínio de aplicação do dispositivo desenvolvido requer a utilização dos recursos da norma IEC 61850 (IEC61850, 2003), será feita uma breve introdução aos conceitos dessa norma.

Em plantas de geração e nas subestações de transmissão e distribuição de energia elétrica é comum a presença de vários elementos sensores, elementos atuadores, reguladores, IED‘s, PLC‘s e unidades de aquisição de dados remotas, conectados entre si, para desempenhar as funções desejadas de automação e proteção do sistema elétrico.

Em uma subestação de distribuição por exemplo, pode-se ter uma grande variedade de equipamentos, tais como:

- Elementos sensores - tais como transformadores de tensão e corrente, transdutores de posição, sinalizadores de estado, etc.;
- Elementos atuadores - tais como contatores, chaves e disjuntores comandados;

- Reguladores - como os reguladores automáticos de tensão presente em transformadores com (*load tap-changer*);
- Dispositivos de proteção - para realizar a proteção diferencial de transformadores, proteção de distância ou diferencial em linhas de transmissão, proteções de sobrecorrente nas saídas dos alimentadores, etc.;
- Controladores Lógico-Programáveis - para desempenhar funções de automatismos em manobras e controle de equipamentos da subestação;
- Unidades de aquisição remota - tais como oscilógrafos e sistemas de aquisição de dados para as finalidades de registro, monitoramento e supervisão de estados e grandezas de interesse da subestação.

Todos esses equipamentos estão normalmente interconectados para que seja possível executar todas as funções e automatismos necessários para a operação da subestação, seja de forma local ou remota. Todas essas funções devem obedecer, necessariamente, a uma série de critérios de intertravamento, sequenciamento e manobra para os equipamentos, com o objetivo de garantir um fornecimento de energia aos consumidores de forma contínua, segura e com qualidade.

Essa interconexão dos equipamentos era feita até então com o uso de fiações dedicadas, ligando entradas e saídas digitais e analógicas e canais de comunicação de dados entre os dispositivos envolvidos. Tais esquemas requerem um enorme quantidade de fios e conexões, que trazem consigo problemas intrínsecos de interfaceamento (inúmeros padrões de tensão, corrente e protocolos de comunicação), projeto, configuração, comissionamento (verificação e validação), manutenção (rastreabilidade de defeitos, substituição de fios e equipamentos), documentação (esquemas elétricos complicados) e confiabilidade.

A norma IEC 61860 é um esforço para se racionalizar tais esquemas de proteção e automação de subestações, através de um conjunto de diretrizes, metodologias e protocolos padronizados. O objetivo principal é simplificar os esquemas de intertravamento e comunicação através de conceitos de sistemas distribuídos e redes de comunicação de dados, além de permitir a aplicação de paradigmas modernos, tais como: uso de modelos de dados com orientação a objetos, visão sistemática da subestação e de seus equipamentos, utilização de dispositivos lógicos, etc.

Nesse contexto, uma das mudanças mais sensíveis da IEC 61850 é a adoção de protocolos de comunicação rápidos, baseados em rede Ethernet, para efetuar a comunicação entre os dispositivos, através de mensagens prioritárias denominadas GOOSE, do inglês *Generic Object Oriented Substation Event*. Tais mensagens contém informações, tais como estados booleanos, valores digitalizados de tensão, corrente, etc., e são transmitidas na

rede de comunicação de forma prioritária, segundo critérios de desempenho definidos na norma, dependendo da área de aplicação do dispositivo. Por exemplo, em subestações de transmissão, os requisitos de desempenho são mais rigorosos do que aqueles designados para subestações de distribuição.

Estruturalmente, tais mensagens GOOSE são pacotes de dados “brutos”, para rede Ethernet, (também chamados de *Raw Ethernet*), que possuem um quadro de dados (*frame*) simplificado, conforme a norma IEEE 802.3. Tal quadro de dados é composto, basicamente, de um endereço designando um destino para a mensagem, de um endereço designando a fonte das informações e de um especificador de mensagem. No caso das mensagens do tipo GOOSE, os endereços de destino são usualmente do tipo *multicast*. Dessa forma, as mensagens que são enviadas pela rede têm como destino todos os demais equipamentos conectado a essa rede. As mensagens podem ter ainda informações específicas em seu quadro de dados para a priorização de seu tratamento dentro da infra-estrutura de rede, para que o trânsito do pacote se dê de forma prioritária e com a menor latência possível.

A norma IEC 61850 prevê dois tipos de mensagens rápidas entre os IED‘s: as mensagens de automação tipo GOOSE normais e as mensagens de valores amostrados tipo GOOSE-SV (“SV” do inglês, *Sampled Values*).

As mensagens GOOSE normais contém uma coleção (*dataset*) de informações, configurável pelos engenheiros projetistas, contendo estados booleanos, valores analógicos e outros tipos de dados, que representam as saídas de um determinado equipamento. Essas informações, no contexto tradicional, seriam enviadas através de pares de fios dedicados provenientes de saídas digitais e analógicas. Por outro lado, no contexto da IEC 61850, os equipamentos publicam seus *datasets*, encapsulados em mensagens GOOSE, por meio de interfaces de rede Ethernet. Devido a natureza *multicast*, toda os dispositivos conectados à uma rede de comunicação recebem as mensagens GOOSE enviadas pelos demais dispositivos. Aqueles que tem interesse em uma determinada informação de uma mensagem, fazem o recebimento e a interpretação do conteúdo dessas mensagens, utilizando os dados em suas lógicas internas.

O protocolo de tráfego de mensagens GOOSE normais não possui mecanismos de garantia de entrega das mensagens aos destinatários na rede de comunicação. Para que haja uma certa confiabilidade nessa entrega, as mensagens são enviadas repetidas vezes pela rede Ethernet, em uma cadência controlada pelo dispositivo remetente dos pacotes. Em situações onde não há mudanças de estado nas variáveis do *dataset*, cada mensagem é enviada em uma frequência mínima, por exemplo, de 1,0 [Hz]. Entretanto, caso haja uma mudança em qualquer estado, uma mensagem é enviada imediatamente, seguida por várias repetições, enviadas a intervalos de tempo crescentes (de forma exponencial

tipicamente) até alcançar a frequencia mínima estipulada, por exemplo, a cada: 4,0 [ms] - 8,0 [ms] - 16,0 [ms] - 32,0 [ms] - ... Dessa forma, tais mensagens GOOSE normais se prestam para realizar a automação, sequenciamento e controle na subestação, de forma consistente e confiável.

Como a norma preconiza a racionalização de fiações na subestação, o passo seguinte foi a determinação de um outro tipo de mensagem GOOSE, com o propósito de carregar informações de valores amostrados de transformadores de tensão, corrente, estados de chaves, etc. - grandezas que antes trafegavam por fiações analógicas exclusivas dentro da subestação. Um IED que antigamente possui transformadores de condicionamento, filtros analógicos e conversores analógico-digitais em sua infra-estrutura de *hardware*, pode substituir todo esse esquema por uma interface de comunicação Ethernet. No campo, junto dos dispositivos de potência, é instalado uma unidade concentradora, denominada pela norma como *Merging-Unit*. Essa unidade é responsável por receber as fiações tradicionais dos sinais analógicos de tensão e corrente, filtrá-los e digitalizá-los de forma sincronizada, constituindo uma coleção de dados (*dataset*) amostrados. Esses dados são enviados como mensagens GOOSE modificadas, denominadas GOOSE-SV (*GOOSE Sampled Values*), a uma taxa configurável de 16, 80, 240 até 255 amostras por ciclo de frequência fundamental. As mensagens tipo GOOSE-SV são enviadas sem confirmação de recebimento e sem as repetições periódicas das mensagens GOOSE tradicionais.

A princípio, cada tipo de mensagem (GOOSE normal e GOOSE-SV) deveria trafegar em redes de comunicação independentes, denominadas: barramento de automação ou estação (do inglês, *station bus*) - no caso das mensagens GOOSE normais; e barramento do processo elétrico (do inglês, *process bus*) - no caso das mensagens de valores amostrados GOOSE-SV. Dessa forma, as mensagens de valores amostrados estariam presentes em uma rede dedicada. Entretanto a própria norma preconiza que, com o uso de *switches* e outros elementos de rede capazes de segregar o tráfego e priorizá-lo corretamente, ambos os barramentos podem ser integrados em uma única infra-estrutura de rede de comunicação Ethernet, onde circulam tanto os pacotes GOOSE-SV e GOOSE normais, como todos os demais protocolos de comunicação tradicionais de redes Ethernet (pacotes TCP/IP, UDP/IP, etc.).

Até 2010, a grande maioria dos fabricantes de IED's de automação e proteção de sistemas elétricos já possui suporte aos preceitos da norma IEC 61850, pelo menos com respeito ao envio e recebimento de mensagens GOOSE normais. Entretanto, existem poucas implementações de *Merging-Units* utilizando os recursos de valores amostrados providos pelas mensagens tipo GOOSE-SV. As soluções existentes ainda são proprietárias ou requerem o uso de barramentos de comunicação dedicados.

Nesse trabalho, será feita uma montagem pioneira com ambos os barramentos (*station*

bus e *process bus*) operando em uma mesma rede de comunicação, em um protótipo de um IED com apenas uma interface Ethernet. No caso, o arcabouço desenvolvido nesse trabalho será responsável por operacionalizar toda a infra-estrutura interna ao IED para o recebimento, tratamento e processamento das mensagens GOOSE e dos algoritmos de proteção envolvidos.

5.2.2 Detalhes do desenvolvimento do IED com o *framework* "TB"

Para o desenvolvimento, inicialmente foi analisado o domínio de aplicação de IED's de proteção em sistemas de potência, conforme mostrado por (SENGER et al., 2008), com o intuito de identificar os blocos funcionais mais comuns, e os requisitos de tipos de dados e capacidade de processamento.

A seguir, os blocos foram implementados no *framework* e testados individualmente quanto ao seu correto funcionamento.

A aplicação final foi então desenhada e testada nos vários interpretadores disponíveis, desde a versão *offline* para uso no computador IBM PC com Windows, até as versões em tempo real com o RTAI e o sistema executivo para a plataforma ARM9.

5.2.2.1 Análise de domínio e definição dos blocos

A partir de alguns cenários de aplicação de IED's de proteção em sistemas de potência, foi determinado um fluxo de dados típico para esse dispositivo. Esse fluxo consiste, basicamente:

I No recebimento de valores amostrados do sinal de corrente alternada do sistema elétrico;

II No processamento digital dessas amostras;

III Na análise desses sinais processados para determinação de grandezas auxiliares de atuação ou restrição;

IV No emprego dessas grandezas para o desempenho das funções de proteção desejadas;

V No intertravamento lógico entre os resultados das funções de proteção, e;

VI Nas sinalizações e comunicações das decisões do processo ao mundo exterior.

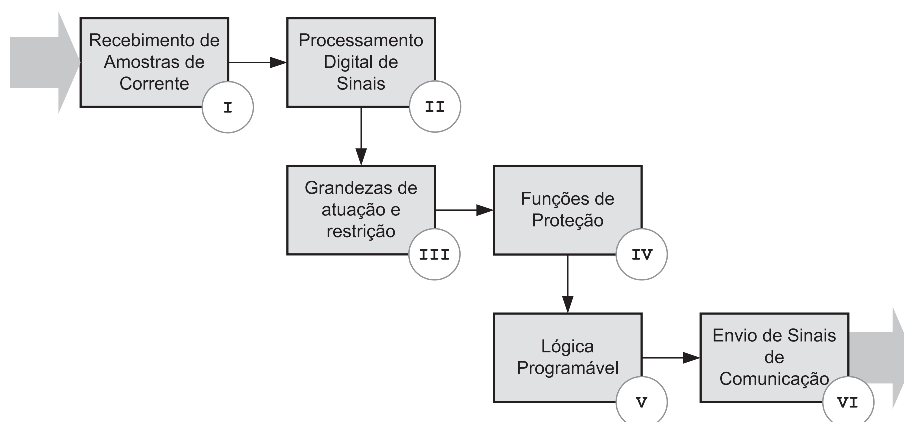

Figura 21: Fluxo de dados simplificado de um IED de proteção.

Esse processo deve ser executado continuamente para cada amostra recebida do sinal de entrada digitalizado. Cada um dos itens enumerados desse processo pode ser visto com mais detalhes na Fig.21.

Nesse contexto, um conjunto de blocos do *framework* foi adaptado para esse cenário típico. Tais blocos desempenham as seguintes funções:

- Filtragem digital, tipo FIR, para obtenção de fasores em coordenadas cartesianas do sinal amostrado da corrente alternada;
- Cálculo de amplitude e fase de um fasor em coordenadas retangulares;
- Condicionamento de grandezas (ajuste de ganho e *offset*);
- Comparação de valores analógicos;
- Lógica booleana com múltiplas entradas;
- *Flip-Flop*;
- Temporizadores e geradores de pulsos;
- Entradas e saídas analógicas e digitais de interface externa;
- Comunicação para entrada de dados via pacotes Ethernet GOOSE-SV de barramento de processo, e;
- Comunicação de entrada e saída via pacotes Ethernet GOOSE para barramentos de automação (*station bus*).

Outros blocos de apoio foram criados para auxílio ao desenvolvimento, tais como: geradores de sinais, registradores de eventos, leitores de dados a partir de arquivos em disco, etc.

Os dados recebidos nesse fluxo são os valores instantâneos da corrente alternada primária do sistema elétrico, condicionadas e digitalizadas por um transformador de instrumentação e um equipamento concentrador de aquisição de dados, também chamado de *Merging Unit*, segundo a (IEC61850, 2003).

Existem várias apresentações para esse sinal digitalizado, com várias características de resolução em número de bits, fatores de conversão da grandeza analógica real para seu valor digitalizado, taxas de aquisição, largura de banda do espectro de frequências, envio dos dados em valores de engenharia em ponto flutuante ou em valores codificados em ponto fixo, etc.

Foi adotado o padrão de apresentação dos dados como o mostrado na norma IEC 61850, parte 9-1. Nessa, os valores das corrente são enviadas em palavras inteiras, de 16 bits, codificadas em valores por unidade [p.u.], segundo uma base pré-definida por um determinado fundo de escala e uma determinada corrente nominal. Mais detalhes a respeito dessa codificação podem ser obtidos na norma (IEC61850, 2003) e também na norma (IEC60044, 2003).

A taxa de envio dos dados pela *Merging Unit* foi adotada com 16 amostras por ciclo, para um ciclo de frequência fundamental entre 45,0 a 75,0 [Hz]. Isso resulta em um período mínimo (t_{min}) e máximo (t_{max}) de aquisição/recepção de dados como mostrado em 5.7.

$$t_{min} = \frac{1,0}{16\ .\ 75,0} \ = \ 833,3[\mu s] \qquad t_{max} = \frac{1,0}{16\ .\ 45,0} \ = \ 1388,9[\mu s] \tag{5.7}$$

O IED desenvolvido deve calcular, para cada amostra recebida por pacotes de rede, a magnitude de uma corrente digitalizada; comparar essa magnitude com valores pré-programados de um limiar de atuação (função ANSI 51); tomar as decisões de *trip* e bloqueio de outros relés de proteção em sua vizinhança; e enviar essas informações através de novos pacotes pela rede Ethernet. Como tudo deve ser feito no mesmo passo de cálculo, a aplicação no *framework* deve ser desenvolvida para o menor período de amostragem possível, ou seja, seu ciclo de execução deve ser inferior ao t_{min} especificado em (5.7), para que não ocorra o *overrun* do tempo de processamento.

Devido à natureza dos dados digitalizados de entrada, o *framework* será desenvolvido com toda sua aritmética em ponto fixo ou aritmética inteira de 32 bits, com saturação dos operandos de entrada e saída dos blocos em até 16 bits, considerando o bit de sinal.

5.2.2.2 Implementação dos blocos

A seguir são descritos alguns detalhes de cada um dos blocos desenvolvidos para o *framework* do IED de proteção.

