ecai

Universidade Federal de Santa Catarina
Departamento de Automação e Sistemas
Curso de Engenharia de Controle e
Automação Industrial

ufsc

Análise e Projeto de Software para Regulador Digital usando Orientação a Objetos e UML

Monografia submetida à Universidade Federal de Santa Catarina
como requisito para a aprovação na disciplina EEL5901
Projeto de Fim de Curso

Daniel Maurício Kamers

Florianópolis, Fevereiro de 1998

Análise e Projeto de Software para Regulador Digital usando Orientação a Objetos e UML

Daniel Maurício Kamers

Esta monografia foi julgada no contexto da disciplina
EEL5901: Projeto de Fim de Curso
e aprovada na sua forma final pelo
Curso de Engenharia de Controle e Automação Industrial

Banca Examinadora:

Eng°. Mauro Pacheco Ferreira
Orientador da Empresa

Prof. Marcelo Ricardo Stemmer
Orientador do Curso

Prof. Augusto Humberto Bruciapaglia
Responsável pela disciplina e Coordenador do Curso

Prof. Alexandre Trofino Neto, Avaliador

Carla Brandão Sartor, Debatedora

Cynthia Beatriz Scheffer, Debatedora

Agradecimentos

Ao Engenheiro Mauro Pacheco Ferreira, pelo apoio, pelo rumo que proporcionou à este trabalho, e à nossa integração à equipe da Reivax.

Ao Professor Marcelo Ricardo Stemmer, pelas "dicas" valiosas.

Ao Engenheiro João Marcos Castro Soares, pela oportunidade na empresa, e pela vasta experiência compartilhada conosco.

Ao Professor Augusto Humberto Bruciapaglia, pelo incentivo ao trabalho e ao empenho pessoal.

Aos colegas do Departamento de Engenharia de Desenvolvimento da Reivax, pela proximidade constante e conhecimentos compartilhados.

À Reivax Automação e Controle, pela oportunidade, e pelo suporte intenso ao crescimento profissional.

Aos professores e amigos do LCMI e da coordenação, sempre prestativos.

À Universidade Federal de Santa Catarina, pelo privilégio de pertencer ao grupo dos que por aqui passam ou passaram.

À minha família. A vocês devo todas as minhas conquistas.

Resumo

Os sistemas de geração de energia elétrica devem ser regulados para garantir a qualidade da energia produzida. Os reguladores desses sistemas têm que ser adaptados à individualidade de cada central de geração. Para isso, precisam ter parâmetros e topologia de sua estrutura de controle facilmente programados, operados e monitorados. Além disso, vários reguladores devem comunicar-se entre si, e com outros sistemas, para realizar estruturas de controle, supervisão e operação remota.

Uma arquitetura básica de software, que atenda essas necessidades e possa evoluir com necessidades futuras, deve ser desenvolvida com métodos adequados.

Esse trabalho apresenta o desenvolvimento, auxiliado por ferramenta CASE (Computer Aided Software Engineering), de um protótipo em C++ daquela arquitetura, através de orientação a objetos usando UML (Unified Modeling Language).

Abstract

The power generation systems must be regulated to guarantee the quality of the produced energy. The regulators for these systems have to be adapted to the individuality of each power generation plant. Thus, they need to have parameters and structure of their control algorithm easily programmed, operated and monitored. Furthermore, several regulators should communicate to each other, and with other systems, to accomplish structures of control, supervision and remote operation.

A basic software architecture that assists those needs and grows with future needs must be developed with appropriate methods.

This work presents the object-oriented development, aided by CASE (Computer Aided Software Engineering) tool, of a C++ prototype of that architecture, using UML (Unified Modeling Language).

Sumário

Introdução

A energia elétrica é produzida, entre outras formas, através do aproveitamento do potencial hidráulico do nosso país. Essa energia deve satisfazer uma demanda variável, e deve ter características de frequência e tensão adequadas. Além disso, o aproveitamento do potencial deve ser maximizado, e o custo de produção minimizado.

O uso de reguladores de velocidade e tensão, e de estabilizador de sistema de potência, tem por objetivo permitir a geração com aquelas características de qualidade. Um dispositivo de controle desses executa algoritmos em geral semelhantes para diferentes centrais de geração, mas que podem exigir algumas adaptações. Eles devem ainda estar coordenados com a operação da usina: processos de partida, parada, sincronismo, rejeição de carga, CAG (controle automático de geração), etc.

Há algumas décadas, esses dispositivos de controle eram implementados com estruturas eletro-mecânicas adaptadas a cada central. Atualmente, reguladores como os da Reivax são baseados em microprocessadores e sistemas de aquisição, e o *software* desses dispositivos é adaptado de acordo com as particularidades da aplicação. Essa implementação por software é predominantemente realizada em "baixo nível", a alterá-la é um processo caro.

Entre as características desejáveis para esses dispositivos de controle, que atendem às necessidades atuais para esse tipo de produto, estão uma *interface homem-máquina de alto nível*, tanto para a programação como para o usuário final (as centrais), e *capacidades de comunicação*, possivelmente via rede ou barramentos padronizados, que possibilitem compor estruturas individuais ou distribuidas/hierárquicas de controle e supervisão.

Essas características deverão ser suportadas por uma arquitetura básica de software comum aos diferentes tipos de reguladores, que seja extensível e reutilizável, entre outros

aspectos. O sucesso desse desenvolvimento depende de uma metodologia apropriada, que favoreça a modularidade, e o desenvolvimento evolutivo.

As metodologias de desenvolvimento propõem diferentes visões do *ciclo de vida* do software, ou *paradigmas de desenvolvimento*. Será feita portanto uma breve descrição dos principais paradigmas e da abordagem adotada nesse trabalho.

O desenvolvimento do protótipo dessa arquitetura básica será auxiliado por uma ferramenta CASE, orientada à Linguagem C++. A análise e projeto serão orientados a objeto, utilizando a notação denominada UML (Unified Modeling Language).

A arquitetura básica de software que se concebeu tem características de *sistema de tempo real*, e deverá permitir a implementação de *processos* auxiliares, como de comunicação e transmissão de dados, registro, etc., além de um processo *principal*, que realizará a execução de algoritmos, e a manipulação de variáveis e parâmetros internos e do sistema.

Sendo um subconjunto do projeto global, o desenvolvimento deste protótipo focalizou especificamente a manipulação dos dados e execução de algoritmos (sejam eles relacionados a controle, a supervisão, ou mesmo eventualmente à comunicação), e em especial, o que será definido como *bloco*. Esses *blocos* são semelhantes aos blocos de sistemas de simulação dinâmica, como o Vissim, ou o SSD da Reivax, e serão o elemento principal da solução que permitirá compor as estruturas de controle e supervisão a que nos referíamos no início desta introdução. Serão portanto a *base* da arquitetura.

O que se propõe aqui é uma avaliação dos riscos envolvidos nessa base de arquitetura, através de testes realizados sobre o protótipo.

Este trabalho tem a seguinte organização:

O capítulo 1 apresenta alguns aspectos sobre o problema de regulação de sistemas de geração hidrelétrica. Permite ter uma visão inicial dos requisitos de programação dos reguladores digitais de velocidade e tensão, e estabilizadores de sistemas de potência, e por

conseguinte, duma arquitetura básica de software (avaliada neste trabalho através de um protótipo) que os satisfaça.

O capítulo 2 introduz os conceitos de desenvolvimento de software, e os diferentes paradigmas, situando nesse contexto e nesse trabalho a orientação a objetos e a "Linguagem Unificada" (UML), justificando seu emprego segundo as necessidades de qualidade no desenvolvimento, para se atingir a qualidade no produto.

O capítulo 3 apresenta um resumo de alguns aspectos da norma IEC1131-3, que orientarão o desenvolvimento da arquitetura básica. Essa norma define uma interface de programação de alto nível para controladores programáveis. O capítulo 3 descreve ainda as características de uma arquitetura de software baseada em blocos, enfocando os aspectos de controle digital e de tempo real.

O capítulo 4 desenvolve a concepção e elaboração do protótipo de software básico dos controladores, em um ciclo de desenvolvimento (*iteração*), conforme o modelo da *Rational*, fabricante da ferramenta CASE *Rational Rose*. Apresenta detalhes de implementação, testes realizados e os resultados obtidos.

O capítulo 5 procura ressaltar os pontos mais importantes do trabalho e definir algumas conclusões.

Apresentação **da** *empresa: REIVAX Automação e Controle*

A REIVAX Automação e Controle é uma empresa de prestação de serviços e desenvolvimento de produtos na área de controle de processos industriais, especializada em sistemas elétricos de potência. A empresa está no mercado desde julho de 1987 como integrante do CELTA (Centro Empresarial para Laboração de Tecnologias Avançadas), localizando-se atualmente no Parque Tecnológico Alfa, em Florianópolis - SC. A empresa atua nas seguintes áreas:

Controle:

- Estabilizadores de sistemas de potência
- Reguladores de velocidade de turbinas hidráulicas
- Sistemas de excitação e regulação de tensão de geradores

Instrumentação:

- Isoladores galvânicos
- Transdutores de velocidade, potência e tensão
- Sistemas de identificação de modelos e aquisição de dados
- Sistema de monitoração e registro de eventos

Software:

- Simulador de sistemas dinâmicos
- Sistema de manipulação e visualização de curvas
- Sistema de aquisição de dados

Serviços:

- Consultoria para controle de processos industriais
- Ensaios de campo para recepção e ajuste de sistemas de controle

Orientada para o mercado de sistemas elétricos de potência, a REIVAX tem hoje como principais clientes empresas como ABB, CHESF, Eletronorte, Eletrosul, Siemens, Cemig (MG), Light, Coemsa e CAMMESA (Compania Administradora del Mercado Mayorista Electrico S. A. – Argentina).

Capítulo 1

Aspectos de Geração de Energia Elétrica

1.1 Introdução

Esse capítulo apresenta alguns aspectos de geração de energia elétrica e do controle desses sistemas para permitir a produção de energia com qualidade, através de reguladores de velocidade e de tensão, e de estabilizador de sistema de potência.

A programação desses controladores deve poder ser realizada em alto nível, o que implica no uso de uma arquitetura de software capaz de mapear os elementos da interface de programação aos elementos de execução da lógica e do controle requeridos pelos controladores.

1.2 Aspectos de Geração Hidrelétrica

As centrais convencionais convertem a energia potencial hidráulica disponível em energia elétrica. Numa hidrelétrica, a turbina converte a energia mecânica potencial da queda d'água em energia cinética transmitida ao eixo do gerador. O gerador disponibiliza potência elétrica em seus terminais. A potência produzida deve suprir uma demanda (ou carga) variável ao longo do tempo. A distribuição da potência elétrica realizada através de linhas de transmissão em corrente alternada deve ter uma frequência constante. Os geradores estão ligados ao sistema elétrico através de um barramento.

O gerador é uma máquina síncrona. A tensão de excitação no gerador, juntamente com o movimento do seu rotor, originam potência elétrica em seus terminais, pelo princípio de conversão eletro-mecânica de energia.

As diversas centrais do país precisam ser coordenadas e equilibradas. Uma central pode ter vários geradores, e cada gerador precisa ter seus *setpoints* ajustados de forma

conveniente, de acordo com a situação do sistema global. Nesse contexto, se aplicam reguladores de velocidade e tensão, e estabilizadores de sistemas de potência.

A realização de um aproveitamento hidrelétrico depende da existência de uma vazão hidráulica e um desnível. Para cada local aproveitável temos diversos esquemas de utilização possíveis. Assim, cada usina tem sua individualidade. Isto reforça a necessidade de flexibilidade e modularidade dos reguladores e dispositivos utilizados na automação de uma central.

1.2.1 Elementos das centrais: turbinas hidráulicas

- Turbina Pelton (quedas de 300 metros ou mais) - princípio da variação do momento.

Controle da potência de entrada realizado através da agulha do *injetor*. Esse sistema permite tempos de 20 a 100 segundos para fechamento com *rejeição de carga* de 100%. Fechamento rápido sem golpe de aríete pode ser realizado através do desvio do jato com defletor, quando a potência de entrada vai a zero.

- Turbina Francis:

O controle de velocidade e potência mecânica fornecida é realizada através das pás das turbinas, pelo anel distribuidor (anel de Fink), ou individualmente. A água entra tangencialmente, e sai radialmente.

- Turbina Kaplan (ou hélice):

Um distribuidor (*porta* acionada por servomotor hidráulico) regula a potência mecânica fornecida para a turbina. As pás têm configuração de hélice, de ângulo de "ataque" variável ou não. Pode existir uma conjungação de movimentos das pás e do distribuidor, para garantir um funcionamento ótimo para todos os níveis de carga.

1.2.2 Operação de Centrais e Regulação

Pode ser dividida em duas etapas principais:

1) execução do paralelo;

2) controle de ativo e reativo (após o paralelo);

Essas etapas devem ser realizadas de acordo com o plano de operação do centro de despacho de carga.

Execução do Paralelo (exemplo - turbinas Francis/Kaplan):

O sistema externo (cargas, sistema de transmissão e outros geradores) pode ser considerado um barramento infinito, com frequência constante. O comando de aumento de referência do regulador de velocidade (que está "preso" ao barramento), reflete num aumento da potência fornecida ao sistema. Com a rejeição de carga surge uma aceleração proveniente da diferença entra a potência de saída e a potência de entrada, que é compensada pela regulação de velocidade. O regulador comanda o fechamento do distribuidor. Esse tipo de regulação é lenta. Para ações mais rápidas, atua-se diretamente na excitação do gerador (regulação de tensão e estabilização da potência).

Potência ativa e reativa:

A potência ativa é a potência de saída efetivamente convertida em trabalho mecânico. A potência reativa é a potência armazenada no circuito (em campos eletro-magnéticos). Muitas vezes se introduzem elementos reativos próximos aos centros de consumo de potência ativa, para reduzir a propagação da energia reativa (que aumenta perdas e reduz a capacidade de transmitir potência ativa) pelas linhas de transmissão.

Para a execução do paralelo, o gerador e a rede deverão apresentar características idênticas: a) forma de onda; b) valor de tensão; c) frequência; d) fase; e) seqüência de fase. A sincronização no que se refere aos itens *b*, *c* e *d*, pode ser feita com auxílio do "fogo girante", ou do sincronoscópio. Os itens *a* e *b* são normalmente assegurados pelos aspectos construtivos [Fuchs, 83]. A sincronia pode ser feita automaticamente, com um acoplador. Ele verifica as condições de tensão e dispara um disjuntor quando a condição é

suficientemente estável. Um sincronizador ajusta a freqüência do gerador. Um repartidor pode dividir a potência nominal entre as diversas máquinas da central.

No controle de ativo e reativo, o regulador de tensão atua na excitação do gerador, controlando basicamente a potência reativa, e afetando muito pouco a potência ativa. Neste controle algumas considerações são levadas em conta: temperatura do rotor e estator do gerador abaixo dos limites; potência do eixo menor ou igual à potência hidráulica da turbina; ângulo do rotor *versus* campo de reação de armadura dentro do limite de estabilidade estática / dinâmica. Ou seja, além de manter a tensão da máquina em valores aceitáveis para o sistema, mantê-lo dentro de sua curva de capacidade.

1.2.3 Aspectos de Regulação: Resumo

O regulador de velocidade é necessário, especialmente nas grandes centrais, para gerar energia elétrica com características técnicas adequadas, ou seja, com uma frequência padrão. Além disso, permite controlar a potência gerada.

O regulador de velocidade controla a potência mecânica desenvolvida pela turbina, atuando sobre a vazão de água através do distribuidor. O distribuidor é usualmente acionado por servomotor hidráulico. O servomotor por sua vez é acionado por uma válvula distribuidora. Essa válvula pode ser controlada pelo regulador através de uma válvula proporcional.

O regulador deve levar o gerador ao ponto de operação, segundo uma referência de velocidade, e aí mantê-lo. A referência de velocidade depende da carga demandada. Estando a máquina sincronizada e "presa" ao barramento, o controle da vazão de água permite o fornecimento de potência para o sistema.

O regulador de tensão deve controlar o fluxo de potência reativa no sistema, atuando na excitação do gerador. A influência na potência ativa através da regulação da excitação é muito pequena.

Já o controle da potência ativa através da regulação da potência mecânica de entrada provoca reação oposta da potência reativa. O regulador de tensão controla a potência reativa para, além de manter a tensão da máquina adequada para o sistema, mantê-la dentro de sua curva de capacidade.

O sinal de atuação do estabilizador de potência é adicional ao sinal do regulador de tensão. Ambos utilizam as medidas de tensão e corrente (quantas e quais, depende do algoritmo. O PWX, estabilizador da Reivax, utiliza o *algoritmo da potência acelerante*), e atuam na excitação da máquina. O estabilizador de potência permite regular a potência ativa do sistema de uma forma mais rápida que a regulação normal, através do controle da velocidade da máquina.

O estabilizador de potência melhora a resposta do sistema, produzindo maior amortecimento das oscilações.

1.2.4 Programação dos Controladores

As características dinâmicas do sistema originam um conjunto de requisitos para a realização do controle (por exemplo, frequência da ação de controle, ou taxa de amostragem, em sistemas de controle digital).

Além da regulação da planta, sua operação exige o controle de diversos mecanismos de sequenciamento, intertravamento, e segurança. Existem sequências pré-definidas de partida, operação e parada dos grupos geradores, e um conjunto de condições precisa ser constantemente verificado. Podemos dizer que o sistema é caracterizado por um conjunto de **estados** de operação (anexo I).

A definição das leis de controle e de lógicas de comando e seqüência é usualmente descrita em termos de diagramas de blocos de função e diagramas *ladder*.

Adicionalmente, uma representação dos estados de operação do sistema, pode facilitar a programação dos controladores.

A norma IEC1131-3 define uma interface de programação que contempla esses 3 elementos, blocos de função, diagramas escada, e diagramas de estado.

1.3 Conclusões

As descrições em alto nível das leis de controle e da lógica dos controladores aplicados ao sistema de geração têm que ser traduzidos para algoritmos de software que serão executados por esses controladores.

Este trabalho desenvolve um protótipo de software para a execução dos algoritmos de controle e lógica, baseado em *blocos*. Esses blocos deverão na medida do possível realizar um mapeamento simples entre os elementos de programação da linguagem de alto nível (blocos, contatos, estados...) e os elementos de execução de algoritmo propriamente dito (o código executado), de forma semelhante a um sistema de simulação.

Capítulo 2

Desenvolvimento de Software

2.1 Introdução

O desempenho do hardware tem aumentado significativamente a cada ano, a custos constantes. Por outro lado, os custos com a manutenção e gastos com softwares não-conformes ou não entregues têm sido notáveis - a chamada "crise do software".

A meta última da Engenharia de Software é incrementar a produtividade e a qualidade global do produto, sendo a redução de custos um efeito colateral posterior, e desejável.

Esses fatores levam à busca da aplicação das técnicas formais de análise e projeto de software.

Este capítulo descreve os diferentes paradigmas de desenvolvimento e a metodologia que se adotou para a realização do protótipo de software básico para os reguladores de velocidade e tensão, e o estabilizador de sistema de potência da Reivax.

2.2 O Mercado de Software

As diferenças entre os vários tipos de softwares são relevantes a ponto de implicar em metodologias e ferramentas de desenvolvimento específicas. Em particular, pode-se definir três famílias de produtos de software que englobam a maior parte da produção nos dias atuais [Bonfatti et ali, 97]:

- software de negócio
- software de tempo-real
- software pessoal (ou para computadores pessoais)

Falaremos a seguir das duas primeiras categorias.

2.2.1 Software de Negócio

Software de negócio (ou sequencial) é essencialmente desenvolvido para manipular dados e realizar transações com eles. O elemento-chave é a grande quantidade de informação para processar, enquanto a maioria dos algoritmos são simples. São usualmente construídos apoiados sobre um sistema de gerenciamento de banco de dados.

Este tipo de software pode ser dividido em procedimentos definidos e pouco inter-relacionados. O projeto de software pode ser desmembrado em subprojetos delegados a equipes diferentes.

Softwares de negócio correspondem à maior parte em volume de código, de mercado e de aplicações, e por isso existe um grande esforço de engenharia de software no desenvolvimento deste tipo de produto, surgindo várias metodologias e ferramentas, que vão das linguagens de quarta geração à ambientes CASE.

2.2.2 Software de Tempo Real

Os produtos de software definidos de tempo-real dão suporte às aplicações de automação, e também são muito utilizados. Consistem basicamente dos softwares de CLPs (controlador lógico programável), e softwares *embarcados*, que compõem máquinas e dispositivos em geral. Define-se software como de tempo-real quando sua efetividade é função não só da *correção* do código (código correto) mas também do tempo de execução. Em outras palavras, software de tempo-real precisa detectar eventos e reagir em um intervalo de tempo pré-definido.

Este tipo de software cobre uma faixa ampla de aplicações em automação, controlando de sistemas anti-blocante (ABS) em automóveis a estruturas grandes e complexas como uma planta de fornecimento de energia.

Muito esforço de desenvolvimento tem sido realizado, em particular, sobre os chamados "*hard* real-time systems" (com restrições severas de tempo), e os "safety critical

systems" (em que uma falha implica em danos à pessoas ou bens materiais). Este esforço busca garantir a robustez e qualidade necessárias do software de controle produzido.

Porém, o software para controladores por muitos anos foi desenvolvido através de linguagens de *baixo nível*, e pouco investimento se fazia para buscar melhores ferramentas e metodologias, pois o usuário final exigia basicamente um custo global baixo. Nos últimos anos, o aumento da eficiência e confiabilidade de uma plataforma PC, a custos constantes, e da variedade de ferramentas de desenvolvimento, tem possibilitado seu emprego em aplicações antes restritas a *mainframes* ou (no âmbito deste trabalho) plataformas de tempo-real dedicadas. Assim, os avanços em Engenharia de Software, e requisitos de interface de alto nível estão se difundindo entre essas aplicações.

2.3 Paradigmas da Engenharia de Software

Paradigmas da engenharia de software são conjuntos de métodos, ferramentas e procedimentos colocados sob forma de etapas. Para se escolher um paradigma deve-se levar em conta a natureza do projeto e da aplicação, os métodos e as ferramentas a serem usados, os controles e os produtos que precisam ser entregues. Existem quatro principais paradigmas da engenharia de software:

2.3.1 O Ciclo de Vida Clássico

O ciclo de vida clássico é ainda um dos paradigmas mais conhecidos na Engenharia de Software. Sua modelagem é do tipo "cascata", como pode ver-se na figura.

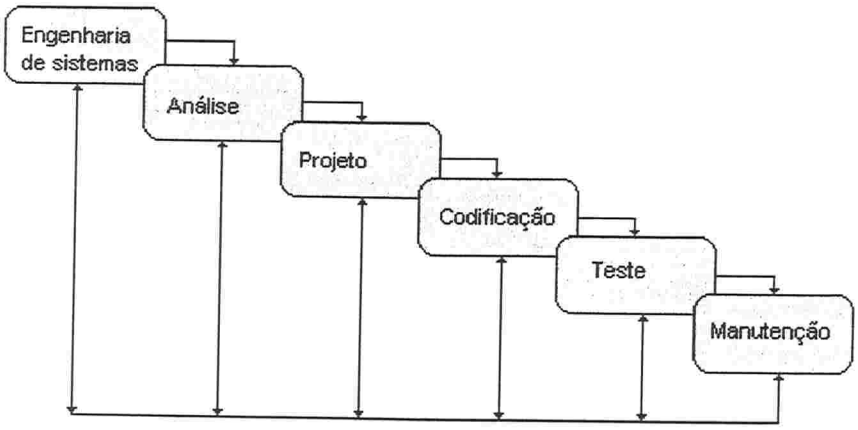


Figura 2.1 - desenvolvimento em "cascata"

Os vários passos do paradigma são descritos a seguir:

- Análise e Engenharia de Sistemas: Como o software faz parte de um conjunto amplo, que é o sistema, é necessário conhecer-se o sistema e através dele coletar os requisitos que devam ser utilizados no software.

- Análise de Requisitos de Software: A análise de requisitos de software é uma coleta mais refinada de informações e requisitos, concentrada especificamente no software. Nela vão as especificações de funcionalidade, desempenho e interface desejados. A especificação de requisitos, tanto de sistema como de software, deve ser bem documentada e revista com os clientes.

- Projeto: O projeto é um processo de quatro atributos distintos: estrutura de dados, arquitetura de software, detalhes procedimentais e caracterização de interface. O projeto é a tradução dos requisitos de forma a se poder avaliar a qualidade que terá o software antes que o mesmo seja codificado.

- Codificação: É a tradução do projeto em uma linguagem que possa ser interpretada pela máquina. Se o projeto é bem feito e o programador tem conhecimento e experiência, o processo de codificação torna-se praticamente mecânico.

- Testes: Após feita a codificação, inicia-se o processo de testes. Através dos testes verifica-se erros e também se o código realmente produz o resultado desejado.

- Manutenção: A manutenção é necessária após a entrega do programa. Nela corrige-se eventuais erros encontrados e adapta-se o software ao ambiente em que ele será instalado. Mudanças funcionais e de desempenho às vezes são exigidas. Na manutenção são reaplicadas as etapas precedentes do ciclo de vida.

As limitações do ciclo de vida clássico são as seguintes:

- Os projetos reais raramente seguem o fluxo seqüencial que o modelo propõe. Alguma iteração sempre ocorre e traz problemas na aplicação do paradigma.
- Muitas vezes é difícil para o cliente declarar todas as exigências explicitamente. O ciclo de vida clássico exige isso e tem dificuldade de acomodar a incerteza natural que existe no começo de muitos projetos.
- O cliente deve ter paciência. Uma versão de trabalho do programa não estará disponível até um ponto tardio do cronograma do projeto. Um erro grosseiro, se não for detectado até que o programa de trabalho seja revisto, pode ser desastroso.
- O custo de desenvolvimento do ciclo de vida clássico inviabiliza sua utilização.

2.3.2 Prototipação

Muitas vezes, o cliente definiu um conjunto de objetivos gerais para o software, mas não identificou requisitos de entrada, processamento e saída detalhados. Em outros casos, o desenvolvedor pode não ter certeza da eficiência de um algoritmo, da adaptabilidade de um sistema operacional ou da forma que a interação homem máquina deve assumir.

Nessas, e em muitas outras situações, uma abordagem de prototipação à engenharia de software pode representar a melhor abordagem. [Pressman, 95]

A prototipação é um dos paradigmas da engenharia de software onde é produzida uma versão (ou protótipo) do software já nas etapas iniciais do projeto. Isto proporciona ao

cliente ou ao programador uma versão para testes. Um protótipo pode assumir uma das formas:

- um protótipo em papel ou modelo baseado em PC que retrata a interação homem-máquina de uma forma que capacita o usuário a entender quanta interação ocorrerá.
- um protótipo de trabalho que implementa algum subconjunto da função exigida do software desejado.
- um programa que executa parte ou toda a função desejada, mas que tem outras características que serão melhoradas em um novo esforço de desenvolvimento.

Uma figura com a seqüência de prototipação é mostrada a seguir.

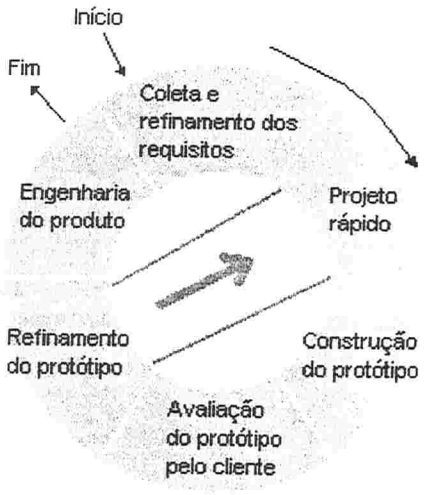


Figura 2.2 - prototipação

2.3.3 O Modelo Espiral

O modelo espiral tem por objetivo fazer com que haja uma constante análise de riscos e uma interação com os clientes. Ele foi proposto com o intuito de abranger as melhores características, tanto do ciclo de vida clássico quanto da prototipação. O modelo possui quatro atividades, que são destacadas a seguir:

- Planejamento: determinação dos objetivos, alternativas e restrições.
- Análise de Riscos: análise de alternativas e identificação / resolução de riscos.
- Engenharia: desenvolvimento do produto no "nível seguinte".
- Avaliação feita pelo Cliente: avaliação dos resultados da engenharia.

Uma figura representando o modelo espiral é mostrada a seguir.

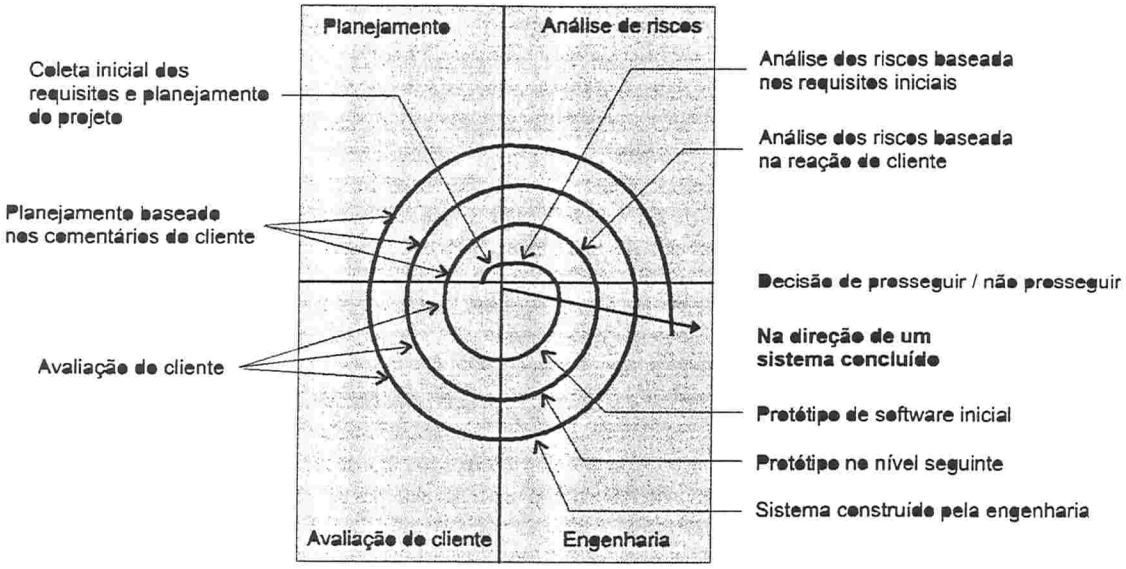


Figura 2.3 - Modelo Espiral

No início, é feita uma coleta de requisitos e planejamento. A análise de riscos é aplicada, baseada nos requisitos iniciais. Aí, avalia-se se é possível prosseguir ou não. Se a decisão de seguir é aceita, faz-se um protótipo do software para a primeira avaliação do cliente. Faz-se um planejamento com base nos comentários do cliente, e uma análise de riscos baseada nas reações dos clientes. Decide-se novamente se é viável a continuação do projeto e caso a resposta seja afirmativa faz-se um novo protótipo do software. A partir daí haverão ciclos (como mostrado na figura) até que haja um refinamento do modelo que resulte num produto final elaborado.

Nota-se que quanto maior for a dimensão radial, mais completo e bem elaborado será o software.

O modelo espiral apresenta alguns inconvenientes. Pode ser difícil convencer grandes clientes que a abordagem evolutiva é controlável. Além disso, o modelo é relativamente novo e não tem sido amplamente utilizado. Outro problema é que se um grande risco não for descoberto, ocorrerão problemas que podem ser de mesma magnitude.

2.3.4 Técnicas de Quarta Geração

Técnicas de quarta geração (4GT) abrangem um conjunto de ferramentas (softwares) que auxiliam a confecção do código do programa na maneira mais fácil possível, aproximando-se da linguagem natural. Sua função é diminuir o tempo de geração do código, através de uma especificação detalhada do software.

Como os outros paradigmas, ele inicia com uma coleta de requisitos. Se o software é pequeno, após a coleta pode ser elaborado diretamente o código do programa, sem passar por uma etapa de projeto detalhado. Mas se forem necessários esforços maiores, é preciso o desenvolvimento de um projeto detalhado, sob o risco do software apresentar problemas de má qualidade, baixa manutenibilidade e má aceitação do cliente.

Uma figura representando as técnicas de quarta geração pode ser vista a seguir.

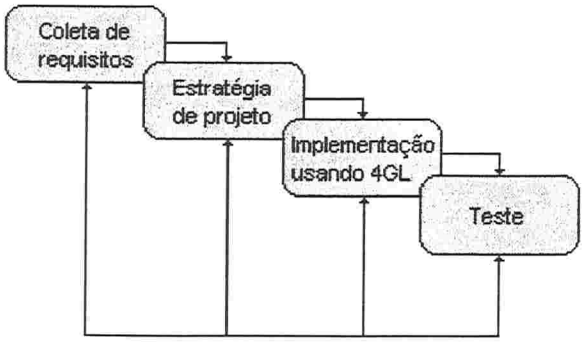


Figura 2.4 - Técnicas de quarta geração

Alguns problemas das técnicas de quarta geração são citadas:

- O uso das 4GT's está atualmente limitado a sistemas de informação comerciais e banco de dados.

- Há uma baixa manutenibilidade para sistemas de grande porte.

2.3.5 Combinando Paradigmas

Muitas vezes são utilizados vários paradigmas no desenvolvimento de software. Por exemplo, o paradigma do modelo espiral combina elementos do ciclo de vida clássico com prototipação. Procura-se combinar as características mais adequadas de cada paradigma para a solução do problema.

2.4 Orientação a Objetos

Um passo importante para a produção de softwares para controladores consiste em desenvolver e reutilizar as unidades mais comuns e suas interações. Sistemas complexos podem ser significativamente diferentes, porém ser compostos das mesmas unidades básicas.

A tecnologia de *orientação a objetos* é particularmente apropriada para dar suporte ao projeto de software modular.

Muitas metodologias de análise e projeto orientadas a objetos têm sido propostas e cada vaz mais utilizadas.

Particularmente, a orientação a objetos tem sido aplicada em sistemas com as seguintes características:

- Sistemas baseados em interfaces gráficas com o usuário;
- Sistemas de tempo real;
- Computação "cliente/servidor";
- Reengenharia - atualização de partes de software mantendo sistemas herdados.

Além da modularidade, a orientação a objetos favorece a representação semântica natural do domínio do problema, e a utilização de uma linguagem comum entre as etapas de análise e projeto.

Entre as notações já desenvolvidas, se destaca a linguagem unificada, ou Unified Modeling Language, proposta por Grady Booch, Ivar Jacobson, e Jim Rumbaugh, baseada em suas experiências pessoais, e incorporando características dos diversos metodologistas

(Shlaer-Mellor, Harel, Meyer, e outros). A UML em sua versão 1.1 foi submetida ao OMG (Object Management Group) em julho de 1997 (no início deste trabalho, portanto).