Filtro digital FIR

O bloco de filtro digital consiste em um *loop* de multiplicações e somas, para constituir um filtro *Finite Impulse Response* (FIR), conforme (OGATA, 1998), de ordem configurável de 1 até 32 *taps*. Os coeficientes do filtro são dimensionados em ponto fixo de 16 bits, codificados como 0,15 bits mais um bit de sinal, podendo alcançar valores individuais entre -1,0 e 0,99997. A aritmética interna do filtro é feita em 32 bits, com saturação. O filtro possui uma entrada inteira de 32 bits, truncada internamente para 16 bits, com sinal, e uma saída inteira, de 32 bits, também truncada internamente para 16 bits com sinal. Internamente, há vetores de dados de 32 bits para armazenamento dos coeficientes e das amostras anteriores da entrada do filtro, utilizadas no processo de convolução. No caso do relé, são usados dois filtros FIR para implementar a transformada discreta de Fourier, responsáveis por calcular a parte real e imaginária do fasor da componente de frequência fundamental do sistema.

Cálculo de módulo e fase

O bloco de cálculo de amplitude e fase para um fasor em coordenadas cartesianas foi criado em aritmética inteira. O bloco possui duas entradas inteiras de 32 bits e duas saídas inteiras de 32 bits, todas com truncamento interno para 16 bits com sinal. Inicialmente foi utilizada uma rotina de cálculo para o módulo e a fase através de polinômios em ponto fixo: um que aproxima a função de raiz quadrada e outro que aproxima a rotina de arco tangente com decodificação de quadrante (*atan2*). Entretanto, durante os testes do bloco, foi notado que a tarefa de elevar as coordenadas cartesianas do fasor ao quadrado, somá-las e submetê-las aos polinômios aproximadores, ocupavam um tempo computacional considerável. Como alternativa, foi feita uma nova implementação de toda a rotina, através de uma função integrada de cálculo de módulo e fase do tipo “vetor CORDIC”, descrita por (ANDRAKA, 1998). A rotina CORDIC apresentou um desempenho cerca de 30,0 [%] melhor que a alternativa original e foi adotada como padrão.

Ganho e condicionamento

O bloco de condicionamento possui uma entrada e uma saída em 32 bits, com truncamento interno para 16 bits com sinal. Esse bloco consiste em um ganho comum, cujo coeficiente multiplicador pode ser dimensionado com até 32 bits, com sinal, com um divisor implícito de 16 bits, ajustável em potências de 2. O ganho criado é equivalente a um número de ponto fixo com 15,16 bits, podendo assumir uma certa gama de valores decimais entre +32767,99998 até -32768,0.

Comparação

O bloco de comparação de valores analógicos possui duas entrada de 32 bits, com truncamento interno para 16 bits, contendo os operadores a serem utilizados na comparação e duas saídas booleanas, complementares, representadas em 32 bits. O bloco pode efetuar as funções de comparação “maior”, “menor”, “maior igual”, “menor igual”, “igual” e “diferente”, conforme a decisão do usuário. Internamente o bloco consiste em uma série de comparações simples do tipo *if...then...else*.

Lógica booleana

O bloco de lógica booleana possui de 1 até 8 entradas booleanas, representadas em 32 bits, e duas saídas booleanas complementares, representadas em 32 bits. O bloco pode efetuar as funções lógicas de AND, OR, NAND, NOR, XOR, XNOR e NOT[2], conforme a decisão do usuário. Internamente o bloco consiste em uma série de *loops*, com as operações lógicas das entradas. Alternativamente, poderia ter sido implementada uma rotina de *look-up table*, que permitisse implementar uma “tabela verdade” de qualquer função combinatória desejada pelo usuário, como feito nas FPGAs modernas.

Flip-Flop

O bloco de *flip-flop* possui cinco entradas e duas saídas booleanas complementares, todas representadas em 32 bits. O bloco implementa um *flip-flop* tipo J-K (mestre-escravo) com entradas J e K sincronizadas a uma entrada de *clock* específica, além de entradas de *set* e *reset* independentes desse *clock*, com prevalência da entrada de *reset*. A

[2]No caso do bloco ajustado para realizar a função NOT, só é processada a primeira entrada configurada no bloco.

menor duração do pulso de *clock* para funcionamento da parte síncrona do *flip-flop* é o período de execução de uma iteração do arcabouço.

Temporizadores e mono-estáveis

O bloco de temporizador possui uma entrada e duas saídas booleanas complementares, todas representadas em 32 bits. O bloco possui dois contadores internos, responsáveis por controlar os sinais de saída do bloco ao longo do tempo, conforme uma borda de subida ou descida ocorre no seu sinal de entrada. Dessa forma, o bloco pode atrasar o tempo de subida e/ou descida do sinal de sua saída, criando pulsos positivos ou negativos, com a largura desejada pelo usuário. As temporizações dos tempos para subir e para descer do sinal de saída são ajustadas em milisegundos pelo usuário, conforme a necessidade. Foi criada também uma variante do bloco temporizador, que se comporta como um gerador mono-estável, ou seja, conforme surge uma mudança de nível baixo para alto em sua entrada, sua saída produz um pulso com largura configurável em sua saída, retornando então ao nível baixo.

Interfaces de entrada e saída de uso geral

Os blocos de interfaces de entrada e saída externas foram criados para o intercâmbio de informações digitais ou analógicas entre o ambiente do *framework* e o restante do ambiente da aplicação. Tais blocos são simples áreas de memória estáticas, onde o sistema operacional ou executivo da plataforma pode ler ou escrever dados. Dessa forma, informações vindas do exterior da máquina de estados do interpretador podem ser obtidas, por exemplo, provenientes de placas de aquisição de dados ou botões da IHM do usuário ou, de forma análoga, informações de saída pode ser escritas em placas de síntese de sinais analógicos e digitais, ou em placas com relés para acionamento, placas com LED's e painéis para sinalização, etc.

Tais blocos possuem até 8 entradas ou 8 saídas inteiras de 32 bits, onde pode-se codificar qualquer informação, desde valores analógicos até booleanos, conforme a necessidade.

Recebimento de dados GOOSE *Sampled Values* (SV)

O bloco de recebimento de comunicação do tipo GOOSE-SV foi criado segundo a especificação da IEC 61850, parte 9-1. Essa norma especifica coleções de dados (*datasets*) enviados através da rede Ethernet por uma *merging unit*, contendo os valores amostrados

de tensões e correntes, valores de entradas digitais, valores de relógios de sincronismo, etc. Ao conteúdo total de informações desse pacote dá-se o nome de GOOSE *Application Protocol Data Unit* (APDU). Esse pode conter uma ou mais dessas coleções de dados, também chamadas de *Application Service Data Unit* (ASDU). Na parte 9-1 da norma (IEC61850, 2003) é descrito um ASDU padrão, denominado de *Universal Dataset*, que contém apenas as informações de valores amostrados de tensão e corrente, codificados de forma binária, com palavras de 16 bits, em orientação *big-endian*, com o conteúdo de dados como o mostrado na Fig.22.

Deve-se atentar que o ASDU mostrado na Fig.22 ainda requer um cabeçalho de alguns *bytes* para constituir o APDU da mensagem GOOSE. O APDU, por sua vez, ainda recebe o cabeçalho Ethernet puro contendo os endereços MAC destino, fonte, tipo de pacote Ethernet (*Ethertype*) e informações de VLAN, caso necessário.

O bloco de recebimento GOOSE-SV implementa toda a lógica de interpretação (*parsing*) do pacote de rede recebido em uma rotina externa ao arcabouço. Essa rotina é responsável por validar, fora do *loop* de execução dos blocos, a origem dos dados do pacote, seu conteúdo e informações principais, tais como: a frequência de amostragem esperada (em 16 amostras por ciclo), a informação de revisão de configuração, o contador sequencial de amostras (para verificação de perda de pacotes durante o recebimento), o identificador do dispositivo lógico que está enviando os dados (LDName), informações de qualidade e validade dos canais amostrados, etc. Caso essas informações estejam corretas, a rotina externa faz a leitura das informações para uma memória interna ao bloco e aciona a máquina de estados de interpretação do *framework* para uma nova iteração. No caso da plataforma "x86" ou ARM, essa rotina externa ainda realiza a inversão dos *bytes* de cada palavra de medida, convertendo-a para a orientação *little-endian*.

Recebimento de dados GOOSE de automação

O bloco de recebimento de comunicação do tipo GOOSE, para barramento de automação (*station bus*), foi criado segundo a especificação da IEC 61850, partes 8-1 e 7-2. Essa norma especifica o conteúdo do pacote GOOSE e seus *datasets* para envio e recebimento, além das máquinas de estados e funções típicas para recebimento de informações e interpretação de seu conteúdo de dados.

Nesse arcabouço foi implementado um bloco de recebimento GOOSE simplificado, capaz de entender apenas *datasets* compostos por tipos booleanos, com até 8 estados. Não é dado suporte a outros tipos de *datatypes* definidos pela norma, tais como: qualidades, estampas de tempo ou valores analógicos.

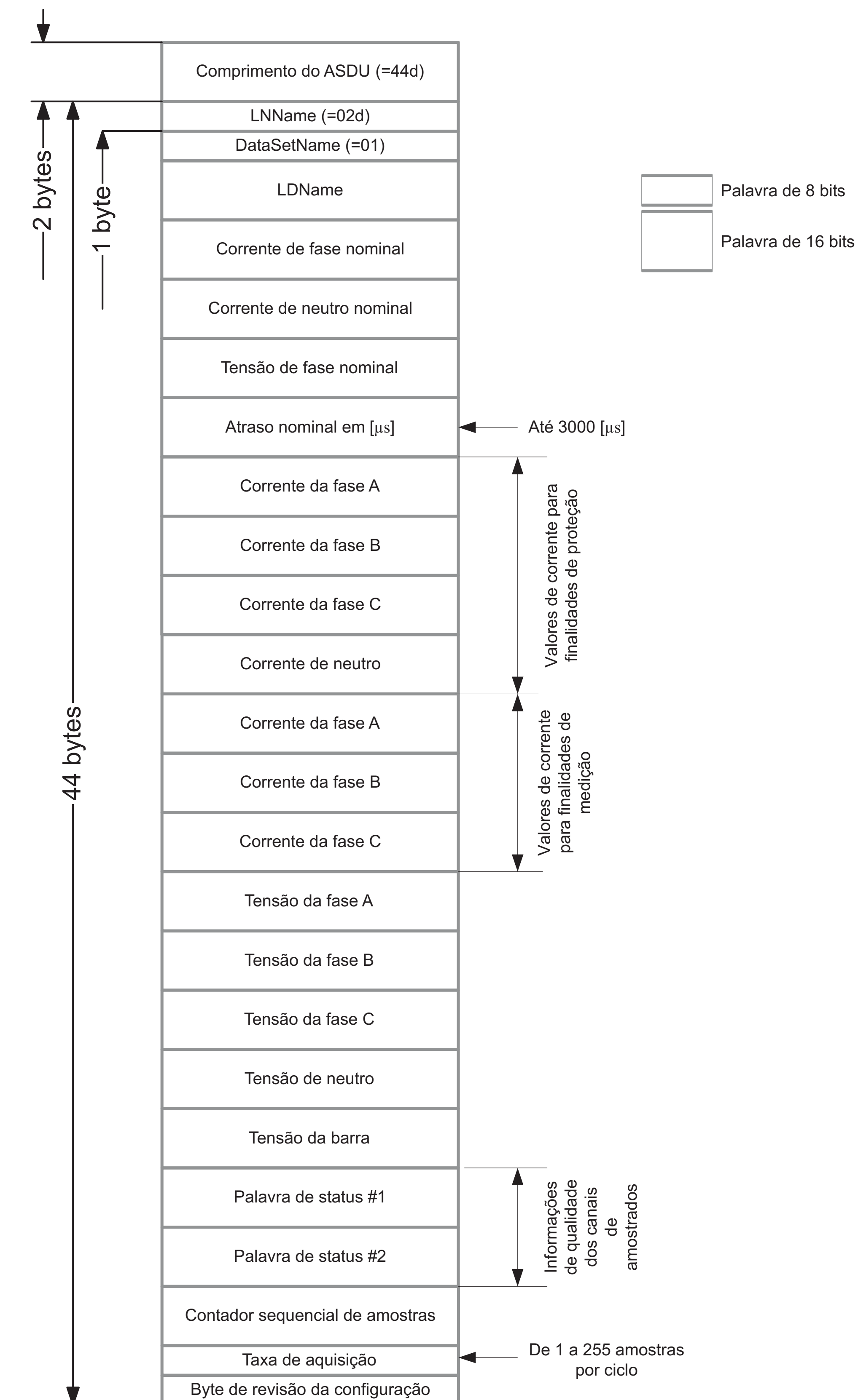

Figura 22: Universal Dataset enviado no ASDU de um pacote GOOSE de valores amostrados.

Como no caso do bloco de recebimento GOOSE-SV, foi feita uma rotina externa ao arcabouço, executada junto das demais rotinas de recebimento de pacotes de rede. A rotina aguarda a chegada de um pacote GOOSE pertinente, compara os valores dos *datasets*, *GOOSEId*, *GooseControlBlockReference*, etc. Caso as informações sejam do fornecedor configurado pelo usuário, as informações do *dataset* de dados booleanos são interpretadas e copiadas para uma memória interna do bloco dentro do *framework*. Na próxima execução de uma iteração (disparada pelo recebimento de um pacote SV), o arcabouço utiliza os dados recebidos no processamento de seu fluxo de dados.

Envio de dados GOOSE de automação

O bloco de envio de mensagens do tipo GOOSE, para barramento de automação (*station bus*), foi criado segundo a especificação da IEC 61850, partes 8-1 e 7-2. Nesse arcabouço foi implementado um bloco de envio GOOSE simplificado, capaz de concatenar até 8 estados booleanos em um *dataset*, para envio pela rede de comunicação. Não é dado suporte a outros tipos de *datatypes* definidos pela norma, tais como: qualidades, estampas de tempo ou valores analógicos.

No caso desse bloco, toda a rotina de envio foi codificada dentro das funções de execução e pós-execução do arcabouço. Dessa forma, durante a execução do fluxo de dados, as informações a serem enviadas são copiadas para dentro de uma área de memória reservada, onde já existe o restante do esqueleto de dados do pacote GOOSE. Ao final da iteração, tal área de memória é sinalizada para envio pelo canal de comunicação, conforme a necessidade.

Deve-se atentar que o envio de mensagens GOOSE convencionais (não SV) não possui confirmação de recebimento da informação por seus consumidores. Para garantir a chegada dos dados ao seu destino, o envio de mensagens possui uma máquina de estados própria, descrita na norma, que executa o envio de uma mesma mensagem repetidas vezes, em avalanche, com uma cadência temporal de tempos de envio crescentes, com envoltória exponencial, até um período máximo. Dessa forma, na mudança de algum de seus estados booleanos, o *framework* instrui o envio imediato de um pacote GOOSE e agenda o envio de uma confirmação, temporizada inicialmente para 4,0 [ms]. Assim que esse tempo agendado expira, o bloco programa o envio de uma nova confirmação, mas com o dobro do período agendado anteriormente resultando, no caso, em 8,0 [ms] para a terceira repetição. Essa lógica persiste, até que o tempo agendado atinja um limite máximo, costumeiramente chamado de "tempo de manutenção".

Restrições de blocos de interface

Deve-se atentar que todos os blocos de interface de entradas e saídas, bem como os blocos de envio e recebimento de mensagens GOOSE e GOOSE-SV mostrados anteriormente, possuem restrições quanto ao seu instanciamento dentro da aplicação no *framework*. Isso significa que um interpretador já nasce com uma determinada quantidade pré-definida desses blocos, como por exemplo, um bloco com um conjunto de 8 entradas de uso geral, um bloco com 8 saídas de uso geral, um bloco de interpretação de pacotes GOOSE-SV, um bloco de envio de 8 estados booleanos em pacote GOOSE convencional, e um bloco para recebimento de até 8 estados booleanos provenientes de uma mensagem GOOSE convencional.

Essas restrições se devem às características das áreas de memória estáticas que tais blocos possuem dentro do código-objeto produzido, que devem estar posicionados em uma região fixa, conhecida e acessível pelo restante do *software* que executa o interpretador, para que seja possível a troca de informações com o fluxo de dados do arcabouço.

Entretanto, os interpretadores podem ser modificados para possuírem quantos blocos de interface o quanto forem necessários, até o limite da capacidade de memória da plataforma de *hardware* ou até o esgotamento de sua capacidade computacional para gerenciamento desse blocos.