A tecnologia de orientação a objetos apresenta entre suas vantagens: a linguagem entre usuários, analistas, projetistas e implementadores torna-se única e homogênea; os modelos gerados refletem o mundo de maneira mais aproximada; descrevem de maneira mais precisa os dados e os processos, propõe uma decomposição do problema baseada em um particionamento natural, sendo mais fáceis de entender e manter. Estes aspectos facilitam o reuso da arquitetura e do código; em geral, uma pequena mudança nos requisitos não significa alterações massivas no sistema em desenvolvimento, que é mais estável.

2.4.1 UML - Unified Modeling Language

A UML incorpora aspectos das diferentes metodologias de análise e projeto orientados a objetos, e constitui um padrão sustentado por nomes como Microsoft, Hewlett-Packard, Oracle, Texas Instruments, MCI Systemhouse, IBM, e outros. O método vem amadurecendo desde 95, com a unificação do OMT-2, Booch e outros, e recebendo diversas contribuições desde então. Cabe ressaltar aqui que as palavras *método* e *metodologia*, são usadas em sentido restrito. A UML estabelece sintaxe e semântica, porém não constitui uma metodologia para atacar o problema de desenvolvimento, tal como *cascata*, ou a *espiral de Boeme*.

A UML define um mapeamento da análise ao projeto e à implementação, através de uma notação expressiva e consistente. Isso facilita a comunicação entre as pessoas, ajuda a apontar inconsistências e omissões. A linguagem suporta análise e projeto de pequeno e grande porte. Suporta ainda as "4+1 Visões da Arquitetura": lógica, de desenvolvimento, de processo, física, e casos de uso/cenários.

Uma descrição resumida da UML pode ser encontrada no apêndice A deste trabalho.

2.5 O ciclo da *Rational Software* - Desenvolvimento Iterativo e Incremental

Desenvolvimento Iterativo e Incremental é o processo de construção de sistemas de software feito em pequenos passos, proporcionando:

- redução de riscos devido ao *feedback* prematuro
- maior flexibilidade para acomodar alterações de novos requisitos
- melhoria na qualidade de software

O ciclo de vida do software é dividido em uma seqüência de ciclos de desenvolvimento, onde o resultado de cada ciclo é a geração de um produto de software. Cada ciclo é uma sucessão de fases:

- Concepção: estabelecimento do negócio para o sistema a ser desenvolvido
- Elaboração: análise e domínio do problema, estabelecimento de uma base arquitetural, e avaliação preliminar de riscos
- Construção: desenvolvimento incremental de um produto de software completo e pronto para a transição para a comunidade usuária
- Transição: colocação do software disponível aos usuários

Uma iteração é um ciclo de desenvolvimento que termina na liberação de um subconjunto do produto final, percorrendo todos os aspectos do desenvolvimento de software:

- análise de requisitos
- projeto
- implementação
- teste
- documentacão

Na conclusão que vem a seguir, procuramos indicar o que foi aplicado a este trabalho.

2.6 Conclusões

Podemos fazer uma analogia entre o modelo de desenvolvimento iterativo e incremental e o modelo Espiral. As definições das etapas do ciclo de vida incremental correspondem em grande extensão às etapas daquele paradigma.

A diferença fundamental aqui é o quão *pequeno* é o passo, ou incremento, e quão *rápido* ele é realizado, para apresentar algo que possa ser chamado "sub-produto" de software.

Um passo pequeno e rápido, que possa ser prontamente avaliado pelo cliente, sem prejuízo da documentação realizada, e com um tempo muito curto de codificação, é o que se propõe com a utilização de uma metodologia de análise e projeto orientados a objetos utilizando a Linguagem Unificada (UML) e uma ferramenta CASE adequada, que seja capaz de realizar Geração de Código e Engenharia Reversa, o que é uma caracterísitica das linguagens de quarta geração.

Os riscos do uso dessa metodologia, aplicada a C++ e a uma plataforma do tipo PC, para a produção de software para controladores aplicados a sistemas de geração de energia elétrica, precisam ser avaliados.

Este trabalho compõe um ciclo (uma iteração: análise, projeto, implementação, teste, documentação), *no âmbito das duas primeiras etapas, concepção e elaboração*. Em outras palavras, não se pretende gerar uma *liberação* ("arranhamos" a etapa de *construção*, mas não se atingiu a etapa de *transição*) de um sub-produto completo de software, mas apenas as etapas iniciais de desenvolvimento, segundo aquele modelo.

Pelas características da metodologia, e graças à orientação a objetos, existe a possibilidade de que determinados elementos (classes) sejam reutilizadas em etapas seguintes (ou seja, na construção).

Por outro lado, a confecção de um protótipo e realização de testes, amparados por documentação farta e consistente, em um tempo o mais curto possível, caracteriza um ciclo ou iteração completa (poderíamos dizer uma "cascata" completa) no desenvolvimento.

Capítulo 3

Caracterização do Software a ser Desenvolvido

3.1 Introdução

Nos capítulos anteriores, avaliamos o problema (produção de software para regulador aplicado a sistema de geração) e consideramos a metodologia a adotar.

Neste capítulo, definiremos o que se espera desse software. Ao final do capítulo, acrescentamos uma especificação preliminar dos controladores da empresa.

A interface que deverá estar disponível ao usuário que vai programar os reguladores deverá expressar com proximidade a implementação do controle, favorecendo a simplicidade e confiabilidade, e deverá ser familiar, facilmente utilizável, já bem conhecida, utilizando blocos de funções, diagramas escada, e diagramas de estado, por exemplo.

Por outro lado, a utilização de "blocos" (de software) de controle facilita o projeto da lei de controle, e favorece a modularidade. Os elementos de implementação do algoritmo correspondem aos elementos de interface, e podem ser testados e reutilizados individualmente.

Estes dois aspectos do projeto serão desenvolvidos a seguir.

Na primeira parte descrevemos alguns aspectos da norma IEC1131, e sua proposta de interface. Na segunda parte destacamos alguns aspectos de controle digital e de sistemas de tempo real que devem ser considerados para a implementação de um protótipo baseado em blocos.

3.2 Padronização da Interface de Programação - A Norma IEC1131-3

A série de normas IEC 1131, que versam sobre Controladores Programáveis (CPs), compiladas pelo subcomitê 65B, da IEC (International Eletrotechnical Comission) consiste das seguintes partes:

Parte 1: 1992, Informações Gerais.
Parte 2: 1992, Requisitos e Testes de Equipamento.
Parte 3: 1993, Linguagens de Programação.
Parte 4: *User Guidelines* (sob consideração).
Parte 5: *Messaging service specification* (sob consideração).

A norma IEC1131 parte 3, ou IEC1131-3, trata especificamente da representação gráfica, sua sintaxe e semântica, dos elementos constituintes das linguagens definidas na parte 1 (entre outras definições relativas à arquitetura de sistemas de CPs) da norma. Está definido na parte 1 da norma a seguinte tabela de *funções programáveis*:

Grupo de Funções	Exemplos
Controle lógico - logica - temporização - contadores	AND, OR, NOT, XOR, flip-flops; *On-delay*, *off-delay*, pulso temporizado; Contagem crescente/decrescente de pulsos;
Controle sequencial	SFC (sequential function chart)
Processamento de sinais/dados - funções matemáticas - manipulação de dados - processamento de dados analógicos	Aritmética básica: ADD, SUB, MUL, DIV; Aritmética extendida: SQRT, funções trigonométricas; Comparações: maior que, menor que, igual a; Seleção, organização, formatação, transporte; PID, integração, filtragem;

Funções de interface - entrada/saída - outros sistemas - MMI - impressoras - memória de massa	Módulos de entrada/saída digitais e analógicos; Conversão BCD; Protocolos de comunicação; Display, comandos; Mensagens, relatórios; Registro;
Controle de execução	Execução periódica, orientada a eventos;
Configuração do sistema	Verificação de *status*

Tabela 3.1 - Funções Programáveis

Todo o conjunto de funções pode ser utilizado através de qualquer uma das 5 linguagens definidas (2 textuais e 3 gráficas), à exceção dos Diagramas de Função Sequencial (SFC - Sequential Function Chart), que deve ser complementado por uma das 4 linguagens: Diagrama Ladder e Diagramas de Blocos de Função (gráficas), e/ou Texto Estruturado e Linguagem de Instruções (textuais).

Essas definições se baseiam em um *modelo de software* que contempla dois elementos básicos: *elementos de programa*, ou seja, *programas* e *blocos de função*; e *elementos de configuração*: *configurações*, *recursos*, *tarefas*, *variáveis globais*, *caminhos de acesso*, que dão suporte à instalação dos *programas* nos CPs.

Uma análise detalhada da norma e suas implicações na construção de uma arquitetura de hardware/software em conformidade com a norma IEC1131 está em andamento na empresa. Neste trabalho nos restringimos a considerar os elementos existentes nas três linguagens gráficas definidas, e suas implicações mais evidentes para a realização da arquitetura básica do software.

Podemos então enumerar as 5 linguagens definidas:

- SFC - *Sequential Function Chart* - representação de alto nível dos *elementos de controle de execução* que estruturam uma *unidade de organização de programa*

do CP, ou **programa**[1] do CP, descrito nas outras linguagens. É basicamente um GRAFCET.

- IL - *Instruction List* - seqüência de instruções. Semelhante ao *Assembly*. Em cada linha deve haver apenas uma instrução. Cada instrução deve conter um *operador* com *modificadores* opcionais, e se necessário, um ou mais *operandos*, separados por vírgula. As operações são em geral realizadas em termos de operandos e tem valor de retorno num *acumulador*.
- ST - *Structured Text* - semelhante às linguagens estruturadas de alto nível, tal como PASCAL.
- LD - *Lader Diagram* - diagrama *ladder*, bem conhecido e muito utilizado na indústria, tal como a linguagem de diagramas de blocos. É formado por *degraus*, *contatos* e *solenóides* de diferentes tipos. Podem conter ainda blocos de função. É basicamente uma linguagem de descrição de lógica, sequenciamento e intertravamento.
- FBD - *Function Blocks Diagram* - *blocos* conectados por *linhas de fluxo de sinal*. Os blocos têm suas entradas conectadas às saídas de outros blocos.

Um SFC proporciona uma forma de particionar um *programa* (ver anotação 1) em um conjunto de *passos* e *transições* interconectados por *ligações orientadas*. Associado a cada passo existe um conjunto de *ações*, e a cada transição é associado um conjunto de *transições*.

Se não é realizada nenhuma partição no programa, ele inteiro é considerado uma única ação que executa sob controle da entidade que invocou o programa.

Um SFC está intimamente relacionado com os *elementos de configuração*, descritos na norma.

---

[1] PROGRAM ORGANIZATION UNIT - são *funções*, *blocos de função*, ou *programas*. POUs não devem ser *recursivos*, ou seja, não devem invocar a execução de outro POU do mesmo tipo. Neste capítulo quando nos referimos ao **programa** de um CP, estamos nos referindo a um POU.

As linguagens textuais são apropriadas para desenvolver algoritmos complexos mas a princípio foram desconsideradas. Não há nenhum recurso contemplado nessas linguagens que não tenha seu equivalente em FBD/Ladder.

O par FBD/Ladder, têm sido muito utilizado nas aplicações industriais e faz parte da linguagem natural no domínio do problema considerado neste trabalho. Associada aos SFCs, constituem um conjunto poderoso e completo, que será considerado na etapa de definição de requisitos no desenvolvimento do protótipo inicial descrito nos capítulos seguintes.

No apêndice B se encontram exemplos e descrições dos principais elementos dessas linguagens.

3.3 Princípios de Controle Digital

Sistemas amostrados eram inicialmente lentos devido a problemas teóricos e de implementação física. Nas últimas décadas, vários desses problemas foram superados, e surgiram novas técnicas de projeto de sistemas de controle discreto. Um sistema de controle discreto, utilizando um computador digital como elemento de controle, tem papel fundamental em sistemas tais como robótica, controle de aeronaves e automóveis, sistemas biomédicos, operação de satélites, e *controle de processos*. [Houpis & Lamon 92]

Controle Digital é um campo extremamente interdisciplinar, e a descrição de seus princípios está além dos limites deste trabalho, estando envolvidos tópicos como equações diferenciais e a diferença, álgebra de matrizes, teoria de controle clássico e moderno, análise numérica, sistemas estocásticos discretos, teoria de controle discreto, arquitetura de sistemas computacionais (hardware/software), tecnologia de circuitos digitais integrados, conversores de sinais, estruturas de informação, projeto de algoritmos de controle, processamento digital de sinais, engenharia de software e sistemas de testes. Destacaremos nesta seção alguns aspectos importantes a considerar.

O desempenho atual dos microprocessadores tornou possível produzir reguladores digitais baseados em microprocessadores a um custo realizável.

Particularmente, uma arquitetura baseada em x86 tem entre suas vantagens:

- tecnologia padronizada e dominada;
- custo baixo;
- variedade de fornecedores;
- modularidade;

O emprego de um computador digital para a realização da compensação em um sistema de controle depende da preparação de uma lei de controle discreto (algoritmo), e de um *software* que a implemente. Esta realização por software normalmente permite maior flexibilidade e precisão comparado a controladores baseados em circuitos eletrônicos analógicos.

A análise e projeto de sistemas de controle discreto pode ser realizada totalmente no *plano-z*, referida como *controle digital direto*, ou inteiramente no *plano-s*, referida como *"digitization" (discretização)*, ou *controle digital discreto*. [Houpis 92] A discretização requer o emprego de técnicas de transformação por aproximação, como as transformações de *Padé*, e *Tustin*.

As aproximações têm a desvantagem de não manter necessariamente todas as propriedades requeridas do compensador original, o que deve ser garantido com a definição de determinados *critérios* de correlação entre os planos s e z, de acordo com o sistema modelado e projetado.

Existem vários métodos de aproximação da função de transferência do domínio do plano-s para o plano-z, ou seja, aproximações de $z = e^{sT}$. Um dos mais populares é o *Tustin*, ou trapezoidal. [Houpis 92].

3.3.1 Aplicação dos Princípios de Controle Digital ao Domínio do Problema - Arquitetura Baseada em Blocos

As técnicas de transformação podem ser usadas na discretização de "blocos" (ou funções de transferência), que são implementados e testados individualmente. A lei de controle é desenvolvida no plano-s, e sua implementação será realizada pelo usuário utilizando os blocos "contínuos" disponíveis numa interface de alto nível, que será mapeada para uma implementação discreta através da execução do algoritmo de cada bloco numa determinada seqüência. Blocos dinâmicos típicos são *integradores*, *avanço-atraso*, *reset*, *pólo-zero*, *constantes de tempo*, *filtros biquadráticos*, etc (blocos de 1ª e 2ª ordem). Essa técnica compõe os sistemas de *simulação* de sistemas dinâmicos, que em geral dispõe de diversos métodos de transformação, ou *métodos de integração*. No apêndice C se encontra uma aplicação da transformação retangular a blocos de 1ª e 2ª ordem do modelo utilizado para testes do protótipo de software.

Está fora do escopo deste trabalho o detalhamento do processo de projeto e discretização. Os blocos dinâmicos que serão utilizados no software já têm seus algoritmos implementados e testados. A Reivax dispõe de uma ferramenta poderosa de projeto e simulação, o SMO (sistema de monitoração de oscilações), baseado no SSD (sistema de simulação dinâmica), que disponibiliza vários blocos dinâmicos, dentre outros, que podem ser classificados segundo a tabela 3.1 apresentada na seção anterior.

Os métodos de transformação mapeiam os parâmetros das funções de transferência contínua para os coeficientes das equações a diferença correspondentes. Essa transformação não foi contemplada nesta etapa do projeto, e deverá ser implementada num nível mais alto da arquitetura. Este protótipo de software deverá portanto ser capaz de executar os algoritmos de cada bloco a partir de um conjunto de parâmetros pré-calculados (os coeficientes das equações a diferença, entre outros, dependendo do tipo de bloco).

Um problema associado à implementação da lei de controle por blocos é o da realimentação. Os algoritmos devem ser executados em uma seqüência que garanta o cálculo correto e a propagação dos valores das entradas para as saída. Para isso, são empregadas técnicas de ordenação (*sort*) da execução dos blocos. Tais técnicas não estão contempladas nesta etapa, e se assume que o sistema conhecerá uma lista de blocos já ordenada, a qual deverá ser executada. A norma IEC1131-3 faz algumas considerações a

esse respeito, basicamente definindo que um bloco não deveria utilizar a saída de outro que ainda não "rodou" na iteração atual.

Outra consideração importante na implementação do algoritmo de controle é o tamanho da *palavra*, ou seja, o número de bits utilizado na representação numérica. Além do erro associado à quantização, existe a propagação de erros de arredondamento nas operações aritméticas. Estas não linearidades podem diminuir e eficiência do controle ou mesmo instabilizar o sistema.

Por outro lado, o tamanho da palavra não pode ser aumentado arbitrariamente, não apenas por limitações físicas ou de custo, mas também por perdas na velocidade de processamento.

O compromisso custo-velocidade-precisão é um ponto crítico do projeto, e a tomada de tempos de execução deve ser considerada neste protótipo.

Na implementação dos algoritmos dos blocos, é utilizado, no SMO, um método de integração ajustável por bloco, de acordo com a seguinte equação:

$$y = y_0 + \lambda h \, [\, \gamma . dy/dt + (1-\gamma) . dy_0/dt \,]$$

Esta equação é a implementação de um caso mais geral que as transformações retangular (lambda = 1, gama = 0) e trapezoidal (lambda = 0.5, gama = 0.5).