Outros blocos de suporte ao desenvolvimento

Para facilitar o trabalho do engenheiro que cria as aplicações no *framework*, alguns blocos foram produzidos para auxiliá-lo no processo de desenvolvimento e depuração do fluxo de dados, tais como:

- Bloco de monitoramento de grandezas: responsável por coletar e mostrar na tela variáveis e estados do sistema;
- Bloco de registro de oscilografias: responsável por coletar um registro de longo prazo de um determinado conjunto de variáveis internas, armazenando os registros em memória durante a execução do *framework*, para posterior captura e armazenamento em disco em arquivos em formato MatLab, *Comma Separated Values* (CSV) ou IEEE *Common Format for Transient Data Exchange* (COMTRADE);
- Bloco de registro de eventos: responsável por capturar os momentos das transições ou mudanças detectadas em algumas variáveis de interesse do sistema, registrando

os dados em memória, para posterior transferência e armazenamento em disco em formato CSV, e;

- Bloco de geração de sinais: responsável por sintetizar sinais em várias naturezas (senoidais, ondas dente-de-serra, triangulares, pulsos modulados tipo PWM e valores constantes), com parâmetros ajustáveis de intensidade, frequencia, fase, *duty-cycle*, etc.

Tais blocos são específicos para o auxílio nas tarefas de análise do algoritmo ou então para emular as entradas de dados e outros processos que só existiriam no interpretador executado na plataforma final em tempo real. Costumeiramente eles só são utilizados quando o *framework* é interpretado em modo *offline* na plataforma do engenheiro de desenvolvimento, mas podem ser utilizados também dentro da aplicação final, conforme a necessidade e a disponibilidade de recursos computacionais.

5.2.2.3 Cenário de aplicação do IED

O IED "A" desenvolvido com esse arcabouço será aplicado em um cenário de seletividade lógica, com falha de disjuntor, como mostrado na Fig.23.

Na Fig.23, o IED "A" está posicionado para a proteção do trecho entre seu disjuntor e o disjuntor do IED Jusante, de um sistema de distribuição radial. Acima do IED "A" existe o IED Montante, responsável pela proteção do outro trecho de rede mostrado. Todos os IED's possuem recursos de envio de mensagens rápidas tipo GOOSE, de barramento de automação (*station bus*) segundo a IEC 61850, estando conectados a um mesmo barramento de comunicação em rede, padrão Ethernet.

No local onde está instalado o IED "A" existe uma unidade concentradora, *Merging Unit*, capaz de digitalizar de forma síncrona, a 16 amostras por ciclo, as tensões e correntes do sistema de distribuição, logo a jusante de seu disjuntor. As amostras são coletadas, condicionadas e codificadas segundo o padrão do *Dataset Universal* descrito pela norma IEC 61850, parte 9-1.

As informações são enviadas para a mesma rede Ethernet onde estão conectados os IED's descritos anteriormente. Dessa forma, há a integração dos barramentos de processo (*process bus*) e de automação (*station bus*), fazendo com que todas as mensagens GOOSE publicadas (do tipo *multicast*), por cada IED e pela *Merging Unit*, circulem por toda a rede de comunicação, de forma irrestrita.

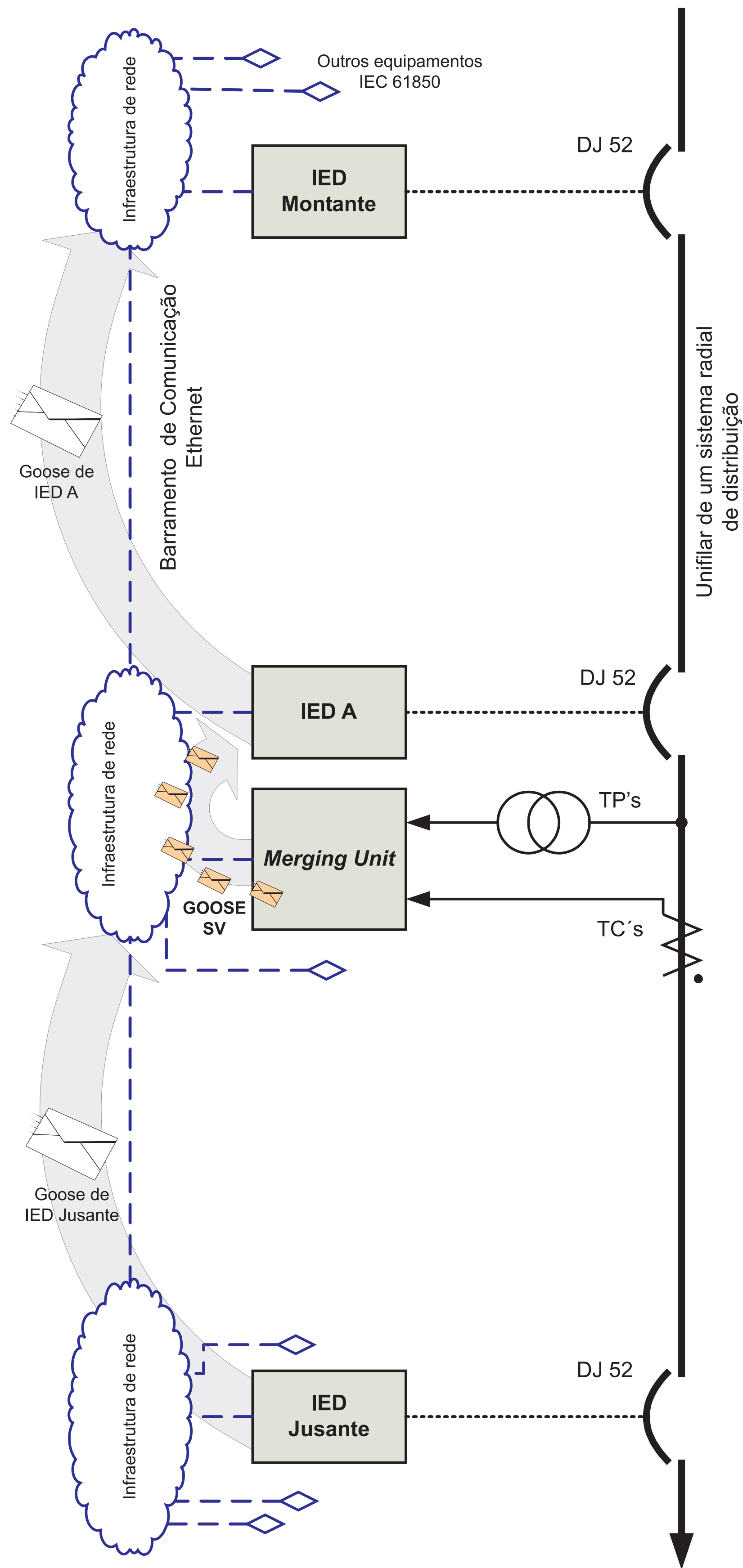

Figura 23: Cenário de aplicação do IED desenvolvido nesse trabalho.

5.2.2.4 Lógica de proteção e fluxo de dados desenvolvido

A lógica de proteção do IED "A", nesse contexto, é enumerada a seguir:

1 O IED "A" deve permanecer monitorando a corrente de uma das fases do sistema de distribuição, através do recebimento e processamento de pacotes GOOSE SV provenientes da *Merging Unit*, através da rede Ethernet;

2 Em paralelo, o IED "A" também deve monitorar uma mensagem GOOSE produzida pelo IED a jusante de sua posição, para verificar a presença de algum sinal de bloqueio de seletividade lógica proveniente desse outro equipamento;

3 No momento de ocorrência de uma falta na rede elétrica, quando a corrente monitorada assumir um valor superior ao limite (*threshold*) pré-ajustado, o IED "A" deve enviar imediatamente um bloqueio de atuação por seletividade lógica (ANSI 68) para outros IED's a montante de sua posição, através da sinalização de um estado em uma mensagem GOOSE para esses equipamentos;

4 Enquanto a corrente ainda está acima do limiar ajustado, o IED "A" deve aguardar um tempo pré-programado para então executar o *trip* de seu disjuntor, na tentativa de extinguir o defeito. Esse *trip* só irá ocorrer se o dispositivo a jusante não estiver enviando um sinal de bloqueio por seletividade lógica para o IED "A";

5 Caso o IED "A" esteja sendo bloqueado por outro IED a jusante, sua operação de *trip* é inibida. Entretanto, no momento de liberação desse bloqueio (atuação da função de falha de disjuntor no IED a jusante), é desejado que ocorra o *trip* imediato do disjuntor do IED "A";

6 Uma vez ocorrido o *trip*, o IED "A" deve manter seu disjuntor aberto e em *lockout* (ANSI 86), ou seja, com seu fechamento manual inibido. Para restaurar o modo normal de operação, um operador deve "resetar" a função de *lockout* ANSI 86 através da IHM;

7 Caso ocorra o *trip* do disjuntor do IED "A" e a corrente de falta não seja extinta por até 100,0 [ms], o dispositivo deve acusar a falha de seu disjuntor e deve liberar o estado de bloqueio ANSI 68 que está sendo enviado aos IED's a montante. Esses IED's, uma vez liberados para atuação, tentarão fazer o *trip* com o intuito de eliminar a falta em definitivo.

Segundo essa especificação funcional, foi elaborado o diagrama de blocos mostrado na Fig.24, utilizando os blocos *template* presentes nessa versão especial do *framework* "TB". Tal esquema de blocos, quando descrito na metalinguagem do arcabouço, resulta na listagem mostrada a título de curiosidade no Apêndice D.

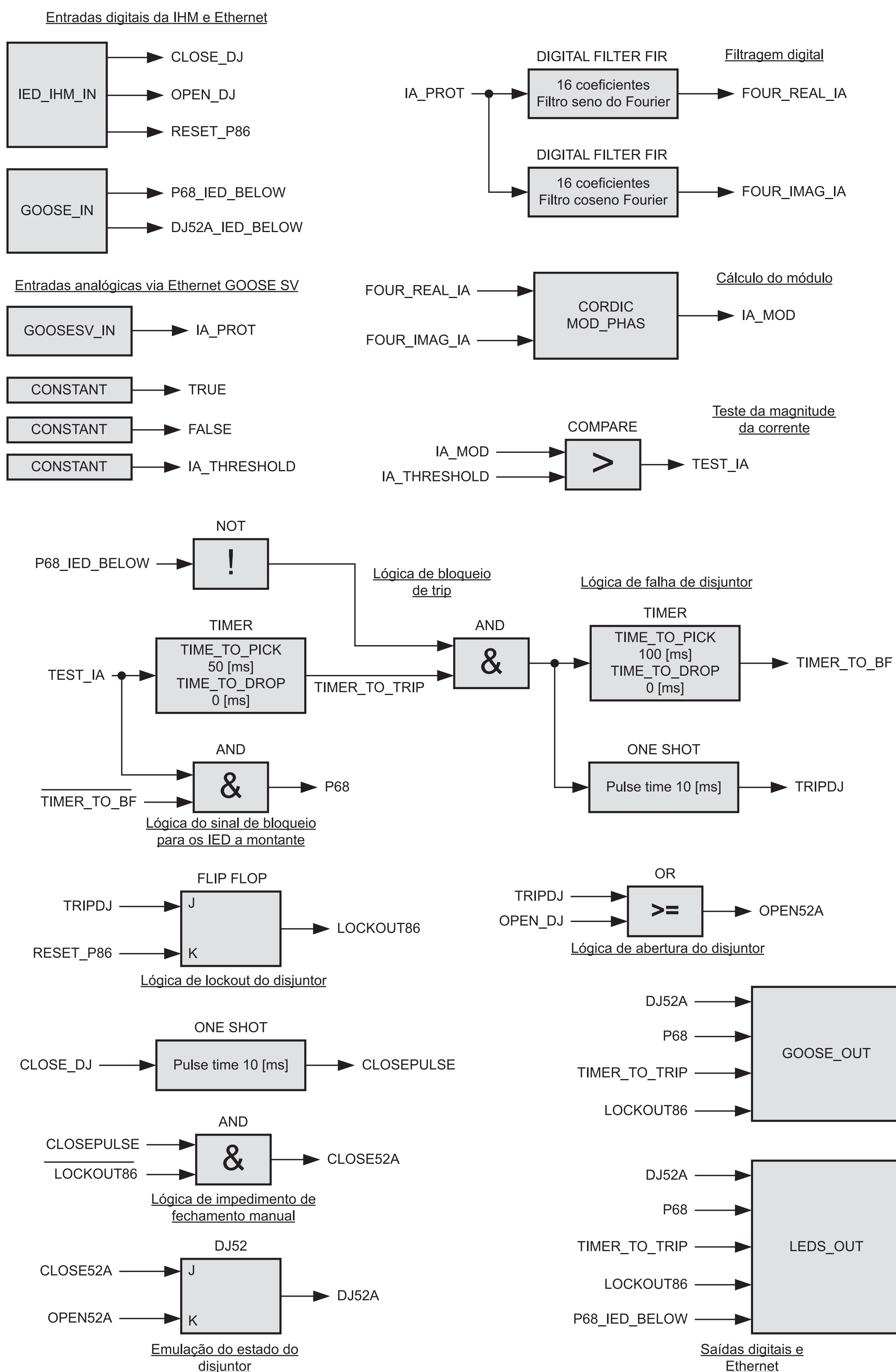

Figura 24: Diagrama de blocos do IED “A” especificado.

Na Fig.24 foi incluido um bloco para simulação interna do estado do disjuntor monitorado pelo IED "A", sendo constituído por um *Flip-Flop*.

Na Fig.24 foram omitidas as ligações de alguns blocos, tais como sinais de habilitação (*enable*) ou inibição (*disable*) presentes em *timers*, *flip-flops*, etc. No caso, todos esses sinais estão ligados a blocos de constantes, como o TRUE e FALSE mostrados.

Na Fig.24 também foram omitidos os blocos de função de oscilografia e registro de eventos para a simplificação do desenho. Entretanto, tais blocos foram utilizados para obtenção dos registros dos testes mostrados adiante nesse capítulo.

5.2.3 Testes da aplicação "TB" do IED de proteção

Os blocos mostrados na Fig.24 foram criados no *framework* "TB" utilizando os blocos elementares especialmente desenvolvidos, resultando em um total de 24 blocos, com as seguintes quantidade de rotinas internas agendadas para execução:

- 24 Rotinas de inicialização, executadas apenas durante o *startup* do dispositivo;
- 17 Rotinas de execução;
- 10 Rotinas de pós-execução;
- 1 Rotinas de finalização.

5.2.3.1 Testes no interpretador *offline*

A aplicação criada para o *framework* "TB" foi executada inicialmente em um interpretador criado para o ambiente de desenvolvimento sob o sistema operacional Windows, sem os requisitos de tempo real e sem o envio ou recebimento de mensagens Ethernet.

Em caráter de curiosidade, foram efetuadas medidas de tempo específicas com o registrador interno TSC da arquitetura "x86", para captura e análise do melhor caso de tempo de execução de cada um dos blocos instanciados nessa aplicação. Os resultados, em termos de números de ciclos e em [μs] são mostrados na Tabela 4

Tabela 4: Detalhes do tempo de execução de alguns blocos para uma iteração de processamento do arcabouço "TB".

	Tempo em ciclos	Tempo em [μs]
Filtro digital	541	$0,338$
Vetor CORDIC	422	$0,264$
Timers	71	$0,044$
Logicas	93	$0,058$
OneShot Pulse	70	$0,044$
Flip-Flop	104	$0,065$

Os números mostrados na Tabela 4 não incluem a sobrecarga operacional de chamada dos blocos, estimada em cerca de 115 ciclos no Windows XP e 34 ciclos no Linux com RTAI em tempo real.

Devido às características do sistema operacional Windows, tais dados são aproximados e possuem um erro de cerca de ± 10,0 [%]. Mesmo assim pode-se notar em quais blocos há o maior custo computacional - como são os casos do bloco de filtragem digital e vetor CORDIC, e em quais blocos há a maior ineficiência devido a sua baixa granularidade.

5.2.3.2 Testes no interpretador em tempo real - plataforma PC

O mesmo conjunto de blocos criado foi submetido a testes em uma plataforma de execução em tempo real com a configuração (d) mostrada no início do capítulo.