Estes ajustes no método de integração permitem eliminar ou diminuir os efeitos dos erros e não-linearidades dos modelos.

Este ajuste altera o resultado dos cálculos dos coeficientes das equações a diferença, e portanto não é realizada pelo protótipo de software. Os blocos dinâmicos são implementados utilizando os coeficientes da equação a diferença. Esses coeficientes são calculados previamente, com as técnicas de transformação citadas, ou com o próprio SMO da Reivax, para fins de teste do protótipo.

3.4 Sistemas de Tempo Real - Aplicação à Arquitetura de Blocos

Um sistema de controle é evidentemente de tempo real na medida em que está submetido à restrição do tempo de resposta dentro do intervalo de amostragem.

Um núcleo de tempo real poderá ser utilizado na arquitetura de software.

Os parâmetros mais importantes nesta etapa inicial são a *latência de interrupção*, e o tempo de *chaveamento entre tarefas*, pois serão as características de maior influência nas métricas iniciais de desempenho do protótipo. Aprofundar, porém, a descrição desses sistemas está fora do escopo deste trabalho. Segue uma definição daqueles termos:

- Latência de interrupção: tempo entre a *ocorrência real* da interrupção, e a execução da rotina que efetivamente *trata* a interrupção ocorrida.
- Chaveamento entre tarefas: tempo entre a interrupção de uma tarefa e o início de execução de outra. A interrupção de uma tarefa ocorre por iniciativa da própria tarefa, ou quando por algum motivo lhe falta algum recurso para prosseguir. A interrupção também pode ocorrer se o sistema realiza *time-sharing* (não é o caso, a princípio, considerado no projeto).

Cabe acrescentar que existem muitas técnicas formais, e extensões às técnicas clássicas, que permitem incorporar às atividades de especificação de requisitos, análise, e projeto, características de representação adequadas a sistemas de tempo real.

Nesta etapa do projeto, um núcleo de tempo real comercial (RTKernel, para DOS) tem sido utilizado e integrado ao protótipo para a realização de testes e a avaliação de riscos.

No protótipo inicial, basicamente se utilizam *n* tarefas sinalizadas, cada uma, por um *semáforo* próprio. Os semáforos proporcionam um mecanismo de seqüenciamento e estabelecem prioridades entre tarefas. Essas tarefas têm a responsabilidade de coordenar: 1) a execução de blocos ou conjuntos de blocos que componham o algoritmo de controle dos reguladores, e 2) outras tarefas auxiliares, como por exemplo supervisão, registro ou comunicação. No final do capítulo seguinte esse tema é retomado, fazendo-se algumas considerações acerca da implementação do protótipo.

Convém notar que a UML possui extensões para sistemas de tempo-real. Estas extensões, porém, são muito recentes, e não integram a ferramenta CASE utilizada. A

simplicidade deste protótipo minimiza o "prejuízo" ao projeto, em não se utilizar uma notação formal.

3.5 Requisitos

Este capítulo apresentou nas duas seções precedentes uma visão geral do que se pretende desenvolver numa primeira iteração. A diversidade de temas abordados sugeriu um conjunto de pontos críticos a serem considerados, ao mesmo tempo em que permitiu definir o âmbito inicial do projeto (ressaltamos aqui que o escopo deste trabalho é um subconjunto do projeto dos controladores da Reivax).

Abaixo segue a definição dos requisitos de hardware e software do sistema, descritos em linguagem natural. Esta descrição se refere ao projeto global dos controladores e por isso inclui os aspectos de hardware. Dessa descrição, em conjunto com os aspectos já considerados nas seções anteriores, e do conhecimento do domínio do problema, serão desenvolvidos os *casos de uso* do capítulo seguinte.

Especificações para a *Nova CPU*[2]

Custo: Estimamos como custo máximo de montagem (componentes, placa, serviços) o valor unitário X.

Aspectos Funcionais:

A estrutura de controle (função de transferência e lógica) são definidas pelo usuário com interface amigável (ex. SSD 2.0)

O sistema deve ser tolerante a falhas. Para tanto deverá ter facilidades para trabalhar com mais de uma CPU em esquema redundante com troca a quente, sem distúrbios para o sistema controlado, assim que seja detectada uma falha. Essa detecção é feita por um sistema de auto-diagnóstico com funcionamento independente do restante da CPU, podendo parte deste sistema ficar fora da placa, se necessário, pois o diagnóstico inclui também componentes externos à CPU. Deve ser à prova de travamentos, com reinicialização automática na condição mais próxima possível do ponto de operação

[2] Por CPU, refere-se à estrutura de hardware e software que será utilizada em comum para produção dos controladores diversos: regulador de velocidade, regulador de tensão, estailizador de sistema de potência.

anterior.

Será feita troca de parâmetros com o sistema em operação. É aceitável que a troca de estrutura seja feita com esquema diferente, off-line ou em retaguarda.

A CPU terá capacidade de comunicação com outras *irmãs* e com outros sistemas. Por exemplo, deverá ser acessada por nível hierárquico superior ou por sensores inteligentes com um padrão da indústria como o Fieldbus e poder utilizar um PC para troca de estruturas e/ou programas.

Haverá um conjunto completo de testes portáveis para a manutenção e produção em série que facilitarão a identificação de problemas, reduzindo custos.

Alguns aspectos de melhoria de precisão da transdução de sinais, como tensão terminal (melhor que 0,05%) e freqüência (melhor que 0,01%) deverão ser especialmente considerados no projeto.

Deverá ser possível colocar todos controladores de um gerador (reguladores de tensão e velocidade, e estabilizador de sistema de potência) em uma única CPU.

Aspectos de Software

Será basicamente programado em alto nível - C/C++ - sendo que o Assembly não deverá ultrapassar 10% do esforço total de programação.

Para facilidade de desenvolvimento e manutenção, sugere-se o uso de kernel de tempo real de algum padrão industrial.

A documentação deverá ser completa, sugerindo-se o uso de ferramenta CASE para melhorar a qualidade do desenvolvimento.

Deverá ter previsão para futuro acréscimo de algoritmos avançados de controle, como fuzzy, self-tunning, e outros.

A característica de registrador de eventos, com trigger, armazenamento e transmissão deve estar contemplada.

Aspectos de Hardware

Todas as funções de um canal de controle deverão ser feitas por apenas uma CPU, sem necessidade de configurações e jumpers por aspectos funcionais, apenas por opções de hardware.

Interface com outros sistemas para programação (serial para PC) e comunicação com outras CPUs irmãs para a composição de uma estrutura mais complexa.

Expansibilidade de I/O analógico e digital. Mínimo de ( ) saídas analógicas, ( ) entradas analógicas, e ( ) entradas e saídas digitais configuradas e bufferizadas.

Prever SMD e multilayer.

Sugere-se o uso de emuladores/simuladores para facilitar desenvolvimento.

Ter imunidade a EMI de acordo com as normas (...) .

Sugere-se sobredimensionamento de memória para aplicações futuras.

Previsão de pontos de teste e medição para instrumentação (como por exemplo analisador lógico).

Adotar padrão industrial para dimensões de placa e conector (Euro, IMS). Caso haja necessidade de barramento externo tornar compatível com algum padrão (STD, VME).

Deve existir display para mostrar variáveis internas, mensagens, informação de status atual da CPU e do sistema controlado.

Obs.: requisitos iniciais, revisados e complementados com o andamento do projeto.

3.6 Conclusões

Podemos enumerar as principais considerações deste capítulo:

A interface amigável que se pretende utilizar, em particular mantendo conformidade com a norma IEC1131-3, e ainda segundo o costume do usuário deste tipo de sistema (o programador da lógica e dos algoritmos dos reguladores) implica a necessidade de uma representação homogênea e flexível do que se entende por *bloco*, e sua correspondência com uma *função*, *algoritmo*, ou *função de transferência*. Esse *bloco* poderá eventualmente oferecer serviços de configuração e interface do controlador. Ainda, o uso de blocos lógicos e "analógicos" de forma amigável em diagramas de blocos de função e de escada, e a estrutura de controle e sequência baseada em diagramas de estado implicam em suportar, em uma estrutura modular básica, o processamento e manipulação de vários tipos de dados (eventualmente diferentes tamanhos de palavra), e a capacidade de gerenciar uma quantidade razoável deles (prevendo a realização de registro de sinais no tempo), e o compartilhamento entre diagramas "concorrentes".

As facilidades de um núcleo de tempo real deverão estar integradas ao protótipo, e resultados desejáveis são métricas de capacidade de processamento e tempo de resposta.

O protótipo deve atender às restrições dinâmicas de controle digital. Deve trabalhar com múltiplos passos de execução. Existirá uma correspondência direta entre os elementos de interface de programação e os blocos dinâmicos e lógicos que, executados em sequência, realizarão o algoritmo de controle e lógica dos reguladores de tensão e velocidade, e do estabilizador do sistema de potência, eventualmente integrados em um único *programa*.

Toda a carga de pré-processamento de um modelo, inclusive a relacionada à troca de parâmetros e estrutura prevista, deve ser realizada *off-line*. Para isso, tarefas em *background* poderão utilizar os tempos *ociosos* do sistema. Outras tarefas auxiliares vao rodar em *background*, relacionadas à supervisão, registro e comunicação.

Capítulo 4

Análise, Projeto, Implementação e Testes do Protótipo de Software dos Controladores

4.1 Introdução

Segundo o modelo de desenvolvimento proposto, e utilizando UML, realizamos a *concepção* do sistema, especificando seus requisitos iniciais, e estabelecendo um *negócio* (objetivo). Em seguida, realizamos a análise baseada nos casos de uso (ver Apêndice A) e Cenários.

O capítulo anterior vai fornecer a base de requisitos para o desenvolvimento descrito neste capítulo.

A seguir, será realizado o desenvolvimento segundo o modelo *iterativo e incremental*, em um ciclo envolvendo a Análise, Projeto, Implementação e Testes do protótipo de software para os controladores.

Durante a implementação, faremos algumas observações sobre os detalhes de integração do protótipo com um núcleo de tempo real.

4.2 Análise

A realização de casos de uso e cenários não é muito diferente de uma abordagem estruturada em que se procure detalhar requisitos, por exemplo, através de DFDs. É porém uma aproximação natural ao problema, utilizando a descrição em linguagem natural do seu domínio, objetivando a caracterização inicial das classes-chave do projeto.

Por outro lado, a linguagem natural é ambígua e nem sempre é trivial encontrar classes e suas características apenas analisando cenários. Técnicas como CRC (cartão classe-responsabilidade) podem ser utilizadas, especialmente se a equipe não está suficientemente madura em técnicas de desenvolvimento orientadas a objetos.

4.2.1 Descrição do *Negócio*

Este sistema constituirá um protótipo de núcleo de execução dos algoritmos de controle que deverá rodar nos controladores digitais microprocessados da Reivax.

O protótipo permite medir o desempenho de execução dos algoritmos de controle, e dos mecanismos internos do programa.

O protótipo implementa e executa um modelo simplificado composto por blocos dos tipos *ganho, somadores, constantes, blocos dinâmicos de 1ª e 2ª ordem.*

O modelo mantém independência entre os dados calculados e os algoritmos dos blocos.
Os blocos avaliam os valores de suas entradas, e definem os valores de suas saídas. Esses valores devem conter a história do fluxo de sinal daquela conexão.

Um conjunto de blocos e sinais deve compor um diagrama. Diferentes diagramas devem ser construídos para representar os diferentes estados em que o modelo pode estar, e os diagramas devem poder ser coordenados segundo as definições da linguagem SFC, ou seja,

sequencialmente, ou paralelamente (simultaneamente).

Cada diagrama poderá executar um passo (ou período de amostragem) diferente.
O modelo é carregado no núcleo de execução através de um arquivo ASCII conforme pontos definidos pelo usuário.

O modelo inicia e termina a execução dos blocos por comando do usuário.

O modelo gera arquivos de dados com as informações processadas sob comando do usuário ou de evento programado pelo usuário.

4.2.2 Diagramas de Caso de Uso

Diagrama Principal

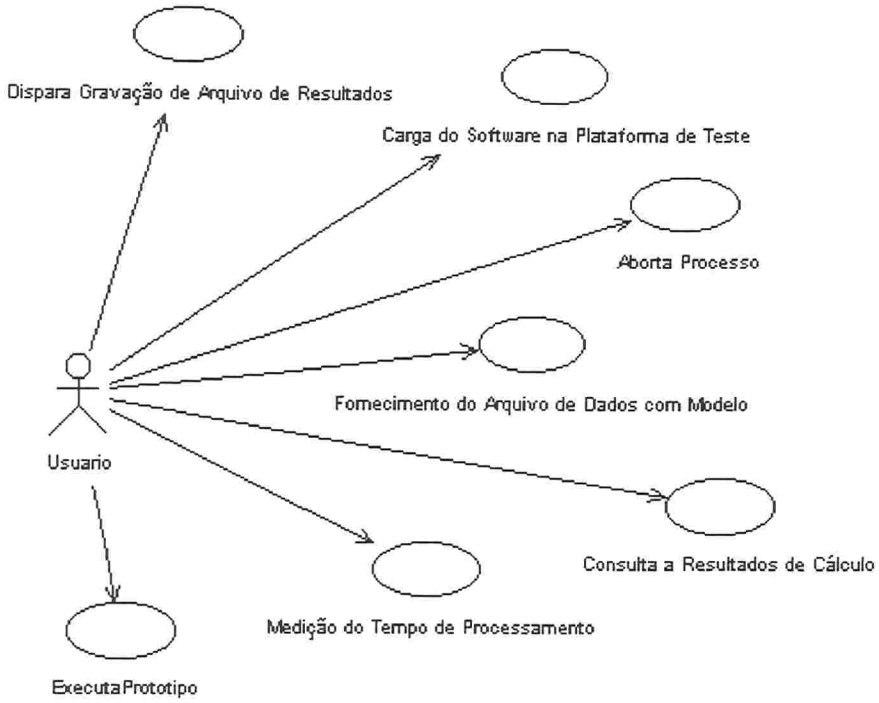
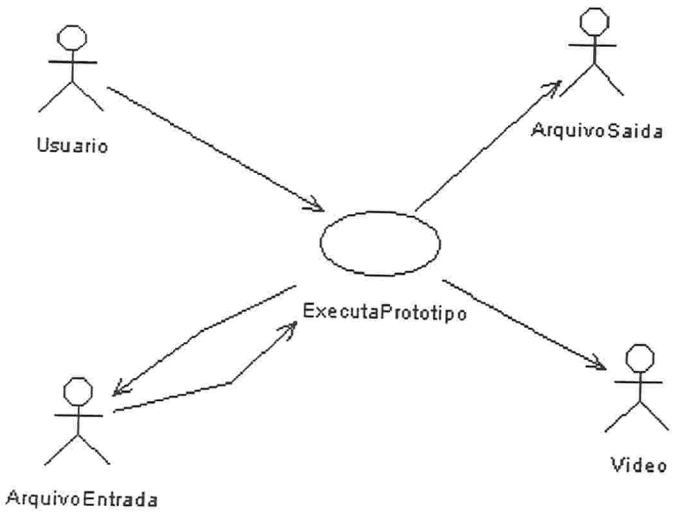


Diagrama Complementar



O diagrama principal define o negócio que será desenvolvido. O diagrama complementar é uma visão alternativa reduzida, já que este trabalho foi desenvolvido basicamente a partir do caso de uso *Executar Protótipo*. Os demais casos de uso, ou acabaram "convertidos" em atores (interface do problema) ou foram tratados como considerações de implementação.

Atores

- Usuário: é o próprio usuário do protótipo;
- Arquivo de Saída: memória de massa (possivelmente o winchester do micro);
- Arquivo de Entrada: memória de massa (idem);
- Video: vai apresentar resultados em "tempo real" necessários;

Descrições Sumárias (casos de uso):

- executar protótipo: é um caso de uso restrito definido a partir do negócio a ser projetado, e vai implicar na existência de um único cenário, descrito a seguir.

4.2.3 Cenário

Fluxo de eventos: Cenário Principal

O usuário comanda o carregamento do protótipo do núcleo ao sistema operacional. O protótipo busca as informações de gerenciamento de execução do modelo, definidos pelo usuário. O protótipo busca primeiro as informações gerais sobre o modelo, e em seguida as informações específicas sobre os diagramas do modelo.

O protótipo interpreta as informações de modelo, e realiza a montagem das listas de execução. O protótipo procura a informação de cada lista, e comanda sua construção.

O protótipo recebe então as informações das conexões entre os diferentes diagramas, e comanda a construção dessas ligações.

O protótipo então busca as informações de saída do sistema. O protótipo recebe uma lista de quais informações deverão estar disponíveis ao final da simulação, e quais deverão ser registradas para análise posterior.

O protótipo configura o sistema de sequenciamento das listas de execução, e comanda a execução de cada lista de acordo com a configuração do sistema.

As informações correntes de processamento são acompanhadas pelo usuário.

Cada lista executa seus blocos, ou seja, seus algoritmos, ou função de transferência.

Sob comando do usuário ou de um evento programado, o protótipo encerra a execução das listas, e dispara o registro das informações processadas.

Complementação do Cenário

Fluxo de Eventos: busca de informações de gerenciamento de execução do modelo:

O arquivo de entrada fornece o número de diagramas (ou listas) que comporão o modelo de execução, por exemplo, 6. O protótipo então busca os 6 nomes dos diagramas que deverão ser montados: "diagrama1", "diagrama2", ..."diagrama6".

Fluxo de Eventos: Montagem das listas de execução:

O protótipo solicita ao arquivo de entrada as informações relativas ao "diagrama1". O

protótipo recebe o número de blocos e de conexões (6 blocos e 6 conexões), e o passo de execução do diagrama (0,001 segundo).

Cada conexão constitui um fluxo de sinal de uma saída de bloco para uma entrada de outro bloco. O protótipo então recebe a "profundidade" (memória dinâmica) de cada sinal, e seus tipos (tipo de dado - tamanho de palavra do sinal), e cria os mesmos. O arquivo de entrada define profundidade mínima de 3 para todos os sinais (restrição para simplificar o problema, inicialmente - se evita checar a correção do modelo, quanto a este aspecto).

O protótipo solicita do arquivo de entrada os blocos: seu tipo, número de entradas, saídas e parâmetros, e a definição de cada um. Para o "diagrama1", o arquivo de entrada define um bloco constante (0 entradas, uma saída, um parâmetro), um bloco somador (2 entradas, uma saída, nenhum parâmetro), etc... (ver modelo no apêndice C). Para o somador, define que sua entrada 0 (primeira) vai "olhar" para a conexão (sinal) 0, a entrada 1 (segunda) vai olhar para o sinal 2. Cada bloco/sinal portanto tem um número que o identifica, e que é definido no momento da criação do bloco ou sinal, de acordo com a ordem estabelecida pelo arquivo de entrada.

O protótipo repete estes passos de acordo com o esquema geral da tabela 4.1, para os diagramas "diagrama2", "diagrama3", etc...

Fluxo de Eventos: coordenação dos diagramas:

O protótipo solicita ao arquivo de entrada os pares de conexão entre diagramas, ou seja, quais blocos de qual diagrama estão "enxergando" quais sinais de quais diagramas. As informações são definidas de acordo com a tabela 4.2.

Fluxo de Eventos: Busca das Informações de Saída:

O protótipo solicita ao arquivo de entrada quais sinais deverão ser enviados para vídeo, e quais sinais deverão ser armazendos para registro posterior, ou seja, enviados ao arquivo de saída, de acordo com a tabela 4.3.

Faixa Esperada	Tipo	Seção			
1..65535	UI	Numero de Diagramas (N)			
Nome válido de arquivo	Char*	Nome Diagrama1			Diagramas
...	...	...			
...	...	Nome Diagrama N			
0..65535	UI	Numero de Pares (M)			Pares de conexões entre diagramas
0..(N-1) (diagrama válida)	UI	Diagrama Origem 1	Origem 1	Par 1	
Sinal válido de (diagrama origem 1)	UI	Sinal Origem 1			
0..(N-1) (diagrama válida)	UI	Diagrama Destino 1	Destino 1		
BlocoTransporte válido De (diagrama destino 1)	UI	Bloco Destino (Transporte) 1			
...	...	...		...	
0..(N-1) (diagrama válida)	UI	Diagrama Origem M	Origem M	Par M	
Sinal válido de (diagrama origem M)	UI	Sinal Origem M			
0..(N-1) (diagrama válida)	UI	Diagrama Destino M	Destino M		
BlocoTransporte válido De (diagrama destino M)	UI	Bloco Destino (Transporte) M			

Tabela 4.1 - Arquivo de Entrada - informações do sistema - pacote ASCII

Faixa Esperada	Tipo	Seção	
0..65535	UI	Numero de Vetores de Saída (Sinais de Saída) (N)	
Diagrama válida 1	UI	Número do diagrama 1	Saída 1
Sinal válido do diagrama 1	UI	Número do Sinal 1	
...	UI	...	...
Diagrama válida N	UI	Número do diagrama N	Saída N
Sinal válido do diagrama N	UI	Número do Sinal N	

Tabela 4.2 - Arquivo de Entrada - lista de saídas do sistema - pacote ASCII

Fluxo de Eventos: Configuração do sistema de execução das listas O protótipo deve então solicitar ao núcleo de tempo real que crie uma tarefa que execute cada diagrama, e a associe a um semáforo. O protótipo deve definir como evento de disparo dos semáforos contadores que contemplem a taxa de execução de cada diagrama. O protótipo deve solicitar ao usuário o evento que irá encerrar a execução das listas (tempo, ou número de iterações)

Fluxo de Eventos: Execução dos algoritmos das listas: O protótipo então avalia os contadores do sistema e dispara os semáforos correspondentes, implicando na execução da tarefa de cada diagrama. Essa tarefa comando ao diagrama que execute as funções de transferência de todos os seus blocos.

Fluxo de Eventos: Acompanhamento do processamento pelo usuário: O protótipo envia para o vídeo as informações de cada passo de cálculo. O vídeo deve mostrar os sinais escolhidos, e seus valores atuais (recentemente calculados).

Faixa Esperada	Tipo	Seção			
1..65535	UI	N°Sinais (N)			
1..65535	UI	N°Blocos (M)			
>=0	Float	Passo			
1..65535	UI	Prof. Sinal 0			Profundidades dos Sinais
...	...	⋮			
...	UI	Prof. Sinal N-1			
∈ CtipoBloco	Inteiro	Tipo		Bloco 0	Blocos
0..65535	UI	No. Entradas (E)			
0..65535	UI	No. Saídas (S)			
0..65535	UI	No. Parâmetros (K)			
0..65535	UI	Entrada 0	Entradas		
...	...	...			
...	UI	Entrada E-1			
0.65535	UI	Saída 0	Saídas		
...	...	...			
...	UI	Saída S-1			
Float válido	Float	Parâmetro 0	Parâmetros		
Idem	...	...			
Idem	Float	Parâmetro K-1			
...	...	⋮		⋮	
...	...	⋮		Bloco M-1	

Tabela 4.3 - Arquivo de Entrada - pacote ASCII

Fluxo de Eventos: Encerramento e registro: O protótipo aguarda a ocorrência do evento definido pelo usuário. Em ocorrendo o evento, as tarefas são interrompidas, e o protótipo envia para o arquivo de saída os sinais escolhidos, com toda a sua história no tempo (cada iteração).

4.2.4 Classes do Cenário Principal - executar protótipo

A partir da descrição do cenário principal, definimos as seguintes classes:

- Protótipo: em geral cada cenário coresponde à criação de uma classe de controle, que coordena os serviços das demais classes. Essa classe poderia chamar-se "gerente de diagramas, ou de modelo".
- Entrada: é uma classe de fronteira cuja responsabilidade é fornecer os serviços de entrada para o protótipo, ou seja, interfaceia com o ator Arquivo de Entrada.
- Saída: classe de fronteira que fornece os serviços de saída para o protótipo. Interfaceia com os Atores Arquivo de Entrada, Arquivo de Saida, e Video.
- Modelo: classe de entidade que vai coordenar os diagramas, a manter as informações correspondentes às ligações entre diagramas.
- Diagrama: classe de entidade que coordena os blocos e sinais de cada diagrama, e suas ligações.
- Bloco: entidade que define um bloco e suas propriedades/serviços solicitados pelo diagrama a que pertencem.
- Sinal: entidade que define um sinal ou conexão, e suas propriedades/serviços solicitados pelo diagrama a que pertencem, ou por blocos com que se relacionam.

4.2.5 Diagramas de Seqüência para Cenário Principal - Executar Protótipo:

Diagrama Principal

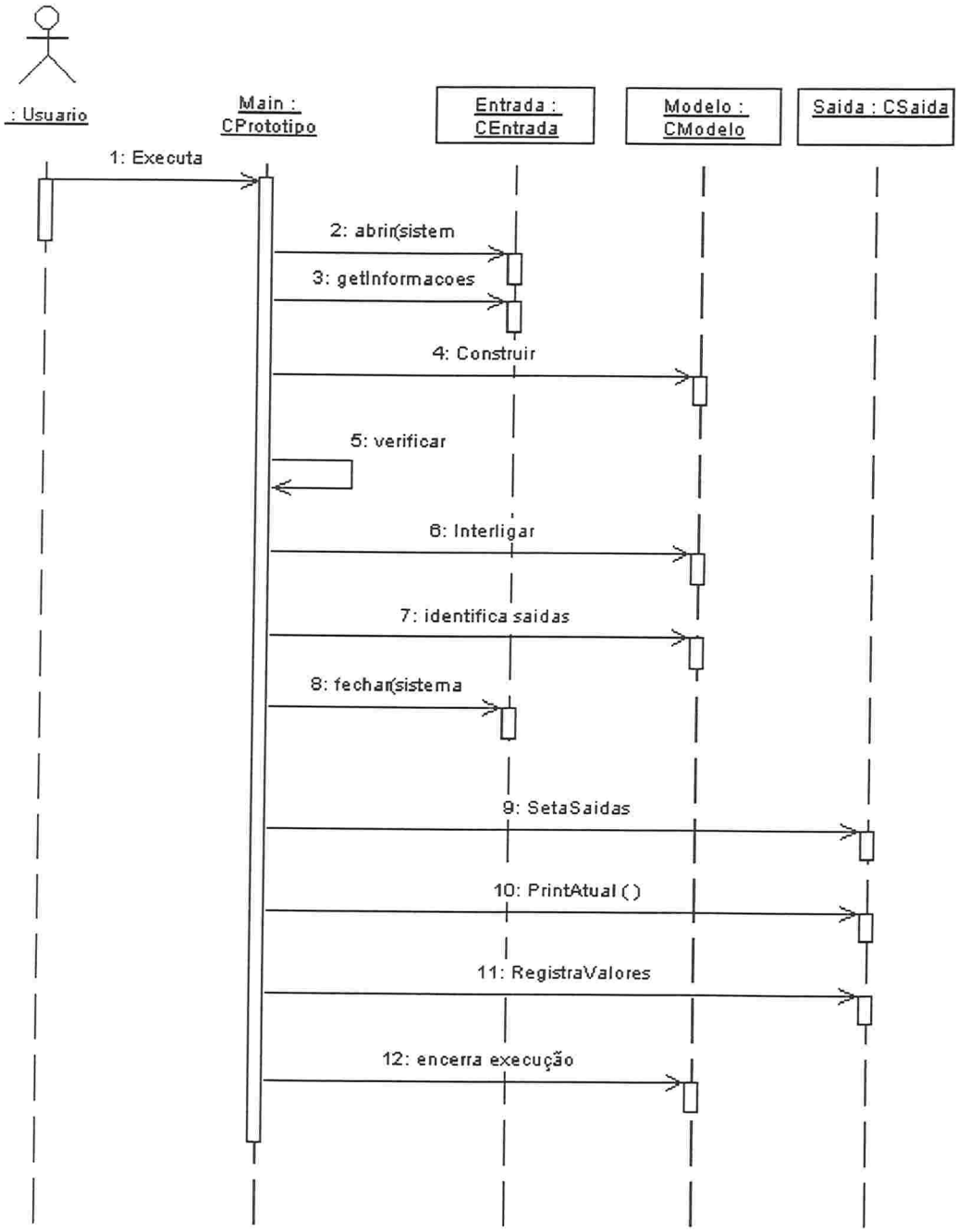
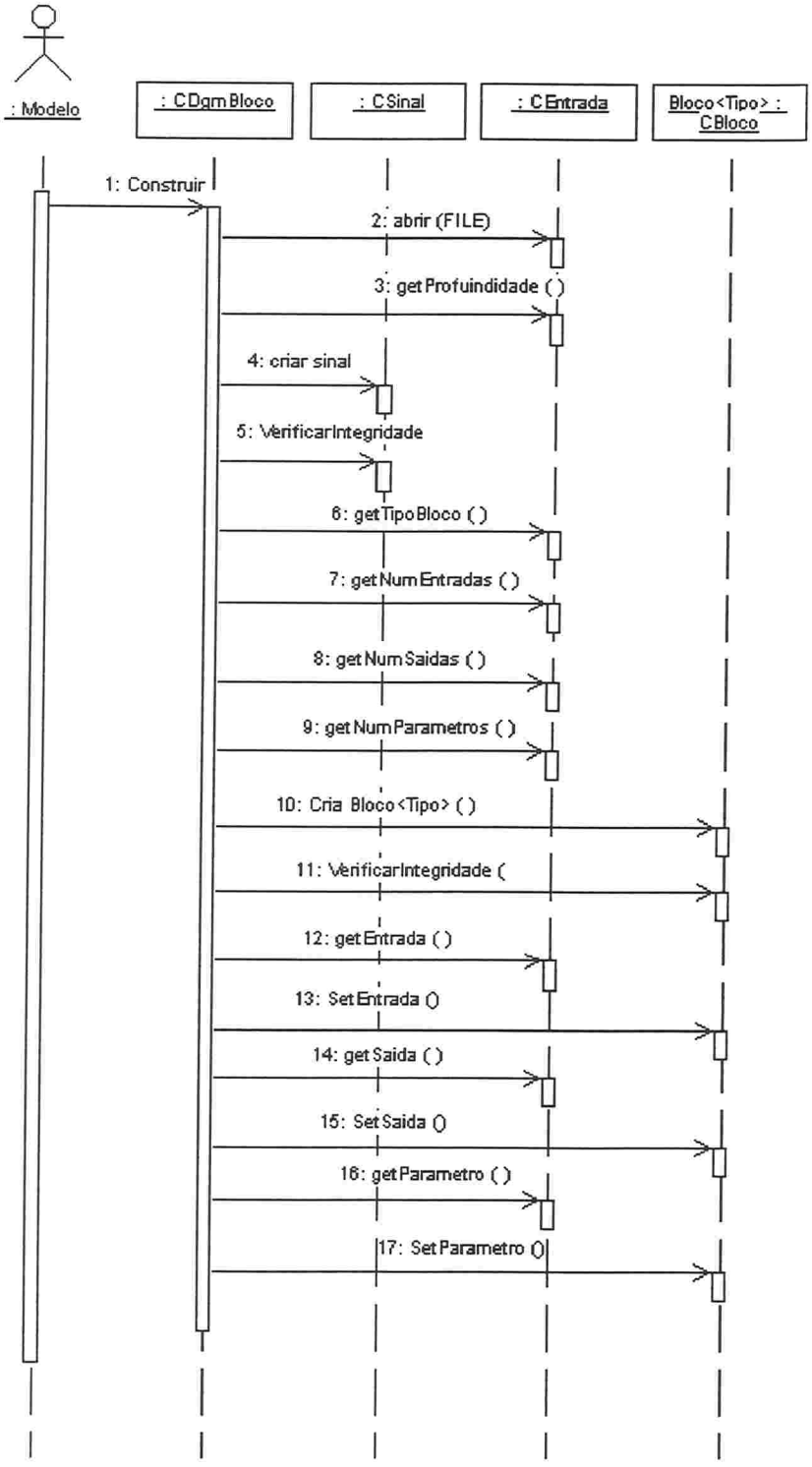
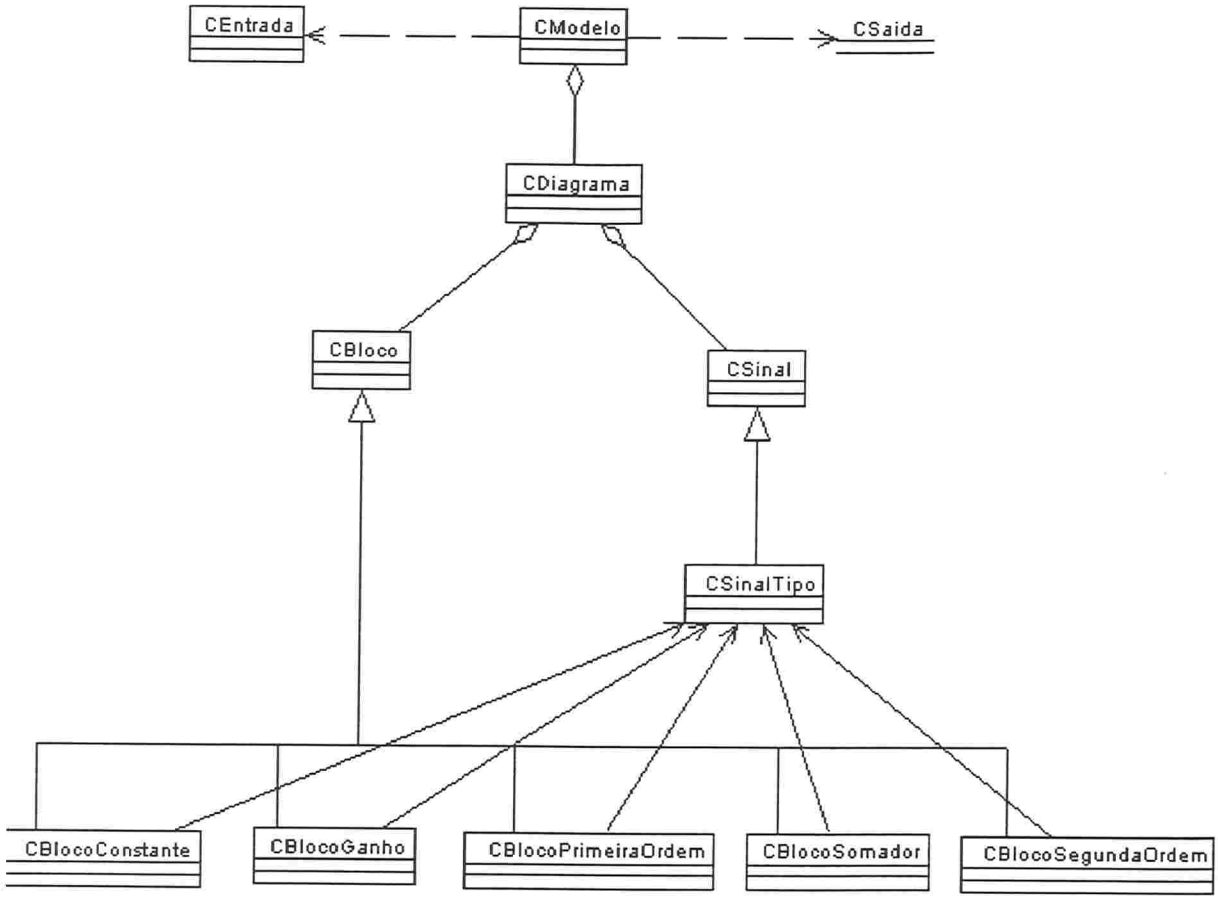


Diagrama Auxiliar - construir diagramas de blocos:



4.2.6 Diagrama de Classes



Este diagrama apresenta as classes-chave do problema e seus principais relacionamentos.