Um outro computador, conectado à rede Ethernet com essa plataforma de testes, executou uma versão de um gerador de pacotes, especialmente desenvolvido, capaz de simular a cadência de envio de dados de uma *Merging Unit* real. Durante a geração de pacotes, esse simulador de *Merging Unit* emite dados sintetizados, de grandezas senoidais, a 16 amostras por ciclo, considerando uma rede em 60,0 [Hz], gerando sinais de 4 correntes

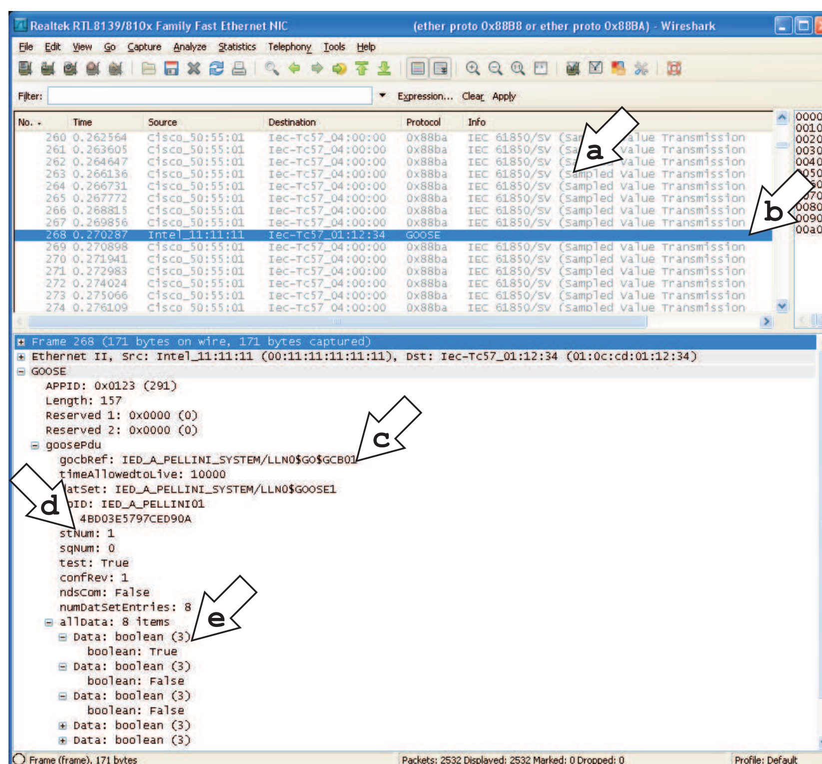

Figura 25: Pacotes GOOSE SV e GOOSE *station bus* capturados pelo *software* Wireshark.

para finalidades de proteção, de 3 correntes para finalidade de medição e de 4 tensões, conforme o conteúdo do *Dataset Universal* mostrado na Fig.22.

Enquanto isso, o interpretador TB, em tempo real, realiza a coleta desses dados, o seu processamento e o envio dos pacotes de saída tipo GOOSE. Alguns desses pacotes foram capturados com o auxílio do programa WireShark, versão 1.2.8 e são mostrados na Fig.25 com alguns detalhes salientados por meio de setas nomeadas de (a) a (e).

Na Fig.25, em (a) são ressaltados os pacotes de *Sampled Values* da IEC 61850, enviados pelo módulo de simulação da *Merging Unit*.

Em (b) é mostrado um pacote GOOSE enviado para o barramento de automação (*station bus*) pelo *framework* “TB” com seu bloco interno de envio de mensagens GOOSE IEC 61850.

Em (c) são mostradas as informações do cabeçalho da mensagem GOOSE capturada. Notam-se os nomes dos *datasets*, *GooseID* e outros parâmetros relevantes, para identificação dessa mensagem GOOSE na rede Ethernet por outros instrumentos e IED’s que necessitam de seu conteúdo.

Em (d) são mostrados os parâmetros de repetição (*Sequence Number*, ou *sqNum*) e número do estado atual do *dataset* (*State Number*, ou *stNum*), dados relevantes para a

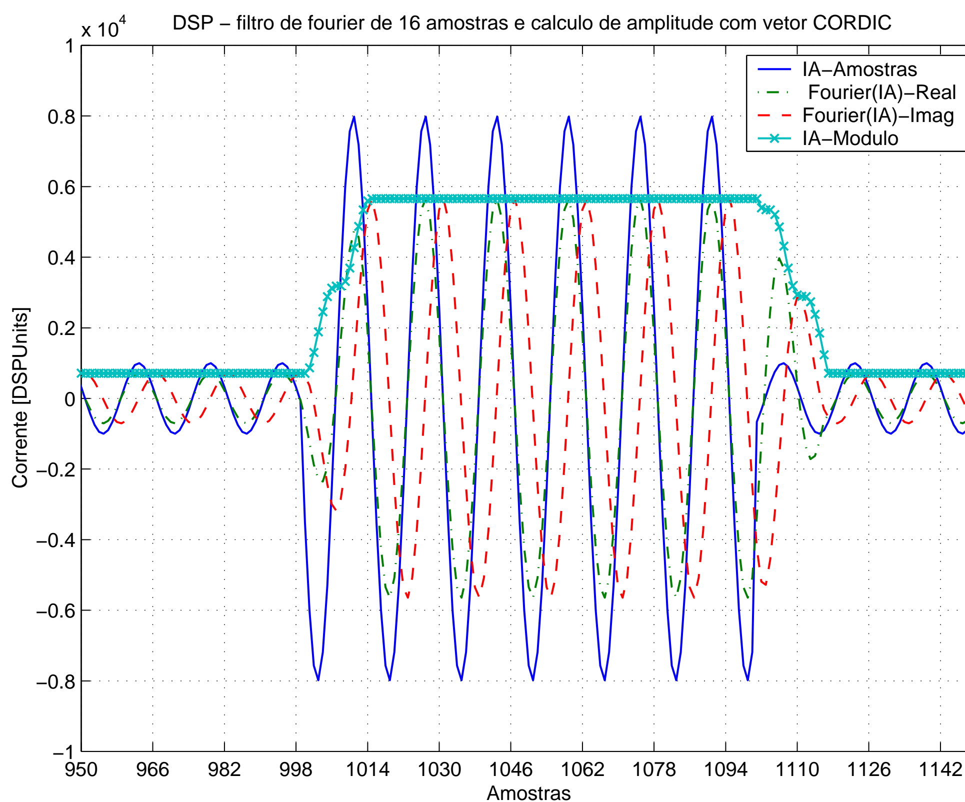

Figura 26: Resultado do processamento digital de sinais para tratamento da corrente da fase A.

identificação de um novo evento sendo retratado na rede, ou apenas uma de suas confirmações.

Em (e), finalmente, é mostrado o *dataset* contendo os 8 *dataitens* booleanos enviados dentro do pacote.

O programa simulador da *Merging Unit* criado permite a geração de um evento de sobrecorrente em uma determinada fase do sistema, com um intervalo de tempo pré-definido. Tal sinal foi utilizado para testar o fluxo de dados desenvolvido.

Através da rotina de oscilografia interna do *framework*, foi coletado um registro de vários sinais internos de interesse. Esses registros são mostrados nas Figs. 26 e 27.

Na Fig.26 são mostrados os resultados do processamento digital de sinais efetuado pelos filtros digitais FIR e pelo bloco de cálculo de amplitude VetorCordic utilizado. Pode-se notar o valor RMS da amplitude da onda senoidal de corrente ao longo do tempo, incluindo seu transitório durante o evento de sobrecorrente.

Na parte superior da Fig.27 são mostrados o sinal de corrente da entrada, a medida da amplitude desse sinal e o limiar programado para *pickup* do elemento de proteção.

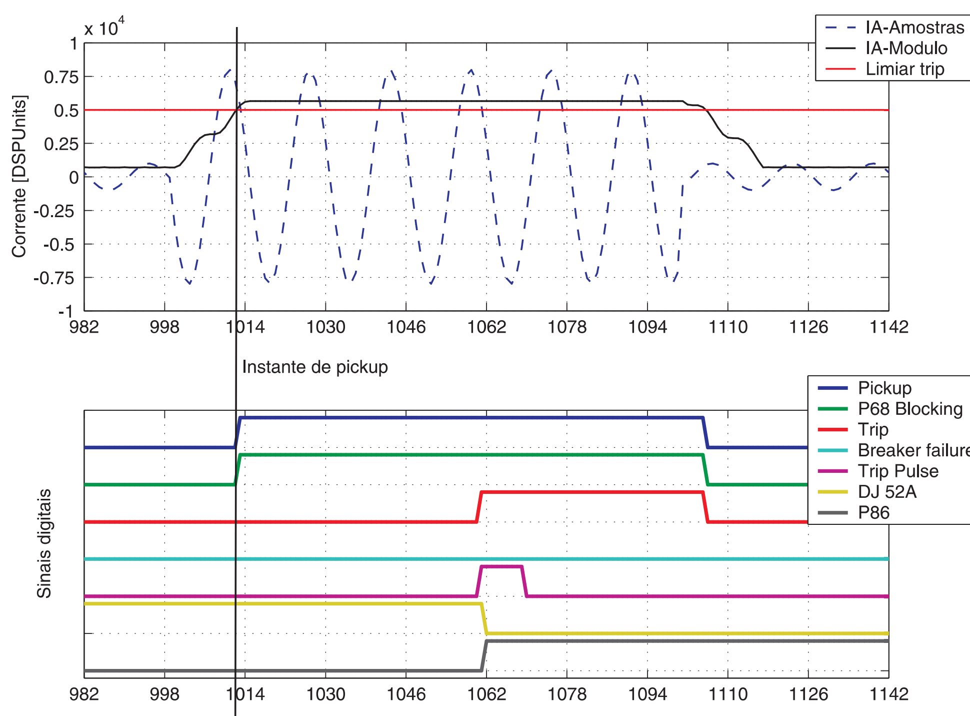

Figura 27: Sinais digitais internos da lógica de proteção implementada.

Na parte inferior são mostrados, na sequência das legendas, os sinais digitais internos ao fluxo de dados. Pode-se observar o sinal de *pickup* e o sinal de bloqueio P68 partindo no mesmo instante, o sinal de *trip* do elemento, ocorrendo com 48 amostras após a detecção da falta (praticamente os 50,0 [ms] programados). Também podem ser vistos o sinal de *lockout* (P86) e o sinal de disjuntor aberto (DJ 52A = 0) no momento do trip.

Como o simulador de *Merging Unit* trabalha em malha aberta, ele não tem como detectar a abertura do disjuntor simulado internamente no IED. Dessa forma, após o *trip*, a corrente não é extinta. Como o transitório de corrente causado pelo simulador apresenta um tempo de falta de apenas 100,0 [ms], não há tempo suficiente para que a lógica de falha de disjuntor atue e retire o bloqueio P68 enviado aos demais IED's.

É importante frisar que esses resultados podem ser reproduzidos com o *framework* em execução *offline*, bastanto para isso incluir um bloco de geração de sinais internos para simular a recepção de amostras provenientes da *Merging Unit*.

A latência para envio das mensagens pelo conjunto sistema operacional mais interpretador "TB" não pode ser estimado de forma precisa com um simulador tão simples da *Merging Unit*. Entretanto, pode-se notar que o desempenho geral do mecanismo de envio de mensagens GOOSE é insatisfatório na configuração (d), devido a várias evidências de mensagens GOOSE que deveriam ter sido enviadas em tempos muito próximos, como durante a mudança de estados, mas que apareceram no *software* de captura como

Figura 28: Pacotes GOOSE enviados na cadência errada devido a operação da rede Ethernet do computador PC, sem requisitos de desempenho em tempo real.

se tivessem sido enviadas em avalanche (*burst*). Isso não acontece com frequência, mas pode impactar no desempenho global do sistema de proteção, ao atrasar o envio de um pacote de bloqueio por exemplo, justamente durante um evento na rede elétrica. Esse fato pode ser visto na Fig.28 com algumas detalhes salientados por meio de setas (a) e (b) mostradas.

Na Fig.28, em (a) é mostrado o primeiro pacote, que sinaliza a mudança de estado do sistema. Em (b) é apontado o 5° pacote enviado, que deveria estar na rede cerca de 32,0 [ms] após o primeiro pacote, considerando-se a cadência de tempos exponencial esperada, ou seja, os seguintes tempos:

- Pacote 1, com t = 0,0 [ms] - instantâneo;
- Pacote 2, com t = 4,0 [ms];
- Pacote 3, com t = 8,0 [ms];
- Pacote 4, com t = 16,0 [ms], e;
- Pacote 5, com t = 32,0 [ms].

Entretanto, pode-se notar, pelos tempos de chegada registrados no *software* Wireshark, que quase todos os pacotes foram enviados em uma janela de pouco mais de 1,5 [ms].

Isso foi investigado com mais cuidado e pôde-se notar que existe a necessidade de se modificar a camada de rede do Linux com o RTAI, de forma a obter também um desempenho de execução em tempo real. Isso pode ser feito com um pacote *patch* denominado RTNET. Sem esse pacote, a camada de rede do SO opera somente nos instantes em que há ociosidade na interface e nos instante que o escalonador preemptivo do sistema operacional Linux permite. No caso mostrado na Fig.28, os primeiros pacotes GOOSE ficaram

Figura 29: Kit de desenvolvimento com processador ARM9 STR912 da STMicroelectronics, excutando o *framework* "TB" desenvolvido.

presos em memórias internas (*buffers*) da pilha de comunicação do sistema operacional, até que as camadas de serviço de rede fossem executadas pelo *kernel*.

Para os pacotes GOOSE de manutenção, quando não ocorre nenhum evento de mudança de estado, foi detectado um desvio/jitter máximo no período de envio dos pacotes de cerca de 30,0 [ms], durante um período de análise de pouco mais de 5 minutos (300 pacotes de manutenção, enviados a cada 1,0 [s]).

5.2.3.3 Testes no interpretador em tempo real - plataforma ARM

O mesmo conjunto de blocos foi submetido a testes em uma plataforma de execução em tempo real com a configuração (e) mostrada no início desse capítulo. Uma foto desse *hardware* é mostrada na Fig.29.

Um outro computador, conectado pela rede Ethernet com essa plataforma de testes microcontrolada, executou a mesma versão do gerador de pacotes / simulador de *Merging Unit* usado anteriormente.

Nessa plataforma, o interpretador TB, em tempo real, opera sob um sistema executivo dedicado, onde as tarefas de recepção e envio de pacotes de rede são controladas de forma praticamente direta pelo *framework*, sem drivers ou rotinas de controle de acesso, etc. Dessa forma, o desempenho no envio de mensagens se comportou como esperado, com a envoltória de envio de pacotes GOOSE obedecendo as temporizações programadas de 0,

4, 8, 16, 32, etc. [ms]. De qualquer forma, foi notado que várias melhorias podem ser feitas no esquema de interrupções do microcontrolador, para que seu desempenho seja ainda melhor que o verificado.

Nessa plataforma foi possível medir a ocupação computacional de toda a aplicação em execução, através da medida em osciloscópio do período de alguns sinais digitais acionados pelo microcontrolador durante a execução de suas tarefas. Da mesma forma, foi possível verificar a latência para envio de pacotes GOOSE, conforme citado pela norma IEC 61850, parte 8-1. Os resultados são mostrados na Tabela 5.

Tabela 5: Tempos obtidos para execução do *framework* no microcontrolador ARM9.

	Tempo em [μs]	Jitter em [μs]
Execução da aplicação no *framework* "TB"	210	20
Latência para envio de pacotes GOOSE	$80,3$	$32,2$
Latência para recebimento de pacotes GOOSE	$12,3$	$0,81$
Tempo total de comunicação (IEC61850)	114	$31,1$
Turnaround Time com *framework*	335	$60,2$

Na Tabela 5, o tempo para execução do *framework* foi medido entre o início das rotinas de execução dos blocos até o término das rotinas de pós-execução dos mesmos. O *jitter* observado é devido a outras interrupções concorrentes em execução no microcontrolador, que eventualmente ocorrem durante a interrupção da aplicação.