Até aqui, multiplicidade, papéis, e parametrizações não foram consideradas.

Este modelo agora deve ser analisado e refinado. Uma implementação já pode ser feita das classes básicas, como os blocos e sinais, e testes são realizados com as classes individualmente.

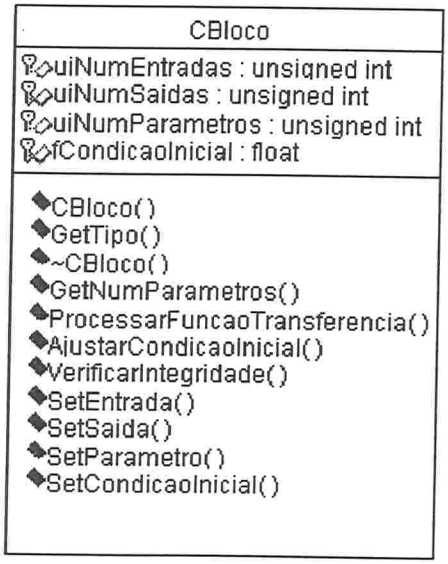
Neste modelo, o uso da herança, especializando os blocos e sinais de um tipo específico a partir de suas superclasses CBloco e CSinal, virtuais, permite manter a uniformidade de interface para que qualquer bloco ou sinal, e mesmo blocos e sinais de tipos que sejam criados posteriormente, sejam manipulados da mesma forma por um diagrama ou outra entidade.

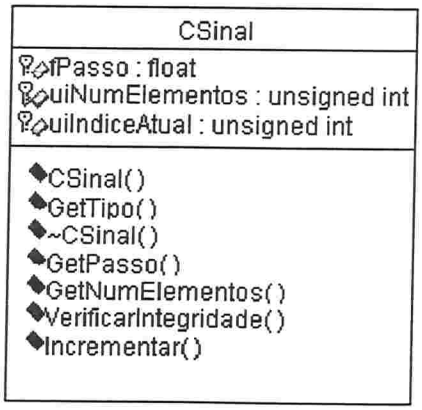
A classe CProtótipo não foi representada, por ser uma classe de controle, e não acrescentar maiores informações ao diagrama, no que diz respeito a sua estrutura lógica.

4.2.7 Implementação e Testes - Blocos e Sinais

Neste ponto, podemos definir, implementar, e realizar alguns testes, para validar a estrutura de blocos e sinais proposta.

As classes CBloco e CSinal são virtuais, e têm a seguinte estrutura:



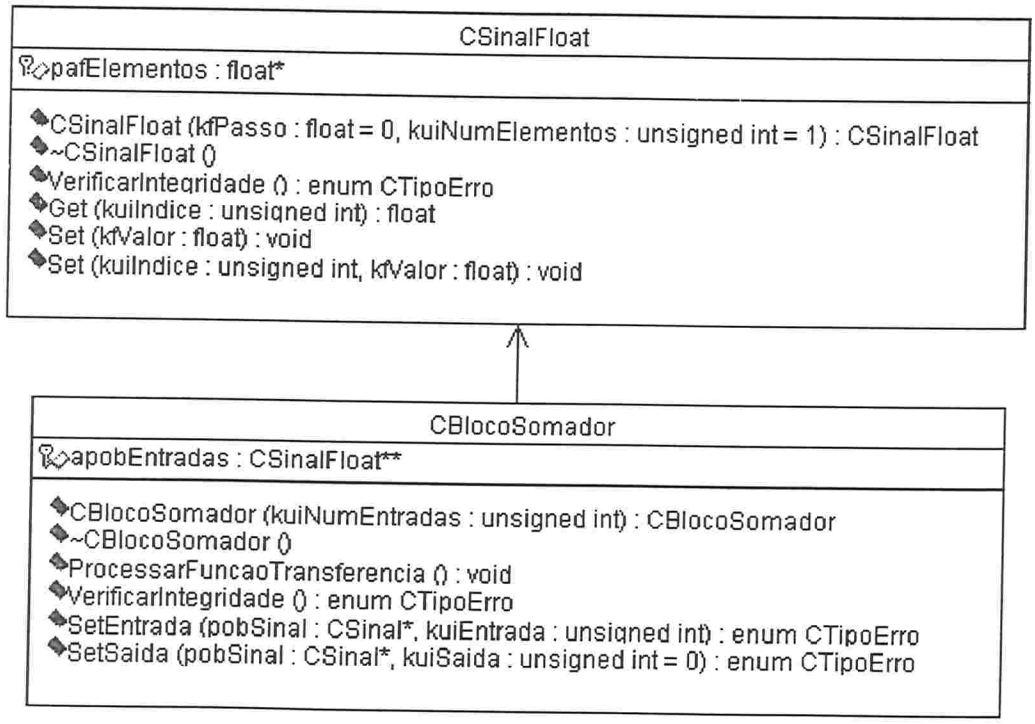


Essas classes não oferecem implementação para os métodos (são virtuais). Porém, o compotamento esperado está bem definido. Segue uma descrição dos métodos-chave:

- Processar função de transferência: o bloco utiliza os valores dos elementos de cada sinal de entrada (valor atual, e valores em instantes anteriores, se necessário), e seta o valor atual dos seus sinais de saída.
- SetEntrada, SetSaída permitem associar a cada entrada e saída do bloco, uma intância de CSinal específica, que vai fornecer seus serviços, como se fosse um *buffer*, dos valores atuais e anteriores.

- Incrementar: realiza a rotação do sinal (será implementado como um buffer circular), ou seja, permite manter a cada iteração do diagrama de blocos a semântica de "valor atual", "valor anterior", etc...

Como cada sinal tem um tipo específico (por exemplo, um determinado sinal pode conter um fluxo de valores em ponto flutuante, e outro em dupla precisão), existem alguns métodos-chave que serão implementados com a mesma assinatura nas subclasses (tanto blocos como sinais), garantindo o polimorfismo. Temos então, por exemplo:



Aqui, um bloco somador está relacionado, por suas entradas ou saídas, a sinais do tipo float. Os métodos implementados complementam o mecanismo-chave de interacão bloco-sinal:

- Get (no sinal) : este serviço retorna o valor atual do sinal para o bloco solicitante, ou o valor anterior, ou o valor correspondente à profundidade que for. Assim, o parâmetro *kuiIndice*, enviado pelo solicitante (um bloco) define se o sinal deve retornar o valor atual - Get(0), ou o valor anterior - Get(1) - e assim sucessivamente. Existem condições de teste que garantem que algum dos valores do sinal vai ser retornado, porém a correção lógica deve ser garantida através do estabelecimento de conexões de blocos a sinais que tenham a profundidade necessitada pelo bloco. Ou seja, na construção do

modelo, se deverá garantir que um bloco que solicita "entradas anteriores" a um sinal, o faça a sinais que têm essa capacidade (profundidade 2 ou maior).

- Set (método do sinal): permite definir o valor atual do sinal.
- SetEntrada, SetSaida: estabelece uma ligação das entradas e saídas dos blocos com os sinais respectivos.

Deve-se notar que, cada sinal vai estar associado a uma e apenas uma saída de um único bloco. E que este bloco deverá ter a responsabilidade de enviar a mensagem *Incrementar* ao seu sinal de saída, antes de atualizar seu valor atual (do sinal), para que o sinal reflita a sequência correta de valores no tempo. Esta análise aparece claramente nos diagramas de seqüência apresentados mais adiante.

Verifica-se também nessa etapa a inclusão de uma "classe" adicional, CTipoErro, que vai auxiliar na manipulação de blocos e sinais, oferecendo uma série de condições de erro que são verificadas, descritas a seguir:

```
enum CTipoErro
{       SEM_ERRO, /* Condicao normal */
        ERRO_ALOCACAO, /* Erro de alocação de alguma variável alocada dinamicamente */
        DADO_INCONSISTENTE, /* Algum atrbuto (ou combinação) com valores inconsistentes */
        ENTRADA_INEXISTENTE, /* Referencia a uma entrada inexistente */
        SAIDA_INEXISTENTE, /* Referencia a uma saida inexistente */
        SINAL_INVALIDO, /* Referencia a um sinal invalido ou inexistente */
        METODO_NAO_IMPLEMENTADO, /* Chamada a um metodo não implmentado na classe derivada*/
        BLOCO_DESCONHECIDO,       /* Tradutor nao conhece o tipo declarado */
        PARAMETRO_INEXISTENTE   /* Referencia a um parametro inexistente  */
}
```

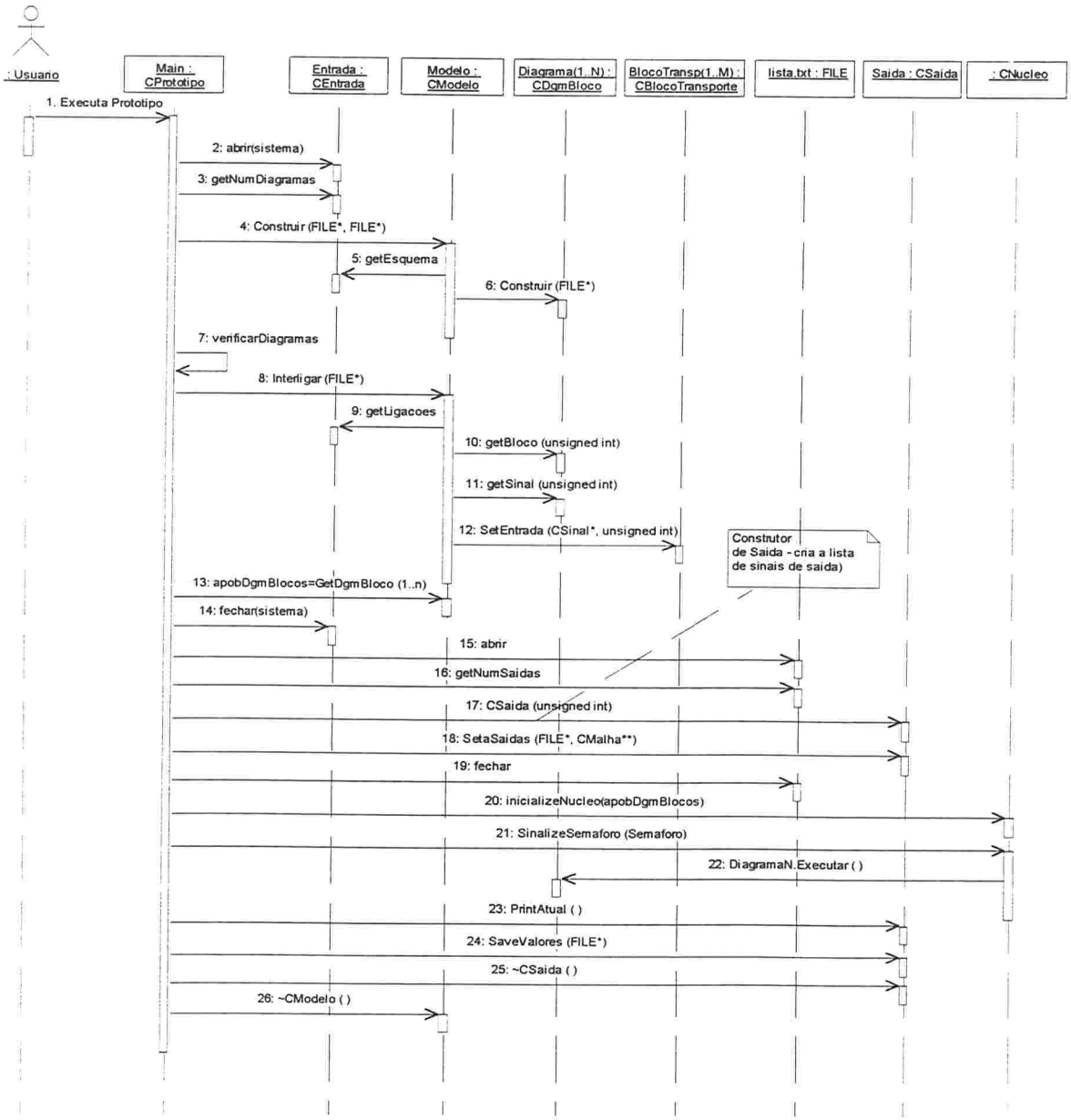
Note-se que a classe foi implementada em C++ como uma enumeração, já que cada instância basicamente constitui-se de um único atributo, o erro em questão.

4.3 Projeto e Implementação

Procedeu-se a partir dos diagramas expostos e dos testes de implementação realizados ao projeto detalhado do comportamento do protótipo, descrito no início do capítulo.

4.3.1 Diagramas de Sequência

Diagrama de Seqüêcia Principal Atualizado



As principais mudanças neste diagrama surgem a partir da adição das classes CNúcleo, FILE, e CBlocoTransporte.

A classe CNúcleo encapsula os serviços do núcleo de tempo real. A "classe" FILE reflete a decisão de implementação, de utilizar a biblioteca padrão de I/O do C++, e apesar de constar nos diagramas de classe, não foi implementada como tal.

Finalmente, a classe CBlocoTransporte. Essa classe, especializada de CBloco (e portanto é capaz de interagir com sinais e diagramas como um bloco qualquer), tem uma responsabilidade um pouco maior, de conectar diagramas diferentes entre si. Para isso, esse bloco se comporta basicamente como um ganho unitário (ou seja, na execução de sua função de transferência, lê seu sinal de entrada, e escreve em seu sinal de saída). Sua entrada, porém, não pode ser setada na forma convencional, e o bloco responde com um erro à mensagem "SetEntrada". Para outros blocos e sinais, este bloco não aparenta ter entradas. Porém, as entidades que conhecem a natureza do bloco (pode perguntar ao bloco, com uma mensagem "GetTipo"), ou seja, basicamente o Gerente de Diagramas, ou CProtótipo, "sabem" que ele possui um método especializado, "SetOrigem". Esse método é fundamentalmente idêntico a SetEntrada.

O CProtótipo utiliza as informações de conexão de diagramas para comandar aos blocos de transporte de cada um quais sinais, de quais diagramas, eles devem "olhar". Se um bloco transporte não tiver sua entrada conectada, ele "joga" zero na saída. Para o diagrama corrente, a conexão é transparente (ainda que o zero não seja logicamente esperado).

Este bloco surgiu da necessidade de garantir que cada diagrama de blocos trabalhe no seu conjunto de sinais, permitindo que taxas diferentes, ou preempções do núcleo de tempo real evitem que os sinais (que seriam de outra forma compartilhados) sejam corrompidos. Pode ser feita uma analogia deste bloco com um *sample-hold* de hardware. Cabe observar que, num diagrama de blocos, todos os seus blocos da classe CBlocoTransporte devem ser os primeiros a executar, e nesse período, as preempções devem ser interrompidas. É como realizar uma leitura de um *sample-hold*: as interrupções são desabilitadas. Porém esses blocos são muito rápidos, pois apenas fazem um *Get* num sinal, e um *Set* em outro.

Diagrama de Seqüência Auxiliar Atualizado

Esse diagrama, que descreve a sequência de criação de um diagrama de blocos, é basicamente o mesmo que o anterior, com alguns detalhes de implementação acrescentados.