Notou-se que alguns blocos podem ser otimizados para utilizarem melhor os recursos do microcontrolador ARM, que nesse caso, possui até suporte em *hardware* ao processamento digital de sinais com alto desempenho, recurso que atualmente não está sendo explorado. Outra melhoria que pode ser feita é a criação de blocos contendo uma maior quantidade de algoritmos. Esse aumento de granularidade, como comentado anteriormente, permite evitar ou controlar o *overhead* das chamadas das rotinas dos blocos. Um exemplo seria a codificação de toda a rotina de proteção ANSI 51, que poderia ter sua lógica de comparação e sequenciamento programadas de forma monolítica, dentro de um

único bloco, ao invés de se utilizar dos blocos elementares do arcabouço.

De qualquer forma, segundo as especificações criadas, esse microcontrolador pode executar o algoritmo trifásico de proteção de sobrecorrente ainda dentro dos limites de tempo de processamento impostos anteriormente na Eq. (5.7).

Na Tabela 5, a latência para envio dos pacotes GOOSE foi medida entre o ponto de sinalização, dentro da aplicação do *framework*, para envio do pacote GOOSE e sua efetiva chegada para serialização no PHY Layer (camada física da rede Ethernet) no *hardware*.

Na Tabela 5, a latência para recebimento dos pacotes GOOSE foi medida entre o ponto de sinalização no PHY Layer do kit de desenvolvimento (camada física da rede Ethernet) no *hardware*, e a efetiva chegada da informação dos pacotes, nos *buffers* de memória dentro do microcontrolador.

Segundo a norma IEC 61850, em sua parte 5, capítulo 13, o tempo total de transmissão entre o início de envio de uma mensagem de rede (em um processador de comunicação de um lado da rede Ethernet) e o final do recebimento desse pacote (em outro processador de comunicação em outro ponto da rede) deve possuir um tempo máximo que atenda a determinados pré-requisitos, dependendo da aplicação. Segundo a norma, os requisitos mais severos são para as mensagens rápidas Tipo 1A ("Trip"), com dispositivos da classe P2 e P3 (elementos de proteção para *bays* de transmissão de energia). Nesses casos, segundo a IEC 61850, esse tempo total de comunicação deve ser inferior a 3,0 [ms]. Um teste foi efetuado entre duas placas de kit de desenvolvimento executando o arcabouço desenvolvido, medindo-se o tempo entre um envio e um recebimento entre as placas, conectadas por um *switch* de comunicações Gigabit Ethernet 3COM, mas com a rede em 100 [Mbps] (capacidade máxima dos kits). O tempo total de comunicação mostrado na tabela anterior mostra que o dispositivo de proteção criado atende plenamente aos requisitos de tempo da norma IEC 61850 (0,114 [ms] $<$ 3,0 [ms]).

No final da Tabela 5, foi testado o *turnaround time* para a aplicação desenvolvida, que consiste no tempo médio entre receber uma informação via mensagem GOOSE e devolver outra informação em outra mensagem, considerando o tempo de processamento interno de toda a proteção e sua lógica. Os tempos resultantes são satisfatórios, considerando que o IED apresenta somente a lógica de proteção para uma das fases de corrente do sistema elétrico. Entretanto, através de uma melhor otimização da aplicação e seus blocos, pode-se melhorar de forma considerável todo esse desempenho.

5.3 Comentários finais dos resultados

O objetivo de criar rapidamente uma determinada aplicação embarcada, sem os transtornos de codificação direta em linguagens como ANSI C e Assembly, foi atingida com sucesso com o uso do *framework* "TB" desenvolvido.

Os melhores resultados foram obtidos nas plataformas IBM PC, devido a grande disponibilidade de recursos e capacidade de processamento dessas arquiteturas.

A tarefa de codificar o *framework* e seus blocos, ainda demanda conhecimentos específicos de programação para as arquiteturas desejadas. Entretanto, esse trabalho, uma vez realizado, pode ser amortizado em outros projetos, desde que sejam utilizadas as mesmas plataformas. Por essa razão, uma das principais regras, para o sucesso na implantação desse tipo de ferramenta, é a escolha de uma plataforma adequada e versátil.

Ainda, é importante que seja designada uma plataforma que contenha tanto um *hardware* quanto um *software* básico de sistema operacional ou sistema executivo, ambos poderosos e capazes de manter o desempenho desejado em tempo real. Nos resultados obtidos, um sistema operacional em tempo real, mas com sua camada de rede implementada de forma não determinística, mostrou o quão importante é essa escolha na qualidade dos resultados.

O *framework* impõe uma penalidade em desempenho. Entretanto, seus benefícios em tempo de desenvolvimento são marcantes. Toda a aplicação do IED foi desenvolvida em 5% do tempo gasto para o desenvolvimento dos blocos do *framework* (3 horas de trabalho na aplicação *versus* 60 horas de trabalho total para desenvolvimento dos blocos do arcabouço). O desenvolvimento no computador *desktop* PC e no ambiente embarcado foram feitos com sucesso. Os resultados em ambos os cenários de desenvolvimento puderam ser obtidos e comparados de forma simples.

6 Conclusão

6.1 Comentários gerais

Os problemas de desenvolvimento de *software* são amplamente conhecidos e estudados na literatura. É um consenso geral que a melhor abordagem para se diminuir custos e tempo de mercado envolve a escolha de diversas ferramentas e metodologias de apoio. Uma dessas abordagens é a utilização de arcabouços de *software*.

Essa afirmativa é corroborada por diversos relatos e experiências do emprego de arcabouços, encontradas na literatura técnica. Entretanto, de todos os arcabouços, metodologias ou produtos existentes, nenhum se adapta exatamente ao contexto e requisitos de aplicações computacionais em tempo real determinístico dos cenários de Sistemas de Potência. Ou as soluções não alcançam o desempenho necessário, ou não são portáveis para pequenas plataformas embarcadas, ou apresentam custos elevados ou outros fatores limitantes.

Nesse contexto foi criado com sucesso um *framework* de *software* baseado em premissas semelhante às encontradas na síntese comportamental de *hardware* em alto nível e em normas como a IEC 61131 e IEC 61499. A ferramenta auxilia nos trabalhos de criação e desenvolvimento de *software*, principalmente na prototipagem de sistemas e em sistemas que precisam ser continuamente reconfigurados ou reparametrizados.

O arcabouço é constituído por ferramentas de programação e desenvolvido e ferramentas e rotinas de compilação e interpretação em tempo real. As ferramentas de programação consiste em uma coleção de blocos, desenhados especialmente para o domínio da aplicação desejada, e uma metalinguagem de interconexão e relacionamento desses blocos. Essa metalinguagem é então compilada em um código-objeto compacto e monolítico, que é então interpretado e executado em tempo real, por rotinas criadas especificamente para o *hardware* final desejado.

A ferramenta criada permite o desenvolvimento *offline* dos algoritmos e sua posterior transferência para o ambiente de execução sem que sejam necessárias modificações em seu fluxo de dados. Durante esse desenvolvimento *offline*, mais detalhes do funcionamento da

aplicação podem ser monitorados através de blocos específicos de depuração e análise.

A utilização de linguagem C para a criação das ferramentas permitiu bons resultados de portabilidade e desempenho, em pelo menos duas plataformas diferentes, as arquiteturas IBM PC e ARM, de 32 bits.

O desempenho do interpretador do *framework* é muito bom se comparado ao de outras ferramentas interpretadas e compiladas. A sobrecarga operacional do interpretador criado é mínima. O *footprint* de memória RAM e ROM do código-objeto e das rotinas do interpretador em um dispositivo embarcado é pequeno, permitindo seu uso em um ampla gama de microcontroladores e DSP's.

A granularidade dos blocos *template* possui impacto no desempenho global do sistema. Cenários com muitos blocos atômicos são bem menos eficientes que cenários utilizando menos blocos, mas com algoritmos internos maiores (mais densos).

Os testes do núcleo algébrico do arcabouço mostram que os resultados são coerentes e idênticos aos obtidos por ferramentas de simulação tradicionais. Isso demonstra que os critérios e algoritmos de ordenação aplicados são adequados para os fluxos de dados mostrados.

Uma versão do arcabouço foi realizada especialmente para o cenário de aplicação de um relé digital de proteção de sistemas elétricos com recursos da IEC 61850. Com o *framework* foi criado um pequeno relé para execução da função de proteção de sobrecorrente temporizada monofásica, programável, com recursos de comunicação horizontal da norma IEC 61850 através de mensagens do protocolo GOOSE, tipo barramento de subestação, e mensagens do protocolo GOOSE SV, tipo barramento de processo. A implementação do relé foi feita considerando-se ambos os barramentos (estação e processo) implementados em uma única rede de comunicação, de forma pioneira.

A mesma lógica do relé foi executada em uma plataforma IBM PC e em um ambiente embarcado com microcontrolador com os mesmos resultados e desempenho, exceto pela rede de comunicação Ethernet que não apresentava *performance* em tempo real no caso computador IBM PC.

Os resultados obtidos com o relé de proteção são excelentes do ponto de vista do desempenho da rede de comunicação de mensagens GOOSE e GOOSE-SV, graças ao enfoque prioritário dado aos blocos responsáveis pelo processamento das mensagens GOOSE no dispositivo.

Esse *framework*, associado a uma plataforma de *hardware* e *software* sólidas, pode ser usado como base para o desenvolvimento de uma arquitetura completa de construção de dispositivos de proteção de sistemas elétricos de potência, customizados e flexíveis

conforme as necessidades da aplicação.

Esse tipo de solução do arcabouço pode ser empregada com sucesso em outras áreas da técnica tais como: Sistemas de Controle, Sistemas Inerciais e Processamento Digital de Sinais, mediante a extensão e criação de blocos específicos para tais domínios.

6.2 Desenvolvimentos futuros

Nota-se que diversos pontos podem ser evoluídos no *framework* desenvolvido, entre eles:

- Construir, no futuro, um ASIC para processamento e interpretação do *framework*;
- Os recursos de programação do *framework* podem ser melhorados pela adoção de um ambiente gráfico para desenho e parametrização dos diagramas de blocos;
- A linguagem de armazenamento e descrição textual pode ser portada, no futuro, para XML, aproveitando as capacidades de manipulação e edição de alguns programas mais modernos;
- Mais blocos podem ser implementados, e em outras plataformas, como PowerPC, para testes comparativos;
- Utilizar um sistema executivo multiprocessado em ambiente “x86” para verificar o melhor desempenho possível;
- Desenvolver um gerador de código-fonte para a linguagem C, para obtenção de melhores desempenhos com código compilado para linguagem de máquina;
- Incorporar as outras linguagens da IEC 61131-3, para criar um *engine* completo de PLC, reacendendo os esforços em se construir um PLC ou super IED’s, de arquitetura aberta;
- Aplicar ou adaptar as premissas de gerenciamento de *software* defendidas pela ECSS ou pela RTCA em seu DO-178B, para adequar o arcabouço a tais requisitos de documentação, permitindo também seu emprego em ambientes embarcados da indústria aeroespacial. Tal esforço tem como resultado a criação de um acervo de documentação valioso, que irá favorecer a distribuição do arcabouço em domínio público e o recebimento de futuras contribuições de terceiros, e;

- Tal esforço em se adaptar as diretrizes da RTCA e ECSS para o arcabouço criaria um precedente valioso na indústria de desenvolvimento de *software* e *firmware* embarcados para o setor elétrico, que carece de tais metodologias e pode se beneficiar fortemente dessas iniciativas.

APÊNDICE A – Exemplo de código fonte para declaração de um bloco

A seguir é mostrado o trecho completo de declaração de um bloco elementar (*template*) que realiza a integração numérica no domínio das amostras, com passo de tempo fixo dado pela taxa de execução da aplicação.

A.1 Arquivo header simIntegrator.h

```
#ifndef SIMINTEGRATOR_HEADER
#define SIMINTEGRATOR_HEADER

//COMMON PROJECT INCLUDES
/////////////////////////////////////////////////////////////////////////////
#include "..\..\common\general.h"
#include "..\blockdef.h"

//BLOCK BASIC DEFINITIONS
/////////////////////////////////////////////////////////////////////////////
#define NAME_SIMINTEGRATOR     "SIMPLE_INTEGRATOR"

#define SIMIN_AN_INPUT0_NAME   "INTEGRATOR_INPUT"
#define SIMIN_AN_OUTPUT0_NAME  "INTEGRATOR_OUTPUT"
#define SIMIN_PARAM0_NAME      "CFG_FLAGS"
#define SIMIN_PARAM1_NAME      "TICK_PRESET"
#define SIMIN_PARAM2_NAME      "TICK_TO_GO"
#define SIMIN_PARAM3_NAME      "INTEGRATOR_GAIN"
#define SIMIN_PARAM4_NAME      "INITIAL_VALUE"

#define ID_SIMINTEGRATOR       0xAABBCC05

//Block Structure///////////////////////////////////////////////////////////
typedef struct SimIntegratorData
{
    //Operational data
    TBLONG BlockType;   //ID/Type of the current block
    TBLONG BlockSize;   //Size of the current block

    //Config data
```

```
    TBLONG CfgFlag;     //Multi purpose configuration flags
                        //Configures operation mode, integration type, saturation
                        //behaviour, etc. - bitmaped - to be defined someday
    TBLONG TickPreset;  //Number of iterations to execute the integration

    //Runtime data
    TBLONG StatFlag;    //Multi purpose status flags
                        //Signals the integrator operational status - bitmaped -
                        //to be defined someday
    TBLONG TickToGo;    //Decimation counter of the integrator execution

    //Digital inputs

    //Digital outputs

    //Analog inputs
    TBANVAR *Input;     //Integrator input

    //Analog outputs
    TBANVAR NewOutput;  //New calculated output, before commit
    TBANVAR Output;     //Integrator output

    //Integrator runtime data (depends on the integration algorithm, and
    //other details from each implemented hardware)
    TBANVAR LastSample; //Last integrator input (for trapezoidal rule)
    TBANVAR Gain;       //Gain used on the integration process (takes into
                        //account the time-step, multipliers and dividers)
    TBANVAR IniValue;   //Initial value
} SimIntegratorData;

//Prototypes for real time simulator runtime
////////////////////////////////////////////////////////////////////////////
extern int SimIntegratorInit(long baddr, SimIntegratorData *Data);
extern void SimIntegratorRun(SimIntegratorData *Data);
extern void SimIntegratorCommit(SimIntegratorData *Data);

//Prototypes for simulator parser
////////////////////////////////////////////////////////////////////////////
extern void SimIntegratorDeclaration(BlockTemplateData *Template);

#endif
```