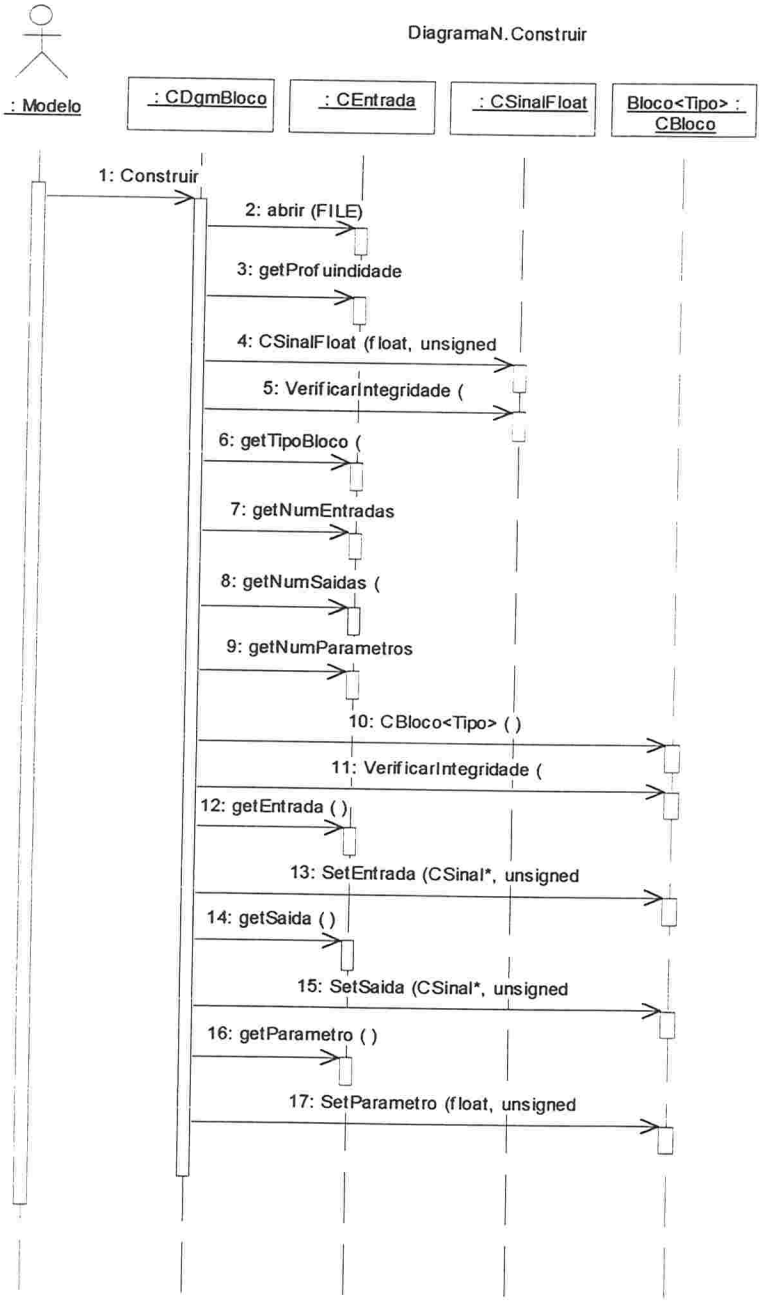
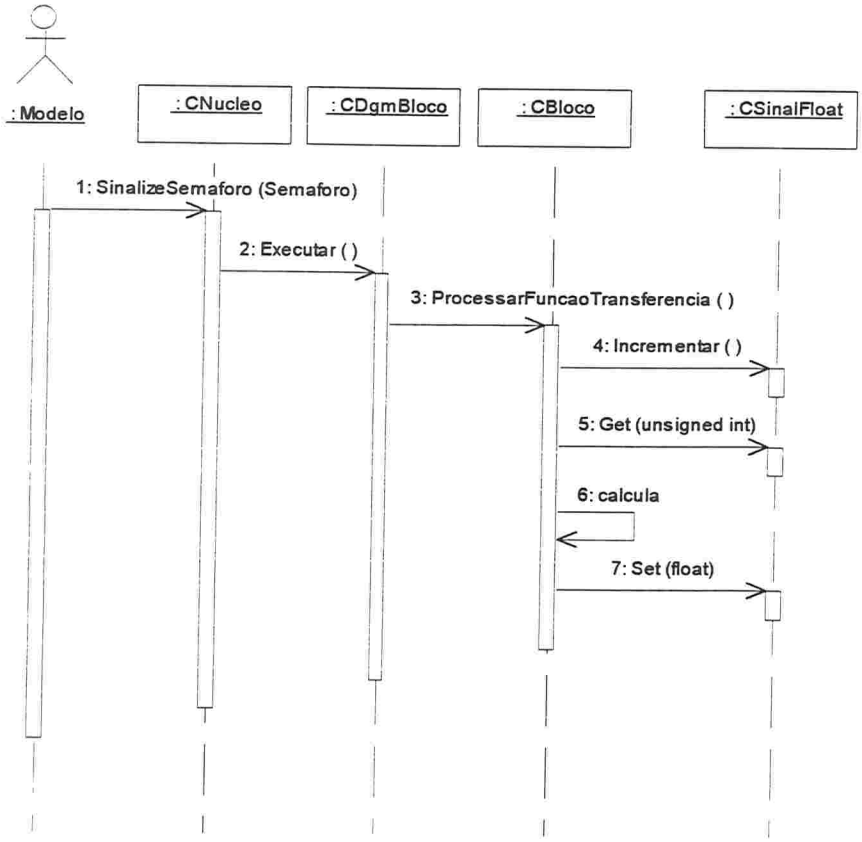


Diagrama de Sequencia Adicional: Executar Diagramas

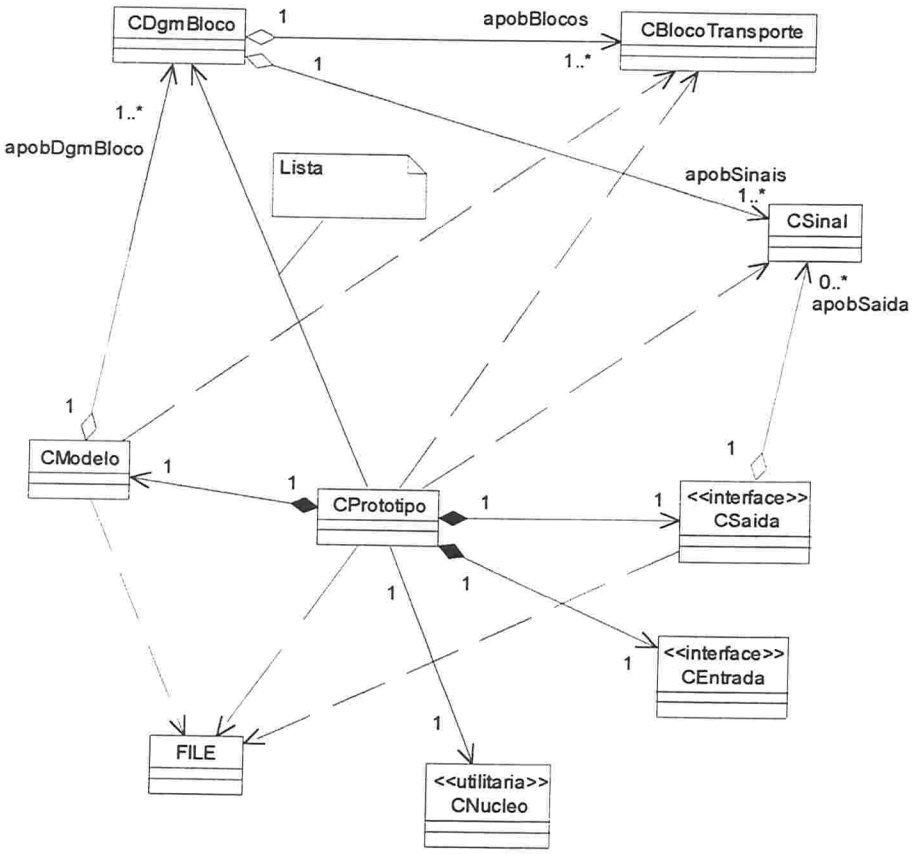


Esta é a sequência de lógica de execução de um diagrama de blocos. A mensagem 2, para o diagrama de blocos, ocorre uma vez. As mensagens subsequêntes, do diagrama para os blocos, se repetem para cada bloco do diagrama.

Abaixo, segue um detalhamento da estrutura lógica, com o devido refinamento dos relacionamentos entre as classes, suas multiplicidades, e os papéis correspondentes.

4.3.2 Diagramas de Classes

Diagrama de Classes Atualizado: Protótipo



Este diagrama proporciona uma visão completa do protótipo, com suas classes chave. Ele deveria ser expressivo o suficiente, mas vamos tecer alguns comentários.

As agregações por valor de CModelo, CSaida, e CEntrada, com multiplicidade 1 indicam as "peças" que compões o protótipo, e que o tempo de vida das instâncias correspondentes é o mesmo.

O núcleo aparece como uma classe utilitária que fornece serviços ao protótipo. De fato, na implementação as características do núcleo não estão de fato encapsuladas desta forma. A implementação de CNucleo não foi feita como um objeto, e seus serviços estão acessíveis em todo o programa (ainda que não utilizados).

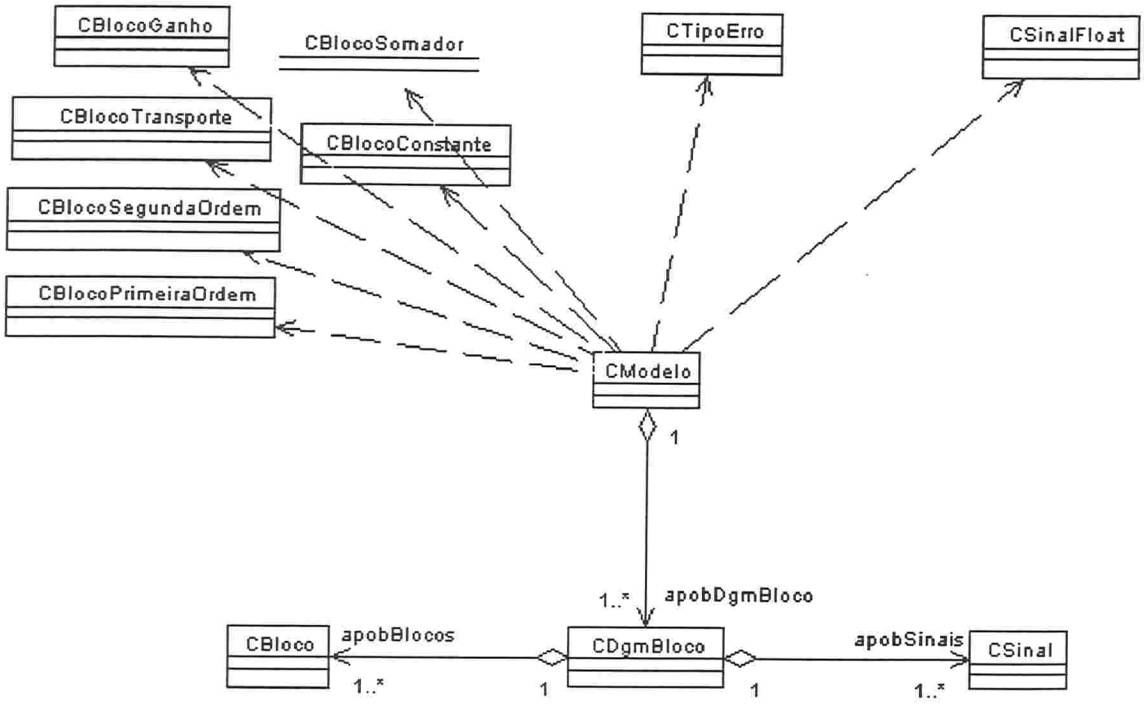
O Modelo do Protótipo agrega um ou mais Diagramas, e esses por sua vez agregam um ou mais blocos, e um ou mais sinais. Os papéis desses relacionamento estão indicados com o nome do atributo que modela a ligação, na implementação. Esses atributos são listas de ponteiros, mas as ligações poderiam ser parametrizadas por classes-lista, tal como indicado na dependência entre o Protótipo e os Diagramas de Blocos.

A classe FILE é auxiliar, e é uma representação, no modelo, de um elemento da implementação do protótipo.

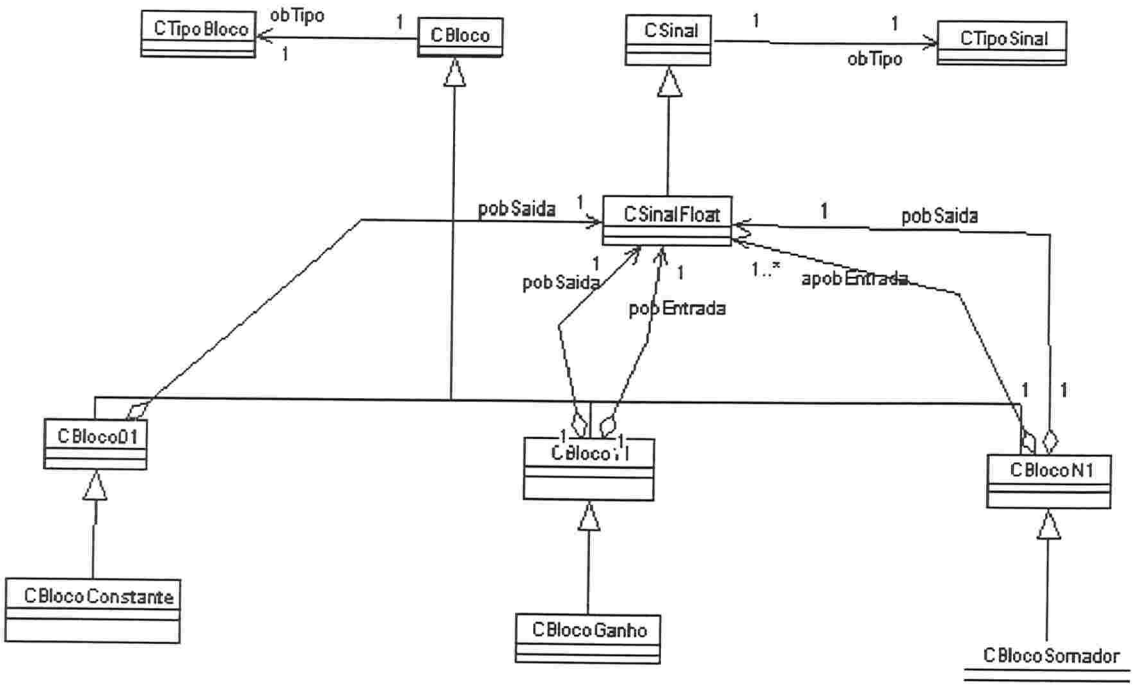
As dependências do Protótipo com sinais, e blocos, se justificam na medida em que o protótipo "conhece" essas classes, e intancia alguns objetos desses (na verdade, algumas *referências* a objetos destes), como auxiliares durante sua etapa de montagem e ligação dos diagramas. Esses blocos e sinais, portanto, não são blocos e sinais que pertencem a um diagrama e têm suas funções de transferência executadas, tal como nos diagramas de blocos.

Note-se que os Diagramas, o Modelo e o Protótipo não manipulam detalhes de blocos e sinais especializados, e não têm restrições quanto à manipulação de blocos ou sinais especializados que venham a ser criados.

Diagrama de Classes Atualizado: Diagramas de Blocos

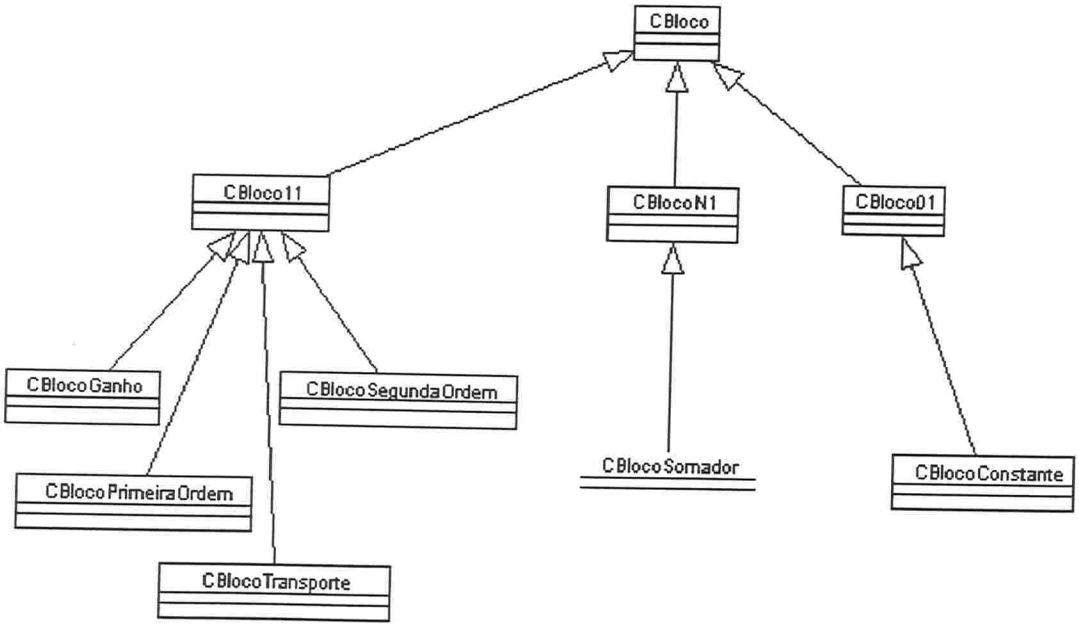


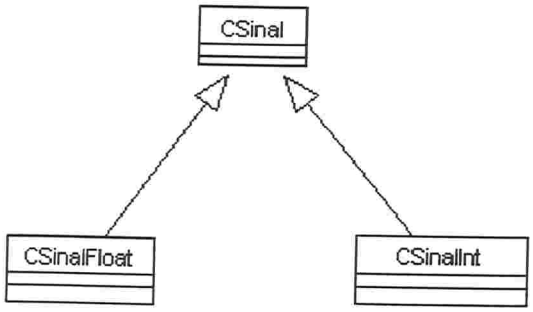
Diagramas de Classes Atualizados: Blocos e Sinais



Este diagrama evidencia a criação de classe intermediárias de herança (uma classificação em dois níveis), de CBloco e CSinal, para os blocos e sinais especializados. As vantagens se apresentam na implementação. Todos os blocos que tem uma entrada e uma saída podem ser especializados já da subclasse CBloco11, economizando a reprogramação de vários métodos e atributos nas subclasses. Apenas a função de transferência do bloco é definida. Suas características de conexão já foram herdadas. As vantagens estarão evidentes adiante, no diagrama de classes, onde se notará a simplicidade com que um novo bloco (ou sinal) pode ser definido e acrescentado ao sistema. Para as demais entidades do sistema, nada mudou (aqui está bem caracterizado o polimorfismo).