A.2 Arquivo de implementação simIntegrator.c

```
////////////////////////////////////////////////////////////////////////////
//Block Definition - SIMPLE INTEGRATOR
////////////////////////////////////////////////////////////////////////////

//COMMON PROJECT INCLUDES
////////////////////////////////////////////////////////////////////////////
#include "simIntegrator.h"

#ifdef RTSIM
//Block simulation support functions
////////////////////////////////////////////////////////////////////////////
int SimIntegratorInit(long baddr, SimIntegratorData *Data)
```

```
{
  //Check survival data
  if(Data->BlockType!=(ID_SIMINTEGRATOR))
    return ERROR_WRONG_BLOCK_ID;
  if(Data->BlockSize!=sizeof(SimIntegratorData))
    return ERROR_WRONG_BLOCK_SIZE;

  //Initial status
  Data->StatFlag=0;

  //LastSample definition
  Data->LastSample=0;

  //pointer arithmetic
  Data->Input = (TBANVAR *)((long)Data->Input + baddr);

  //Initial output definition
  Data->Output=Data->IniValue;

  return OK;
}

void SimIntegratorRun(SimIntegratorData *Data)
{
  if(Data->TickToGo)
    Data->TickToGo--;
  else
  {
    //integrates
    Data->NewOutput = Data->Output +
              ((*Data->Input) + Data->LastSample) * Data->Gain;
    //scrolls input buffer
    Data->LastSample =  *Data->Input;
    //signals new data to publish
    Data->StatFlag |= OUTPUT_CHANGE;
    //schedule next run
    Data->TickToGo=Data->TickPreset;
  }
}

void SimIntegratorCommit(SimIntegratorData *Data)
{
  if(Data->StatFlag & OUTPUT_CHANGE) //if new output is avaliable
  {
    Data->Output=Data->NewOutput;   //refresh output
    Data->StatFlag &= ~OUTPUT_CHANGE; //cut out the flag
  }
}

#else

//Block parser routines
////////////////////////////////////////////////////////////////////////////////
long SimIntegratorDataParser(long bnum, FILE *arq,
              BlockTemplateData *BlockTpl,
              TransBlockCase *SCase,
              TransBlockRuntime *Rt)
{
  SimIntegratorData *Int;
```

```
    long offnlinha;
73  char param[200];
    char orig[200];
75  long offset;

77  //points the simulator runtime data with the block pointer
    Int=(SimIntegratorData*)
79    &Rt->TransBlockData[SCase->Block[bnum].BlockOffset /
      sizeof(TRANSBLOCK_MEMORY_VAR)];
81
    //checks internal block alignment
83  if(Int->BlockType != 0 || Int->BlockSize !=0)
      InternalError("Incorrect block addressing for block %s, simulator block %ld\r\n",
85            BlockTpl->Block[SCase->Block[bnum].BlockTemplateNum].BlockName,bnum);

87  Verbose(VERB_PARSER_RUN,"\t\t%s parser running for simulator block #%ld\r\n",
         BlockTpl->Block[SCase->Block[bnum].BlockTemplateNum].BlockName,bnum);
89
    if(GetParameterFromCurrentSection(arq, SIMIN_AN_INPUT0_NAME, param, &offnlinha, orig))
91  {
      if(SearchBlockOutput(param, BlockTpl, SCase, bnum, &offset))
93    {
        Int->Input=(void*)offset;
95    }
      else
97    {
        ParserError(SCase->SimFile, offnlinha,
99            "Could not find entity specified [%s]\r\n",orig);
      }
101 }
    else
103 {
      Warning(WARN_INPUT_MISSING, SCase->SimFile, offnlinha,
105       "Block input not specificed... routing to null\r\n",
          SIMIN_AN_INPUT0_NAME);
107 }

109 if(GetParameterFromCurrentSection(arq, SIMIN_PARAM1_NAME, param,
                      &offnlinha, orig))
111 {
      Int->TickPreset=atoi(param);
113 }
    else
115 {
      Warning(WARN_PARAMETER_MISSING, SCase->SimFile, offnlinha,
117       "Block tick preset not specificed... assuming zero\r\n");
      Int->TickPreset=0;
119 }

121 if(GetParameterFromCurrentSection(arq, SIMIN_PARAM2_NAME, param,
                      &offnlinha, orig))
123 {
      Int->TickToGo=atoi(param);
125   if(Int->TickToGo<0)
      {
127     ParserError(SCase->SimFile, offnlinha,
              "Invalid Tick_To_Go = %ld\r\n",Int->TickToGo);
129   }
    }
```

```
131   else
      {
133     Warning(WARN_PARAMETER_MISSING, SCase->SimFile, offnlinha,
            "Block tick to go not specificed... assuming zero\r\n");
135     Int->TickToGo=0;
      }
137
      if(GetParameterFromCurrentSection(arq, SIMIN_PARAM3_NAME, param,
139                       &offnlinha, orig))
      {
141     Int->Gain=(TBANVAR)atof(param);
      }
143   else
      {
145     Warning(WARN_PARAMETER_MISSING, SCase->SimFile, offnlinha,
            "Block gain not specificed... assuming one\r\n");
147     Int->Gain=1.0;
      }
149   Int->Gain=(TBANVAR)((Int->Gain*SCase->tstep*(TBANVAR)(Int->TickPreset+1))/2.0);

151   if(GetParameterFromCurrentSection(arq, SIMIN_PARAM4_NAME, param,
                          &offnlinha, orig))
153   {
        Int->IniValue=(TBANVAR)atof(param);
155   }
      else
157   {
        Warning(WARN_PARAMETER_MISSING, SCase->SimFile,
159         offnlinha, "Block initial value not specificed... assuming zero\r\n");
        Int->IniValue=0.0;
161   }

163   //stores the block ID inside the structure
      Int->BlockType=BlockTpl->Block[SCase->Block[bnum].BlockTemplateNum].BlockID;
165   Int->BlockSize=BlockTpl->Block[SCase->Block[bnum].BlockTemplateNum].BlockSize;

167   //stores the commands inside runtime structure
      RuntimeInsertInitFunction(Rt, BlockTpl, SCase, bnum);
169
      RuntimeInsertRunFunction(Rt, BlockTpl, SCase, bnum);
171
      RuntimeInsertCommitFunction(Rt, BlockTpl, SCase, bnum);
173   return OK;
    }
175
    void SimIntegratorDeclaration(BlockTemplateData *Template)
177 {
      long bnum;
179   SimIntegratorData aux; //local instance to get internal address offsets

181   bnum=TemplateIncludeBlock(Template, NAME_SIMINTEGRATOR,
                     ID_SIMINTEGRATOR, sizeof(aux), BLOCK_DELAYED);
183
      //Inputs declaration
185   //Digital inputs
      //Analog inputs
187   BlockIncludeInput(&Template->Block[bnum], AN_INPUT,
                SIMIN_AN_INPUT0_NAME, (long)(&aux.Input)-(long)(&aux));
189
```

```
    //Outputs declaration
191 //Digital outputs
    //Analog outputs
193 BlockIncludeOutput(&Template->Block[bnum], AN_OUTPUT,
               SIMIN_AN_OUTPUT0_NAME, (long)(&aux.Output)-(long)(&aux));
195
    //Internal parameters
197 BlockIncludeParam(&Template->Block[bnum], DIG_PARAM,
              SIMIN_PARAM0_NAME, (long)(&aux.CfgFlag)-(long)(&aux));
199 BlockIncludeParam(&Template->Block[bnum], DIG_PARAM,
              SIMIN_PARAM1_NAME, (long)(&aux.TickPreset)-(long)(&aux));
201 BlockIncludeParam(&Template->Block[bnum], DIG_PARAM,
              SIMIN_PARAM2_NAME, (long)(&aux.TickToGo)-(long)(&aux));
203 BlockIncludeParam(&Template->Block[bnum], AN_PARAM,
              SIMIN_PARAM3_NAME, (long)(&aux.Gain)-(long)(&aux));
205 BlockIncludeParam(&Template->Block[bnum], AN_PARAM,
              SIMIN_PARAM4_NAME, (long)(&aux.IniValue)-(long)(&aux));
207
    //Block operational data
209 BlockIncludeInitializer(&Template->Block[bnum]);
    BlockIncludeRuntime(&Template->Block[bnum]);
211 BlockIncludeCommit(&Template->Block[bnum]);

213 //Stores custom block data parser
    BlockIncludeDataParser(&Template->Block[bnum], &SimIntegratorDataParser);
215 }
  #endif
```

APÊNDICE B – Exemplo de arquivo de entrada com o *design*

A seguir é mostrado o sistema descrito nos capítulos anteriores para teste da ferramenta de compilação do arcabouço.

B.1 Fluxo de dados

O fluxo de dados utilizado, consiste na malha mecânica simplificada de um gerador hidroelétrico, com um pequeno controlador proporcional-integral (PI) em uma malha de velocidade, mais uma turbina hidráulica modelada por um simples ganho (KUNDUR, 1994). O sistema não possui nenhuma não linearidade, tais como saturação (limitadores) da variável de comando ou um algoritmo de *anti-windup* para o termo integral do controlador PI. Esse esquema pode ser visto na Fig.30.

Dos elementos assinalados na Fig.30, "WREF" é o valor de referência do controlador PI, "TTURB" é o conjugado de acionamento da turbina, "MEC_INT" é o integrador cuja saída é a velocidade do conjunto mecânico, e "WERR" é o erro de velocidade do controlador.

Os números em retângulos arredondados próximo dos componentes são aqueles que o compilador adotou para cada um dos blocos, conforme a listagem de saída do compilador, citada no Apêndice C.

O sistema parte do repouso, com condições iniciais nulas. Todos os valores de ajustes estão em [p.u.], e podem ser obtidos a seguir, na listagem da metalinguagem. O ajuste feito é puramente lúdico, para que houvessem oscilações na resposta a um degrau unitário em "WREF" em t=0,0 [s].

Todos os elementos foram criados com os blocos mais elementares da biblioteca, utilizando aritmética de ponto flutuante de dupla precisão e integração numérica trapezoidal, com passo de 100,0 [μs]. O tempo de simulação/execução do algoritmo foi feito igual a 200,0 [s].

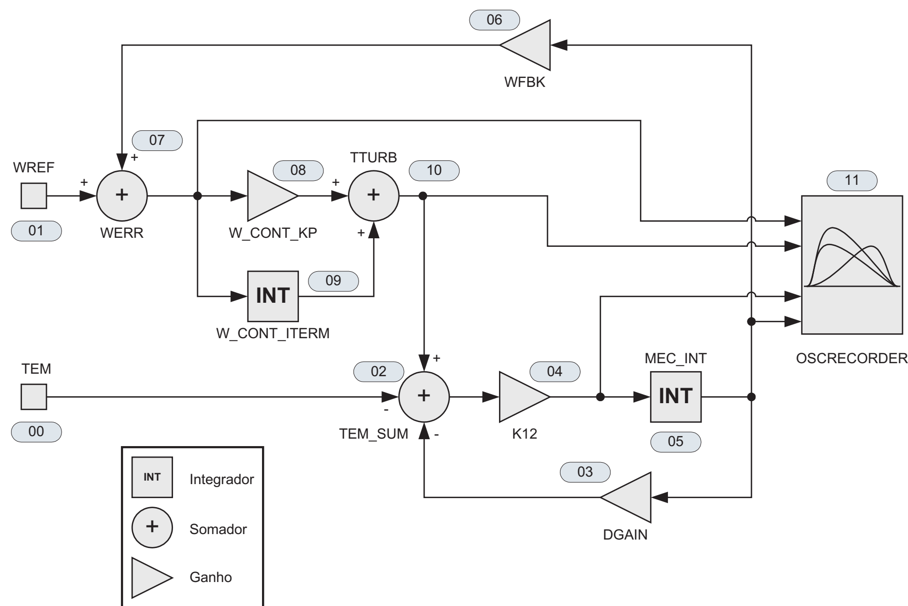

Figura 30: Diagrama de blocos do sistema utilizado nos testes do compilador e interpretador do arcabouço.

B.2 Listagem resultante na metalinguagem

```
[SIMULATION]
NAME=MAQSINCMEC_DT
TSTEP=0.0001
TFINAL=500.0

[ANALOG_CONSTANT]
NAME=TEM
VALUE=0.0

[ANALOG_CONSTANT]
NAME=WREF
VALUE=1.0

[SIMPLE_SUM3]
NAME=TEM_SUM
INPUT_1 = TTURB.Output
GAIN_1  = 1

INPUT_2 = TEM.VALUE
GAIN_2  = -1

INPUT_3 = DGAIN.Output
GAIN_3  = -1

[GAIN]
NAME=DGAIN
INPUT=MEC_INT.INTEGRATOR_OUTPUT
GAIN=0.05 ;COEF D

[GAIN]
NAME=K12
```

```
INPUT=TEM_SUM.OutpuT
GAIN=0.07 ;COEF K12

[SIMPLE_INTEGRATOR]
NAME=MEC_INT
INTEGRATOR_INPUT=K12.Output
INITIAL_VALUE=0.0
TICK_PRESET=0
TICK_TO_GO=0
INTEGRATOR_GAIN=1.0

[GAIN]
NAME=WFBK
INPUT=MEC_INT.INTEGRATOR_OUTPUT
GAIN=-1 ;W Feedback

[SIMPLE_SUM]
NAME=WERR
INPUT_1 = WREF.Value
INPUT_2 = WFBK.OutPut

[GAIN]
NAME=W_CONT_KP
INPUT=WERR.OUTPUT
GAIN=1 ;P GAIN

[SIMPLE_INTEGRATOR]
NAME=W_CONT_ITERM
INTEGRATOR_INPUT=WERR.OUTPUT
INITIAL_VALUE=0.0
TICK_PRESET=1000
TICK_TO_GO=0
INTEGRATOR_GAIN=1.0

[SIMPLE_SUM]
NAME=TTURB
INPUT_1 = W_CONT_KP.OUTPUT
INPUT_2 = W_CONT_ITERM.INTEGRATOR_OUTPUT

[SIMPLE_OSC]
Name=OSCRECORDER
INPUT0=TTURB.OUTPUT
INPUT1=TEM.VALUE
INPUT2=MEC_INT.INTEGRATOR_OUTPUT
INPUT3=WERR.OUTPUT
FILE_TYPE=0; MATLAB_OUTPUT
SAMPLING_PERIOD=0.01
REGISTER_PERIOD=200.0
```

B.3 Listagem equivalente em linguagem ANSI C

O código abaixo é o equivalente ao fluxo de dados da malha de blocos mostrada na Fig.30.

```
    double wref=1.0;      //referencia
```

```
    double lastpi=0.0,  //ultimas amostras da entrada dos integradores
3          lastmec=0.0;
    double piold=0.0,   //valores dos integradores
5          mecold=0.0,
           pinew,           //novos integrandos
7          mecnew;

9   // Esse código é executado em loop, e o valor de 'mecold'
    //é impresso a cada 100 iterações
11
    //calcula integrandos
13  pinew   = wref-mecold;
    mecnew  = (((wref-mecold)*(1.0)+piold)-0.05*mecold)*0.07;
15
    //integracao
17  mecold += (lastmec+mecnew)*(0.0001/2); //integracao de mec_int
    piold  += (lastpi+pinew)*(0.0001/2);   //integracao de w_cont_iterm
19
    //atualiza buffer de amostras
21  lastpi  = pinew;
    lastmec = mecnew;
```

APÊNDICE C – Exemplo de arquivo de saída do compilador

A seguir é mostrada a saída com detalhes da atividade do compilador do “TB”, na atividade de processamento do arquivo mostrado no Apêndice B.

```
TBCompiler - TransBlock compiler (1.0.0)
(c) 2007, by Pellini/Clovis
*********************************************************************
Preparing block API...
Testing block API...
Parsing MAQSINC_MEC.CFG...
*********************************************************************
Phase I - Reading blocks and allocating memory...
Phase I - OK
*********************************************************************
Phase II - Reading block data and routing inputs and outputs...
        ANALOG_CONSTANT parser running for simulator block #0
        ANALOG_CONSTANT parser running for simulator block #1
        SIMPLE_SUM3 parser running for simulator block #2
        GAIN parser running for simulator block #3
        GAIN parser running for simulator block #4
        SIMPLE_INTEGRATOR parser running for simulator block #5
        GAIN parser running for simulator block #6
        SIMPLE_SUM parser running for simulator block #7
        GAIN parser running for simulator block #8
        SIMPLE_INTEGRATOR parser running for simulator block #9
        SIMPLE_SUM parser running for simulator block #10
        SIMPLE_OSC parser running for simulator block #11
Phase II System Report - Execution order not adjusted
Simulation case: MAQSINCMEC_DT
Simulation file: MAQSINC_MEC.CFG
Blocks         : 12
    00 - ANALOG_CONSTANT/TEM
    01 - ANALOG_CONSTANT/WREF
    02 - SIMPLE_SUM3/TEM_SUM
    03 - GAIN/DGAIN
    04 - GAIN/K12
    05 - SIMPLE_INTEGRATOR/MEC_INT
    06 - GAIN/WFBK
    07 - SIMPLE_SUM/WERR
    08 - GAIN/W_CONT_KP
    09 - SIMPLE_INTEGRATOR/W_CONT_ITERM
    10 - SIMPLE_SUM/TTURB
    11 - SIMPLE_OSC/OSCRECORDER
```

```

Relationship matrix:
    00 01 02 03 04 05 06 07 08 09 10 11
 00 00 00 00 00 00 00 00 00 00 00 00 00
 01 00 00 00 00 00 00 00 00 00 00 00 00
 02 01 00 00 01 00 00 00 00 00 00 01 00
 03 00 00 00 00 00 01 00 00 00 00 00 00
 04 00 00 01 00 00 00 00 00 00 00 00 00
 05 00 00 00 00 01 00 00 00 00 00 00 00
 06 00 00 00 00 00 01 00 00 00 00 00 00
 07 00 01 00 00 00 00 01 00 00 00 00 00
 08 00 00 00 00 00 00 00 01 00 00 00 00
 09 00 00 00 00 00 00 00 01 00 00 00 00
 10 00 00 00 00 00 00 00 00 01 01 00 00
 11 01 00 00 00 00 01 00 01 00 00 01 00

Block general execution order:
                 00 01 02 03 04 05 06 07 08 09 10 11
Init order     : 00 01 02 03 04 05 06 07 08 09 10 11
Run order      : 02 03 04 05 06 07 08 09 10 11
Commit order   : 05 09 11
Shutdown order : 11
Phase II - OK
**********************************************************************

Phase III - Adjusting block execution order...
    Sorting execution list...
    Checking for algebraic loops...
Phase III Execution list ended - iteraction [0]
Simulation case: MAQSINCMEC_DT
Simulation file: MAQSINC_MEC.CFG
Blocks         : 12
    00 - ANALOG_CONSTANT/TEM
    01 - ANALOG_CONSTANT/WREF
    02 - SIMPLE_SUM3/TEM_SUM
    03 - GAIN/DGAIN
    04 - GAIN/K12
    05 - SIMPLE_INTEGRATOR/MEC_INT
    06 - GAIN/WFBK
    07 - SIMPLE_SUM/WERR
    08 - GAIN/W_CONT_KP
    09 - SIMPLE_INTEGRATOR/W_CONT_ITERM
    10 - SIMPLE_SUM/TTURB
    11 - SIMPLE_OSC/OSCRECORDER

Relationship matrix:
    00 01 02 03 04 05 06 07 08 09 10 11
 00 00 00 00 00 00 00 00 00 00 00 00 00
 01 00 00 00 00 00 00 00 00 00 00 00 00
 02 01 00 00 01 00 00 00 00 00 00 01 00
 03 00 00 00 00 00 00 00 00 00 00 00 00
 04 00 00 01 00 00 00 00 00 00 00 00 00
 05 00 00 00 00 01 00 00 00 00 00 00 00
 06 00 00 00 00 00 00 00 00 00 00 00 00
 07 00 01 00 00 00 00 01 00 00 00 00 00
 08 00 00 00 00 00 00 00 01 00 00 00 00
 09 00 00 00 00 00 00 00 01 00 00 00 00
 10 00 00 00 00 00 00 00 00 01 00 00 00
 11 01 00 00 00 00 00 00 00 00 00 01 00
```

```
Block general execution order:
                 06 03 01 00 07 08 10 02 04 05 09 11
Init order     : 06 03 01 00 07 08 10 02 04 05 09 11
Run order      : 06 03 07 08 10 02 04 05 09 11
Commit order   : 05 09 11
Shutdown order : 11
Phase III - OK
**********************************************************************
```

APÊNDICE D – Arquivo de configuração para a aplicação do relé

A seguir é mostrado o arquivo de entrada de configuração do arcabouço "TB", representando a lógica de blocos funcionais, presente no relé utilizado como prova de conceito nesse trabalho.

D.1 Listagem resultante na metalinguagem

```
[SIMULATION]
NAME=RELAY_A
TSTEP=0.00104166666
TFINAL=200.0

# Nao modificar a ordem ou os nomes dos blocos de interface
# GOOSE_IN, GOOSE_SV_UDTS_INPUTS, INT_EXTERN_INPUTS, GOOSE_OUT e
# INT_EXTERN_OUTPUTS
##########################################################################
[GOOSE_IN]
NAME=GOOSE_IN
APPID_HEX= 0x0100
CONF_REV = 1
CTRL_BLK_REF_NAME   = IED_NON_ECXISTE_System/LLN0$GO$gcb01
REM_DEVICE_OFF_TIME = 4.0
N_INPUTS = 2
    NAME_INPUT_0 = P68_IED_BELOW
INIVALUE_INPUT_0 = 0
DATAITEM_INPUT_0 = 1

    NAME_INPUT_1 = DJ52A_IED_BELOW
INIVALUE_INPUT_1 = 0
DATAITEM_INPUT_1 = 0

[GOOSE_SV_UDTS_INPUTS]
NAME       = GOOSESV_IN
APPID_HEX  = 0x4000
LDNAME_HEX = 0xED00
SMPRATE    = 16
CONFREV    = 1

[INT_EXTERN_INPUTS]
NAME = IED_IHM_IN
```

```
N_INPUTS = 3
NAME_INPUT_0 = CLOSE_DJ
NAME_INPUT_1 = OPEN_DJ
NAME_INPUT_2 = RESET_P86

[GOOSE_OUT]
Name                = Goose_Out
APPID_HEX           = 0x0123
CTRL_BLK_REF_NAME   = IED_A_PELLINI_System/LLN0$GO$gcb01
DATASET_REF_NAME    = IED_A_PELLINI_System/LLN0$Goose1
GOOSE_ID_NAME       = IED_A_PELLINI01
CONF_REV            = 1
MAC_DST_LSHORT_HEX = 0xAAAA
MAINTENANCE_TIME    = 1.0
N_OUTPUTS           = 4
Source_Output_0 = DJ52.OUTPUT
Source_Output_1 = P68.OUTPUT
Source_Output_2 = TIMER_TO_TRIP.OUTPUT
Source_Output_3 = LOCKOUT86.OUTPUT
Name_Output_0   = DJState
Name_Output_1   = P68
Name_Output_2   = TRIP_GLOBAL
Name_Output_3   = P86

[INT_EXTERN_OUTPUTS]
Name = LEDS_OUT
N_OUTPUTS = 6
Name_Output_0   = DJ_CLOSED_52A
Source_Output_0 = DJ52.OUTPUT

Source_Output_1 = P68.OUTPUT
Name_Output_1   = P68_Sent
Latch_Output_1  = YES

Source_Output_2 = TIMER_TO_TRIP.OUTPUT
Name_Output_2   = TRIP_GLOCAL
Latch_Output_2  = YES

Source_Output_3 = LOCKOUT86.OUTPUT
Name_Output_3   = P86_Lockout

Source_Output_4 = GOOSE_IN.VALUE_0
Name_Output_4   = P68_Received
Latch_Output_4  = YES

Source_Output_5 = TIMER_TO_BF.OUTPUT
Name_Output_5   = BREAKER_FAIL
Latch_Output_5  = YES

Source_Output_6 = IA_MOD.OUTPUT_MOD
Name_Output_5   = IA(RMS)
Latch_Output_5  = NO

# Os demais blocos a seguir podem ser criados e modificados livremente
##############################################################################
[DIGITAL_CONSTANT]
Name  = TRUE
VALUE = 1
```

```
[DIGITAL_CONSTANT]
Name  = FALSE
VALUE = 0

[DIGITAL_FILTER]
NAME        = FOUR_REAL_IA
TICK_PRESET = 0
TICK_TO_GO  = 0
INPUT  = GOOSESV_IN.PRPHACURRENT
NCOEFS  = 16
COEF_0  = 2896
COEF_1  = 2676
COEF_2  = 2048
COEF_3  = 1108
COEF_4  = 0
COEF_5  = -1108
COEF_6  = -2048
COEF_7  = -2676
COEF_8  = -2896
COEF_9  = -2676
COEF_10 = -2048
COEF_11 = -1108
COEF_12 = 0
COEF_13 = 1108
COEF_14 = 2048
COEF_15 = 2676

[DIGITAL_FILTER]
NAME        = FOUR_IMAG_IA
TICK_PRESET = 0
TICK_TO_GO  = 0
INPUT   = GOOSESV_IN.PRPHACURRENT
NCOEFS  = 16
COEF_0  = 0
COEF_1  = 1108
COEF_2  = 2048
COEF_3  = 2676
COEF_4  = 2896
COEF_5  = 2676
COEF_6  = 2048
COEF_7  = 1108
COEF_8  = 0
COEF_9  = -1108
COEF_10 = -2048
COEF_11 = -2676
COEF_12 = -2896
COEF_13 = -2676
COEF_14 = -2048
COEF_15 = -1108

[MOD_PHAS]
NAME = IA_MOD
REAL = FOUR_REAL_IA.OUTPUT
IMAG = FOUR_IMAG_IA.OUTPUT

[DIGITAL_CONSTANT]
Name  = IA_THRESHOLD
VALUE = 5000

```

```
    [COMPARE]
153 NAME    = TEST_IA
    INPUT_1 = IA_MOD.OUTPUT_MOD
155 INPUT_2 = IA_THRESHOLD.VALUE
    TYPE    = >
157
    [TIMER]
159 NAME   = TIMER_TO_TRIP
    INPUT  = TEST_IA.OUTPUT
161 ENABLE = TRUE.VALUE
    TIME_TO_PICK = 0.050
163
    [LOGIC]
165 NAME       = NOT_P68
    LOGIC_TYPE = NOT
167 N_INPUTs   = 1
    INPUT_0    = GOOSE_IN.VALUE_0
169
    [LOGIC]
171 NAME       = TRIPLOG
    LOGIC_TYPE = AND
173 N_INPUTS   = 2
    INPUT_0    = NOT_P68.OUTPUT
175 INPUT_1    = TIMER_TO_TRIP.OUTPUT

177 [TIMER]
    NAME         = TIMER_TO_BF
179 INPUT        = TRIPLOG.OUTPUT
    ENABLE       = TRUE.VALUE
181 TIME_TO_PICK = 0.100

183 [LOGIC]
    NAME       = P68
185 LOGIC_TYPE = AND
    N_INPUTS   = 2
187 INPUT_0    = TEST_IA.OUTPUT
    INPUT_1    = TIMER_TO_BF.NOT_OUTPUT
189
    [LOGIC]
191 NAME       = OPEN52A
    LOGIC_TYPE = OR
193 N_INPUTS   = 2
    INPUT_0    = TRIPDJ.OUTPUT
195 INPUT_1    = IED_IHM_IN.VALUE_1

197 [LOGIC]
    NAME       = CLOSE52A
199 LOGIC_TYPE = AND
    N_INPUTS   = 2
201 INPUT_0    = CLOSEPULSE.OUTPUT
    INPUT_1    = LOCKOUT86.NOT_OUTPUT
203
    [ONESHOT]
205 NAME         = CLOSEPULSE
    INPUT        = IED_IHM_IN.VALUE_0
207 ENABLE       = TRUE.VALUE
    TIME_TO_HOLD = 0.010
209 INPUT_PULSE_POLARITY = ACTIVE_HIGH
```

```
211 [ONESHOT]
    NAME   = TRIPDJ
213 INPUT  = TIMER_TO_TRIP.OUTPUT
    ENABLE = TRUE.VALUE
215 TIME_TO_HOLD = 0.010
    INPUT_PULSE_POLARITY = ACTIVE_HIGH
217
    [FLIPFLOP]
219 NAME    = LOCKOUT86
    INPUT_J = FALSE.VALUE
221 INPUT_K = IED_IHM_IN.VALUE_2
    INPUT_S = TRIPDJ.OUTPUT
223 INPUT_R = FALSE.VALUE
    ENABLE  = TRUE.VALUE
225
    [FLIPFLOP]
227 NAME    = DJ52
    INPUT_J = CLOSE52A.OUTPUT
229 INPUT_K = FALSE.VALUE
    INPUT_S = FALSE.VALUE
231 INPUT_R = OPEN52A.OUTPUT
    INITIAL_STATE = 1
233 ENABLE  = TRUE.VALUE

235 [SIMPLE_OSC]
    Name = OSCRECORDER
237 N_CHANNELS = 12
    SAMPLING_PERIOD = 0.001041
239 REGISTER_PERIOD = 5.0
    INPUT0 = GOOSESV_IN.PRPHACURRENT
241 INPUT_TYPE0 = 1
    INPUT1 = FOUR_REAL_IA.OUTPUT
243 INPUT_TYPE1 = 1
    INPUT2 = FOUR_IMAG_IA.OUTPUT
245 INPUT_TYPE2 = 1
    INPUT3 = IA_MOD.OUTPUT_MOD
247 INPUT_TYPE3 = 1
    INPUT4 = IA_MOD.OUTPUT_PHASE
249 INPUT_TYPE4 = 1
    INPUT5 = TEST_IA.OUTPUT
251 INPUT_TYPE5 = 1
    INPUT6 = TIMER_TO_TRIP.OUTPUT
253 INPUT_TYPE6 = 1
    INPUT7 = TIMER_TO_BF.OUTPUT
255 INPUT_TYPE7 = 1
    INPUT8 = P68.OUTPUT
257 INPUT_TYPE8 = 1
    INPUT9 = TRIPDJ.OUTPUT
259 INPUT_TYPE9 = 1
    INPUT10 = DJ52.OUTPUT
261 INPUT_TYPE10 = 1
    INPUT11 = LOCKOUT86.OUTPUT
263 INPUT_TYPE11 = 1

265 FILE_TYPE=0; MATLAB_OUTPUT
```

Referências

4DIAC. *4DIAC and FORTE homepage*. 2010. Disponível em: <http://www.fordiac.org>.

ABNT$_E$X. 2010. Disponível em: <http://abntex.codigolivre.org.br/>.

ADKINS, B. *The General Theory of Electrical Machines*. London, United Kingdom: Chapman and Hall Ltd., 1964.

ALMEIDA, E. S. de et al. *C.R.U.I.S.E - Component Reuse in Software Engineering*. Recife, Brasil: RiSE - Reuse in Software Engineering Group, C.E.S.A.R - Centro de Estudos e Sistemas Avançados do Recife, 2007. Free book, under Creative Commons License. Disponível em: <http://cruise.cesar.org.br/index.html>.

AMD. *AMD - Advanced Micro Devices Homepage*. 2006. Disponível em: <http://www.amd.com>.

ANALOG. *Analog Devices homepage*. 2010. Disponível em: <www.analog.com>.

ANDRADE, J. G. B. M. de. *Uma contribuição ao estudo da estabilidade de tensão em sistemas elétricos de potência: novos aspectos relacionados à representação da carga*. Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo, Departamento de Eng. de Energia e Automação Elétricas, 2007.

ANDRAKA, R. A survey of cordic algorithms for fpga based computers. In: *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1998. p. 191–200. ISBN 0-89791-978-5.

APOSTOLOPOULOS, C. A.; KORRES, G. N. Real-time implementation of digital relay models using matlab/simulink and rtds. In: *European Transactions on Electrical Power*. Greece: John Wiley and Sons, 2008. Vol. 20, N. 3, p. 290–305.

AQX. *AQX Instrumentação homepage*. 2010. Disponível em: <http://www.aqx.com.br/>.

ASSIS, W. de O.; REZEK Ângelo J. J.; SILVA, L. E. B. da. Projeto e implementação de controle digital para acionamento de uma máquina cc através de chopper. *Seminário Brasileiro sobre Qualidade de Energia Elétrica - I SBQEE*, Uberlândia, MG, Brasil, 1997.

ASTIC, J.; BIHAIN, A.; JEROSOLIMSKI, M. The mixed adams-bdf variable step size algorithm to simulate transient and long term phenomena in power systems. *Power Systems, IEEE Transactions on*, v. 9, n. 2, p. 929 –935, may 1994.

ATP/EMTP. *Alternative Transient Program, Electromagnetic Transient Program homepage*. 2010. Disponível em: <http://www.emtp.org/about.html>.

BALL, S. R. *Embedded Microprocessor Systems: Real World Design*. Newton, MA, USA: Butterworth-Heinemann, 1996. ISBN 0750697911.

BALL, S. R. *Analog Interfacing to Embedded Microprocessors: Real World Design*. Newton, MA, USA: Butterworth-Heinemann, 2001. ISBN 0750673397.

BEEDLE, M. et al. *Manifesto for Agile Software Development*. 2001. The AgileManifesto Homepage. Disponível em: <http://agilemanifesto.org/>.

BHATTACHARYA, A. et al. Hardware software partitioning problem in embedded system design using particle swarm optimization algorithm. In: *CISIS '08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*. Washington, DC, USA: IEEE Computer Society, 2008. p. 171–176. ISBN 978-0-7695-3109-0.

BOEHM, B. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 11, n. 4, p. 14–24, 1986. ISSN 0163-5948.

BROOKS, F. P. *The mythical man-month : essays on software engineering*. 2. ed. Addison-Wesley Pub. Co., 1995. Paperback. Disponível em: <http://www.worldcat.org/isbn/0201835959>.

CATSOULIS, J. *Designing Embedded Hardware*. USA: O'Reilly Media, Inc., 2005. ISBN 0596007558.

CHEN, H. et al. Dynamic simulation of electric machines on fpga boards. In: *Electric Machines and Drives Conference, 2009. IEMDC '09. IEEE International*. [S.l.: s.n.], 2009. p. 1523 –1528.

COSTA, P. et al. Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, USA, v. 14, n. 2, June 2007.

DAVARE, A. et al. A next-generation design framework for platform-based design. Conference on Using Hardware Design and Verification Languages - DCCon, San Jose, California, USA, 2007.

DEMENKO, A. Movement simulation in finite element analysis of electric machine dynamics. *Magnetics, IEEE Transactions on*, v. 32, n. 3, p. 1553 –1556, may 1996. ISSN 0018-9464.

DOUGLASS, B. P. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0201325799.

DSPACE. *DSpace homepage*. 2010. Disponível em: <http://www.dspaceinc.com>.

DUFOUR, C.; BELANGER, J. Discrete time compensation of switching events for accurate real-time simulation of power systems. In: *Industrial Electronics Society, 2001. IECON '01. The 27th Annual Conference of the IEEE*. USA: [s.n.], 2001. v. 2, p. 1533 –1538 vol.2.

DYNASIM. *Dynasim AB homepage*. 2010. Disponível em: <http://www.dynasim.se>.

ECOS. *ECOS homepage*. 2010. Disponível em: <http://ecos.sourceware.org/>.

ECSS. *ECSS - The European Cooperation for Space Standardization, ECSS Standards*. 2010. Disponível em: <http://www.ecss.nl>.

EMBEDDEDRELATED. *Embedded Related Homepage*. 2010. Disponível em: <http://www.embeddedrelated.com>.

EMERSON, M.; NEEMA, S.; SZTIPANOVITS, J. *Handbook of Real-Time and Embedded Systems*. CRC Press, 2006. ISBN: 1584886781. Disponível em: <http://chess.eecs.berkeley.edu/pubs/283.html>.

FAA. *FAA - Federal Aviation Administration*. 2010. Disponível em: <http://www.faa.gov>.

FAUSTINO, M. R. *Norma IEC61131-3: aspectos históricos, técnicos e um exemplo de aplicação*. Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo, Departamento de Engenharia de Energia e Automação Elétricas, 2005.

FIELDBUS. *The FieldBus Foundation Homepage*. 2010. Disponível em: <http://www.fieldbus.org/>.

FREERTOS. *FREERTOS homepage*. 2010. Disponível em: <http://www.freertos.com>.

FRöHLICH, A. A. M. *Application-Oriented Operating Systems*. Tese (Doutorado) — GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Germany, 2001.

GAJSKI, D. D. *Silicon Compilation*. University of California, USA: Addison-Wesley Pub., 1988. Hardcover.

GAJSKI, D. D. et al. *High-level synthesis: introduction to chip and system design*. Norwell, MA, USA: Kluwer Academic Publishers, 1992. ISBN 0-7923-9194-2.

GANSSLE, J. *The Art of Designing Embedded Systems, Second Edition*. Newton, MA, USA: Newnes, 1999. ISBN 0750686448, 9780750686440.

GREGA, W. Hardware-in-the-loop simulation and its application in control education. *29th ASEE/IEEE Frontiers in Education Conference, IEEE*, Session 12b6, p. 7–12, November 1999.

GUO, J.; EDWARDS, S.; BOROJEVIC, D. Designing reusable, reconfigurable control software for power electronics systems. 2008. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.125.418>.

HARTMANN, S. *The World as a Process: Simulations in the Natural and Social Sciences*. 2005. Disponível em: <http://philsci-archive.pitt.edu/archive/00002412/>.

HüBNER, J. F. *Modelo de tese para a Poli/USP*. 2009. Disponível em: <mailto://jomi at inf.furb.br>.

HENZINGER, T. A.; SIFAKIS, J. The discipline of embedded systems design. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 40, n. 10, p. 32–40, 2007. ISSN 0018-9162.

HESS, F. M.; ABBOTT, I. *Comedi - The Linux Control and Measurement Device Interface homepage*. 2006. Disponível em: <http://www.comedi.org>.

ICSTRIPLEX. *ICS Triplex ISaGRAF homepage*. 2010. Disponível em: <http://www.isagraf.com/index.htm>.

IEC60044. *IEC Standard 60044 - Instrument Transformers*. 2003. IEC - International Electrotechnical Commission. Disponível em: <http://www.iec.ch/>.

IEC61131. *IEC Standard 61131 - Programmable controllers*. 2003. IEC - International Electrotechnical Commission. Disponível em: <http://www.iec.ch/>.

IEC61499. *IEC Standard 61499 - Function blocks*. 2005. IEC - International Electrotechnical Commission. Disponível em: <http://www.iec.ch/>.

IEC61850. *IEC Standard 61850 - Communication networks and systems in substations*. 2003. IEC - International Electrotechnical Commission. Disponível em: <http://www.iec.ch/>.

IEEE1076. *IEEE Standard 1076 - VHDL Language Reference Manual*. 2008. IEEE - Institute of Electrical and Electronical Engineering -. Disponível em: <http://ieeexplore.ieee.org/>.

IEEE1364. *IEEE Standard 1364 - Description Language Based on the Verilog Hardware Description Language*. 2005. IEEE - Institute of Electrical and Electronical Engineering -. Disponível em: <http://ieeexplore.ieee.org/>.

IEEE1588. *IEEE Standard 1588 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. 2008. IEEE - Institute of Electrical and Electronical Engineering - Technical Committee on Sensor Technology (TC-9). Disponível em: <http://ieeexplore.ieee.org/>.

INRIA. *SCILAB/SCICOS - The Free Platform for Numerical Computation. INRIA - Institut national de recherche en informatique et automatique*. 2010. Disponível em: <http://www.scilab.org>.

INTEL. *Intel Homepage*. 2006. Disponível em: <http://www.intel.com>.

IREQ. *HyperSim - Real-Time Digital Simulator homepage. IREQ - Institut de recherche d'Hydro-Quebec*. 2010. Disponível em: <http://www.ireq.ca>.

ISO11898. *ISO Standard 11898 - Controller area network (CAN)*. 2003. ISO - International Organization for Standardization. Disponível em: <http://www.iso.org/>.

JAVA. *Sun/Oracle Develper Network homepage*. 2010. Disponível em: <http://java.sun.com/>.

JOSHI, A. *Embedded Systems: Technologies and Markets*. BCC Research. Wellesley, MA, USA, April 2009. Disponível em: <http://www.bccresearch.com/report/IFT016C.html>.

KERNIGHAN, B. W.; RITCHIE, D. M. Book. *The C programming language / Brian W. Kernighan, Dennis M. Ritchie*. [S.l.]: Prentice-Hall, Englewood Cliffs, N.J. :, 1978. x, 228 p. ; p. ISBN 0131101633.

KUNDUR, P. *Power system stability and control*. New York, USA: EPRI Editors, McGraw-Hill Professional, 1994.

LAPLANTE, P. A. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. USA: Wiley-IEEE Press, 1996. ISBN 0780334000.

LI, F. *A Software Framework for Advanced Power System Analysis: Case Studies in Networks, Distributed Generation, and Distributed Computation*. Tese (Doutorado) — Faculty of the Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, 2001.

LóPEZ-VALLEJO, M.; LóPEZ, J. C. On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 8, n. 3, p. 269–297, 2003. ISSN 1084-4309.

LYNX. *Lynx Tecnologia*. 2010. Disponível em: <http://www.lynxtec.com.br>.

MACNEIL, T. *Don't be Misled by MIPS*. 2004. Disponível em http://www.ibmsystemsmag.com/mainframe/enewsletterexclusive/9806p1.aspx.

MATHWORKS. *MathWorks MatLab homepage*. 2010. Disponível em: <http://www.mathworks.com>.

MICHELI, G. D.; ERNST, R.; WOLF, W. (Ed.). *Readings in hardware/software co-design*. Norwell, MA, USA: Kluwer Academic Publishers, 2002. ISBN 1-55860-702-1.

MICROSOFT. *Microsoft Corp. homepage*. 2010. Disponível em: <http://www.microsoft.com/>.

MODELICA. *Modelica Association homepage*. 2010. Disponível em: <http://www.modelica.org>.

MOHAN, N.; UNDELAND, T.; ROBBINS, W. *Power Electronics - Converters, Applications and Design*. USA: John Wiley and Sons, 1995.

MUSSA, S. A. *Controle de um conversos CA-CC trifásico PWM de três níveis com fator de potência unitario utilizando DSP*. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2003.

NATIONAL. *National Instruments Inc. homepage*. 2010. Disponível em: <http://www.ni.com>.

NETO, A. et al. Marte: a multi-platform real-time framework. *IEEE TRANSACTIONS ON NUCLEAR SCIENCE*, JET - Joint European Torus, EFDA - European Fusion Development Agreement, Culham Science Centre, Oxfordshire, UK, v. 57, n. 2, April 2010.

NOERGAARD, T. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. USA: Newnes, 2005. ISBN 0750677929.

OCERA. *OCERA - Open Components for Embedded Real-time Applications homepage*. 2010. Disponível em: <http://www.ocera.org>.

OGATA, K. *Engenharia de Controle Moderno*. São Paulo, Brasil: Prentice-Hall, 1998.

OLIVEIRA, M. A. d. et al. Micro-dvr - uma plataforma de desenvolvimento para dvr e facds. *Anais da Conferência Brasileira sobre Qualidade da Energia Elétrica*, Santos, SP, Brasil, p. 7p, 2007.

OPAL. *Opal-RT Technologies homepage*. 2010. Disponível em: <http://www.opal-rt.com>.

OPENMODELICA. *OpenModelica - The Open Source Modelica Consortium homepage*. 2010. Disponível em: <http://www.ida.liu.se/labs/pelab/modelica/OpenSourceModelicaConsortium.html>.

OSCI. *The Open SystemC Initiative homepage*. 2010. Disponível em: <http://www.systemc.org>.

PALLA, S.; SRIVASTAVA, A.; SCHULZ, N. Hardware in the loop test for relay model validation. In: *Electric Ship Technologies Symposium, 2007. ESTS '07. IEEE*. Virgina, USA: [s.n.], 2007. p. 449 –454.

PARKER-HANNIFIN. *CTC Parker Hannifin MachineLogic Developer Software*. 2010. Disponível em: <http://www.ctcusa.com/ctcnet/MLspec.asp>.

PC104CONS. *PC/104 Consortium homepage*. 2010. Disponível em: <www.pc104.org>.

PELLINI, E. L. *Simulador em tempo real de hidrogeradores*. Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo, Departamento de Engenharia de Energia e Automação Elétricas, 2005.

PETZOLD, L. R. *A Description of DASSL: A Differential/Algebraic System Solver*. Sandia National Laboratories. Canada, 1982. Presented at IMACS World Congress, Montreal, Canada, 1982.

PICIOROAGA, F. *Scalable and efficient middleware for real-time embedded systems. A uniform open service oriented, microkernel based architecture*. Tese (Doutorado) — Ecole Doctorale Sciences Pour l'Ingénieur, Université Louis Pasteur - Institut national des sciences appliquées - ENGEES - URS, Strasbourg, France, 2004.

QNX. *QNX Neutrino homepage*. 2010. Disponível em: <http://www.qnx.com>.

RAJASHEKARA, K. et al. Real time simulation of a synchronous machine in a programmable high speed controller. *Conference Record of the IEEE Industry Applications Society Annual Meeting*, v. 1, p. 254–262, 7–12, Oct. 1990.

RIEHLE, D. *Framework Design - A Role Modeling Approach*. Tese (Doutorado) — Swiss Federal Institute of Technology of Zurich, Zurich, Switzerland, 2000.

RIPOLL, I. et al. *WP1 - RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis by OCERA - Open Components for Embedded Real-time Applications*. Universidad Politécnica de Valencia, Spain and others, 2002.

ROITMAN, M.; SOLLERO, R.; OLIVEIRA, J. Real time digital simulation: trends on technology and t&d applications. In: *Power Systems Conference and Exposition, 2004. IEEE PES*. USA: [s.n.], 2004. p. 1767 – 1769 vol.3.

ROYCE, W. Managing the development of large software systems. In: *Proc. IEEE Wescon*. USA: [s.n.], 1970. p. 1–9.

RTAI. *RTAI homepage. Politecnico di Milano. Dipartimento di Ingegneria Aerospaziale*. 2006. Disponível em: <http://www.rtai.org>.

RTCA. *RTCA DO-178B - Radio Technical Commission for Aeronautics, Inc. - Document DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. 2010. Disponível em: <http://www.rtca.org>.

RTDS. *RTDS, Real Time Digital Simulator homepage. RTDS Technologies Inc.* 2010. Disponível em: <http://www.rtds.com>.

RTEMS. *Real Time Executive for Multiprocessor Systems homepage*. 2010. Disponível em: <http://www.rtems.org/>.

RTLINUXFREE. *RealTime Linux homepage. Wind River*. 2010. Disponível em: <http://www.rtlinuxfree.com>.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 18, n. 6, p. 23–33, 2001. ISSN 0740-7475.

SANTOS, E. Z. A. d. *Simulador em tempo real para teste de reguladores de velocidade de turbinas hidráulicas*. Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo, Departamento de Engenharia de Energia e Automação Elétricas, 2006.

SANTOS, R. C. dos. *Algoritmo Baseado em Redes Neurais Artificiais para a Proteção de Distância de Linhas de Transmissão*. Tese (Doutorado) — Universidade de São Paulo, USP, Brasil, 2004.

SEL. *SEL - Schweitzer Engineering Laboratories, Inc. homepage*. 2010. Disponível em: <http://www.selinc.com>.

SENGER, E. C. et al. Desenvolvimento de relé digital para proteção diferencial de linhas de transmissão. In: *Trabalhos técnicos do Seminário Técnico de Proteção e Controle STPC 2008*. Belo Horizonte, Brasil: [s.n.], 2008.

SIFAKIS, J. A framework for component-based construction. In: *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on*. [S.l.: s.n.], 2005. p. 293 – 299.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*. USA: John Wiley and Sons, 2008.

STRINGHAM, G. *Hardware/Firmware Interface Design: Best Practices for Improving Embedded Systems Development*. USA: Newnes, Elsevier, 2009. Hardcover.

SURESH, D. C. et al. Profiling tools for hardware/software partitioning of embedded applications. In: *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. New York, NY, USA: ACM, 2003. p. 189–198. ISBN 1-58113-647-1.

TANENBAUM, A. S. *Modern Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

TIOBE Programming Community Index for April 2010, TIOBE Software BV - “The Importance Of Being Earnest”, The Netherlands. 2010. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.

USENET. *USENET Google Groups*. 2010. Disponível em: <http://groups.google.com>.

USENMEZ, S. et al. Real-time hardware-in-the-loop simulation of electrical machine systems using fpgas. In: *Electrical Machines and Systems, 2009. ICEMS 2009. International Conference on*. [S.l.: s.n.], 2009. p. 1 –6.

USNSF. *SBES - Simulation-based Engineering Science. Revolutionizing Engineering Science Through Simulation*. US NSF - United States National Science Foundation. USA, 2006.

VISSIM. *Visual Solutions VISSIM homepage*. 2010. Disponível em: <http://www.vissim.com/>.

VITA. *VMEbus International Trade Association*. 2010. Disponível em: <http://www.vita.com/>.

VXWORKS. *VXWorks homepage. Wind River*. 2010. Disponível em: <http://www.windriver.com/products/vxworks/>.

WATSON, N.; ARRIGALA, J. *Power Systems Electromagnetic Transient Simulation*. London, United Kingdom: IEE The Institution of Electrical Engineers, 2003.

WILSON, D.; RAUCH, T.; PAIGE, J. Prototyping and the software development cycle. *ACM CHI'92 - Association for Computing Machinery, Computer-Human Interaction*, 1992. Disponível em: <http://www.firelily.com/opinions/cycle.html>.

WU, X.; FIGUEROA, H.; MONTI, A. Testing of digital controllers using real-time hardware in the loop simulation. *35th Annual IEEE Power Electronics Specialists Conference*, p. 3622–3627, 2004.

ZANETTA JR., L. C. *Transitórios Eletromagnéticos em Sistemas de Potência*. São Paulo, Brasil: EdUSP, 2003.

ZHANG, C. et al. Hardware-in-the-loop simulation of distance relay using rtds. In: *SCSC: Proceedings of the 2007 summer computer simulation conference*. San Diego, CA, USA: Society for Computer Simulation International, 2007. p. 149–154. ISBN 1-56555-316-0.