Diagramas de Classes Atualizados: Blocos e Sinais

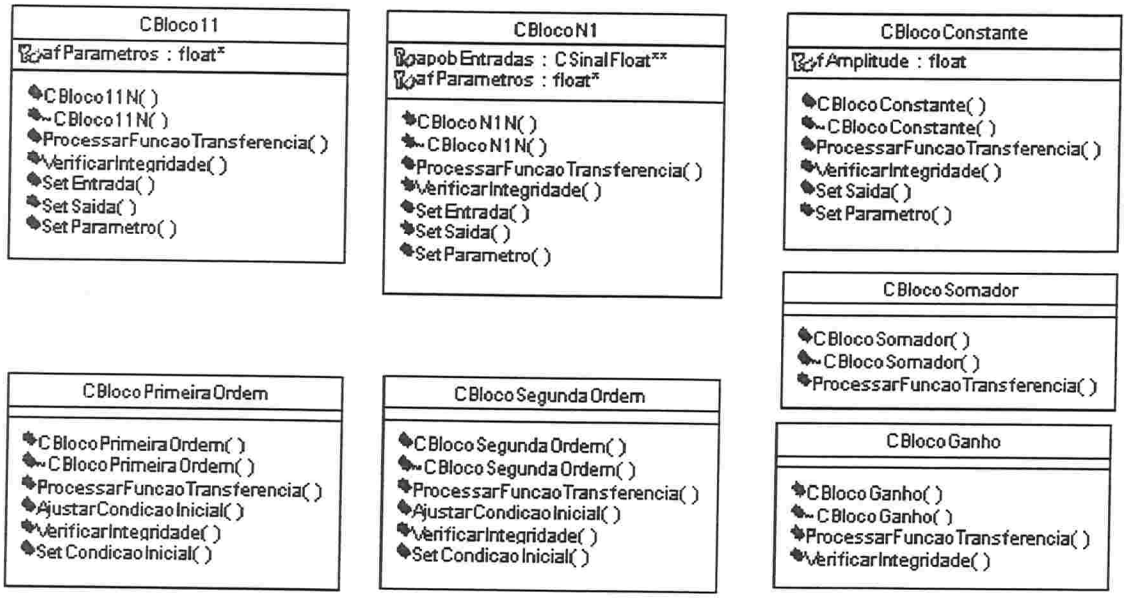




Esses diagramas complementam o anterior, e indicam a criação e anexação dos diferentes tipos de blocos e sinais ao sistema, sob a interface das superclasses CBloco e CSinal.

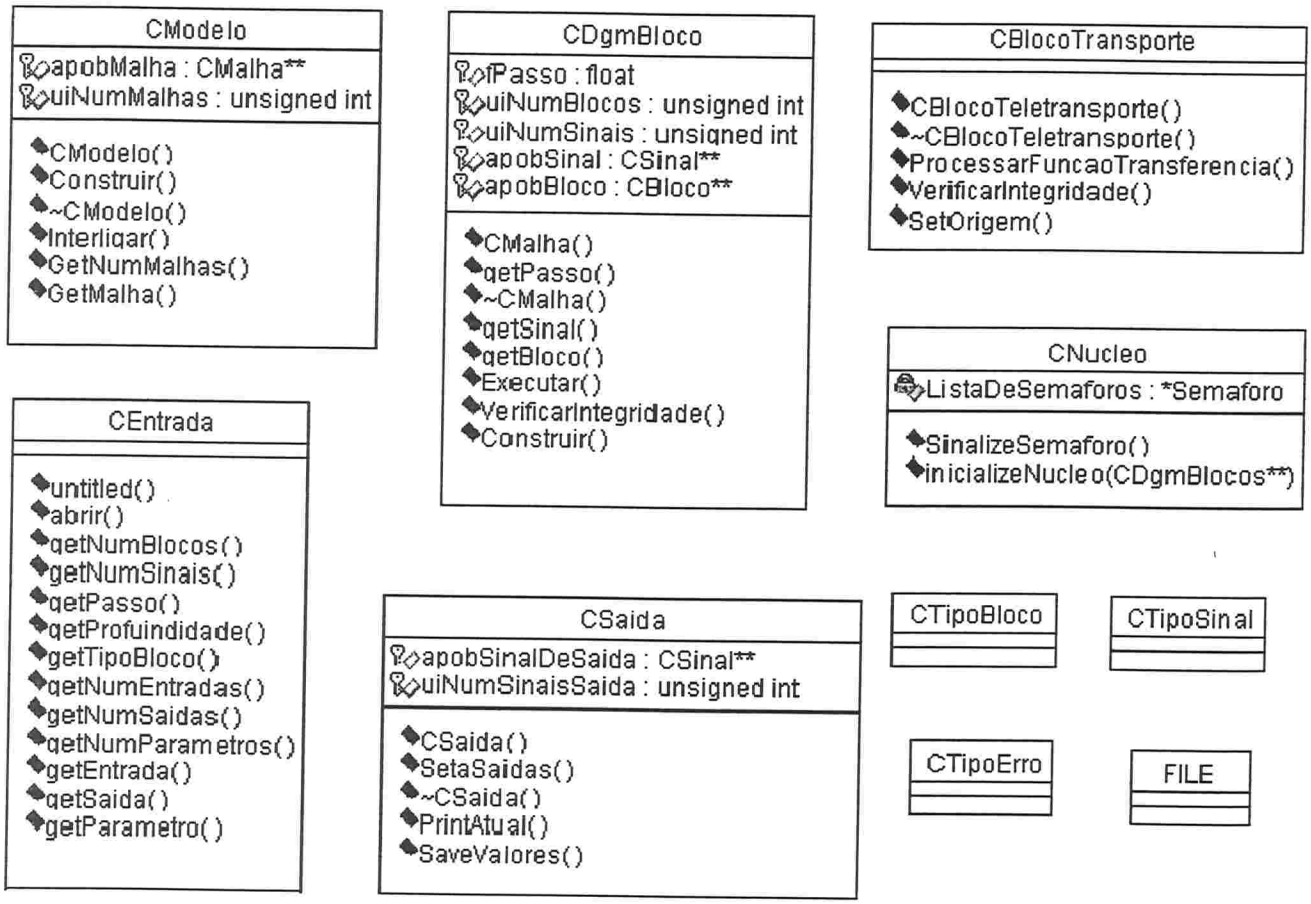
Diagrama de Classes: Operações e Atributos

Descrição das classes, suas operações e atributos, e alguns comentários.



As classes CBloco e CSinal, e as derivações de CSinal, são as mesmas já detalhas e testadas anteriormente. Este diagrama mostra a implementação do comportamento de conexão entre blocos e sinais nas subclasses CBloco11 e CBlocoN1. O somador, por exemplo, pode ser criado apenas definindo sua função de transferência. O comportamento restante já foi herdado.

Este diagrama mostra ainda a classe CBlocoConstante, com todos os atributos e operações definidos, isto é, ele foi derivado diretamente da classe CBloco. Se fez isso para evitar criar uma classe intermediária do tipo que tem nenhuma entrada e uma saída, já que o bloco constante, a princípio, é o único caso em que isso acontece.



Este diagrama mostra CEntrada, e CSaída, classes de fronteira que modelam o comportamento de entrada e saída descrito no cenário principal, com os métodos identificados nos diagramas de seqüência. Na classe CSaida, *PrintAtual* proporciona os serviços de vídeo, e *SaveValores*, os serviços de registro em arquivo. Essas classes, juntamente com CNucleo, CTipoErro, FILE, CTipoBloco e CTipoSinal são classes auxiliares na implementação do protótipo.

O diagrama também mostra CDgmBlocos, uma entidade que modela um diagrama de blocos, e CModelo, entidade que modela o próprio modelo de diagramas. CModelo foi criada para manipular os diagramas, como um "gerente" de diagramas, mas não seria estritamente necessária. O Protótipo poderia ter estas responsabilidades. Como Protótipo, porém, não foi implementado como classe, mas houve um "mapeamento" dos seus atributos e operações para o *Main*, o programa principal em C++, CModelo facilita a leitura do código, e o torna mais modular.

As entidades-chave descritas até aqui foram implementadas em C++, diretamente através de classes, e testadas individualmente. Os mecanismos auxiliares foram implementados com a estrutura tradicional de C++.

4.3.3 Aspectos de Tempo Real na Arquitetura de Blocos

O princípio-chave do uso de um núcleo de tempo real para executar os modelos de controle é permitir que as tarefas menos prioritárias, possam ser realizadas nos tempos que lhe "sobram".

É uma característica desejável que a arquitetura básica seja capaz de executar algoritmos em taxas diferentes. Por exemplo, a transdução de potência pode ser realizada por software, numa taxa de 1 ms, enquano o cálculo do controle pode ser realizado numa taxa de 10 ms, sobre a média dos 10 últimos pontos da transdução.

Supomos a tarefa A rodando em 1 ms, e a tarefa B rodando em 10 ms. Podemos executar as duas na taxa mais alta, 1 ms, o que pode ser uma restrição forte para o núcleo, em termos de tempo de execução dos cálculos de todo o modelo.

Alternativamente, podemos manter a tarefa B rodando sobre um conjunto de valores de saída produzidos por A (o que é possível através dos blocos de "transporte"). Enquanto a tarefa A roda a cada milisegundo, a tarefa B tem 10 ms para rodar, e nesse intervalo é interrompida 10 vezes por A (através do núcleo de tempo real). O sistema deve ser capaz de garantir que em 10 ms, A pode rodar 10 vezes, e B uma vez, adicionando-se aí os tempos de chaveamento entre tarefas, e outras interrupções eventuais de maior prioridade (emergências, por exemplo). Essa é uma restrição de tempo mais leve para o sistema, que tem mais tempo disponível para cálculo.

Outras tarefas, como as de comunicação entre módulos, tarefas de interface e gerenciamento, etc,...deverão, a princípio, rodar no tempo que lhes sobrar.

O diagrama abaixo ilustra o esquema.

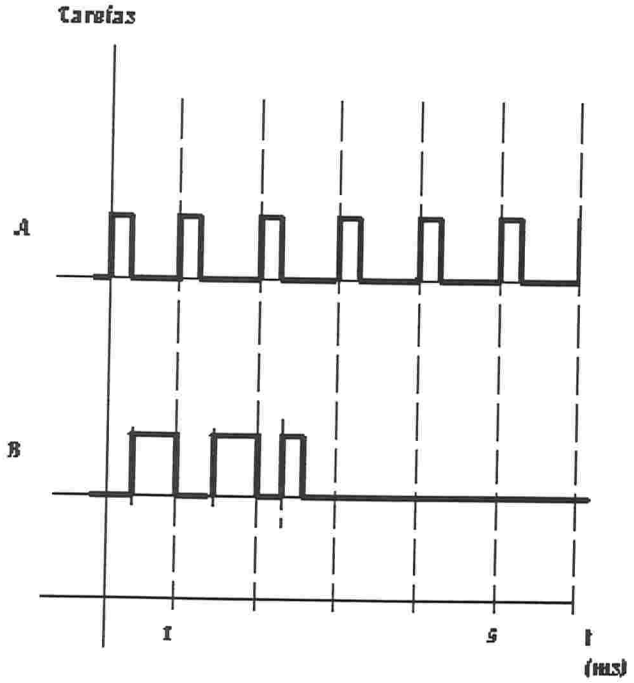


Figura 4.1 - Diagrama de tempo - duas tarefas periódicas

Deve-se notar neste diagrama que a tarefa B não teria tempo de rodar completamente a cada 1 ms. Como a tarefa B, de fato precisa rodar apenas a cada 10 ms, então ela pode rodar em 2,5 "sobras" de tempo da tarefa A, que é prioritária e deve rodar a cada milisegundo.

Na realização deste protótipo, cada diagrama é considerado uma tarefa com uma determinada taxa de execução. O cálculo das funções de transferência de diagramas menos prioritários (quem têm taxas mais baixas, ou períodos maiores, para executar) podem ser interrompidos, uma vez que os blocos de transporte "amostram" os valores que serão trabalhados nos cálculos de um diagrama. Os dados não são, então, corrompidos.

4.4 Testes e Documentacão

De acordo com os requisitos expostos e o cenário descrito, podemos enumerar os seguintes testes de implementação:

- teste da estrutura de blocos e sinais: um diagrama de blocos modelado com a estrutura de blocos e sinais construída no protótipo apresenta um resultado matematicamente correto?
- teste da estrutura de diagramas múltiplos: vários diagramas interconectados interagem corretamente? Um diagrama que utiliza sinais de outro diagrama apresenta valores corretos?
- teste de desempenho: quanto tempo os blocos típicos levam para executar completamente suas funções de transferência (incluindo os mecanismos complementares, ou seja os serviços dos sinais)?
- os mecanismos de tempo-real trabalham corretamente? Um diagrama que sofre uma "preempção" preserva seus resultados?

Para avaliar estas questões dois testes foram realizados.

No primeiro, uma estrutura simples utilizando blocos do tipo ganho, somador e primeira ordem foi implementada diretamente no corpo de um programa de teste (os blocos e sinais foram instanciados e conectados diretamente, no programa).

Este programa de teste incluiu sinalizações nas portas de saída digital de uma placa de arquisição de sinais.

O programa, compilado em diferentes compiladores, e com diferentes bibliotecas de emulação rodou em algumas configurações de hardware alternativas, e foram realizadas medidas dos serviços básicos oferecidos pela estrutura bloco-sinal.

Abaixo, apresentamos uma tabela com alguns dos resultados obtidos:

Borland C 3.1 com templates
Modo de memória Large
Uso de emulação (float - 32 bits)
Medidas (em microsegundos):
t1: Sinal[i].Incrementar( );
t2: Bloco[i].ProcessarFunçãoDeTransferencia( );
t3: idem t2, sem o tempo de chamada (basicamente, o tempo de acesso à tabela virtual);
t4: Sinal[i].Set(float)
...
t7: Sinal[i].Get( )

	t1	t2			t3			t4	t7
		i=1	2	3	1	2	3		
mín	8,52	-	-	-	-	-	-	2,22	2,40
máx	9,50	166	214	560	150	196	548	3,21	3,15

Tabela 4.4 - Exemplo de planilha de resultados de testes de desempenho

Os resultados mais importante dos testes iniciais de desempenho são os seguintes:

- A arquitura implementada em C++ pode ser viável em comparação com uma arquitetura implementada em C "puro". Os benefícios do uso do C++ e da orientação a objetos podem superar o eventual incremento necessário na plataforma de hardware.
- A sobrecarga relativamente pequena das chamadas a blocos frente à seu tempo de execução indica a necessidade de acelerar o processamento matemático.
- O núcleo de tempo real apresenta uma sobrecarga de tempo não-desprezível em relação à estrutura de execução. Deve-se achar um balanço conveniente entre o grau de utilização de tarefas independentes, e a modularidade na execução (diagramas diferentes sendo executados numa mesma tarefa, por exemplo).

O segundo teste realizado envolveu a criação e execução de um modelo a partir da estrutura do protótipo.

A partir do diagrama básico de blocos descrito no apêndice C, foram criados os pacotes ASCII necessários para a composição do modelo, segundo as tabelas 4.1 a 4.3.

Os pacotes (anexo II) definem, de acordo com o cenário, 6 diagramas de blocos. Cada diagrama corresponde à estrutura básica. Porém, cada par (três pares) interage entre si: o segundo diagrama tem como sinal de referência a saída de um bloco do primeiro diagrama.

Os três pares foram criados com a intensão de avaliar a modularidade do sistema. Os valores dos sinais sendo processados foram acompanhados em vídeo, em uma estrutura como a demonstrada a seguir:

```
(...)
273.770874, 17.166702, 273.770874, 17.166702, 295.537689, 20.629845,
273.776733, 17.166719, 273.776733, 17.166719, 295.548950, 20.629847,
273.782593, 17.166737, 273.782593, 17.166737, 295.560211, 20.629848,
273.788452, 17.166754, 273.788452, 17.166754, 295.571472, 20.629850,
273.794312, 17.166771, 273.794312, 17.166771, 295.582733, 20.629852,
273.800201, 17.166788, 273.800201, 17.166788, 295.593994, 20.629854,
273.806091, 17.166805, 273.806091, 17.166805, 295.605255, 20.629856,
Sinal 0: 3
Sinal 1: 3
Sinal 2: 30000
Sinal 3: 3
Sinal 4: 100
Sinal 5: 100
(fim de execução )
```

Tabela 4.5 - Saída em Vídeo

Os valores indicados permitiram analisar se os pares de diagramas estabilizavam em valores coerentes com o esperado segundo simulações realizadas no SMO ou Vissim.

No final da tabela 4.5 se verifica uma indicação de quantos valores anteriores, a partir do último valor, dos sinais escolhidos e definidos nos pacotes ASCII, foram registrados no Arquivo de Saída (em disco rígido).

Os valores registrados podem ser importados, por exemplo, no Vissim, e ter seu "comportamento dinâmico" avaliado. Abaixo, o resultado de um dos teste, envolvendo um dos pares de diagramas (portanto, dois sinais, as "saídas" de cada diagrama).

Figura 4.2 - Saídas dos modelos - valores registrados e simulados

Este gráfico mostra o resultado do modelo descrito no apêndice C, modelado e executado pela estrutura implementada. Ele corresponde a dois modelos daqueles (na verdade, 6, de acordo com o cenário descrito), onde definimos que os sinais de saídas de dois dos diagramas serão registrados após o término da execução.

Os valores dos sinais foram importados e plotados no Vissim, junto com os resultados da simulação dos dois modelos.

Este teste procurou validar a coerência lógica do modelo proposto.

4.5 Documentação

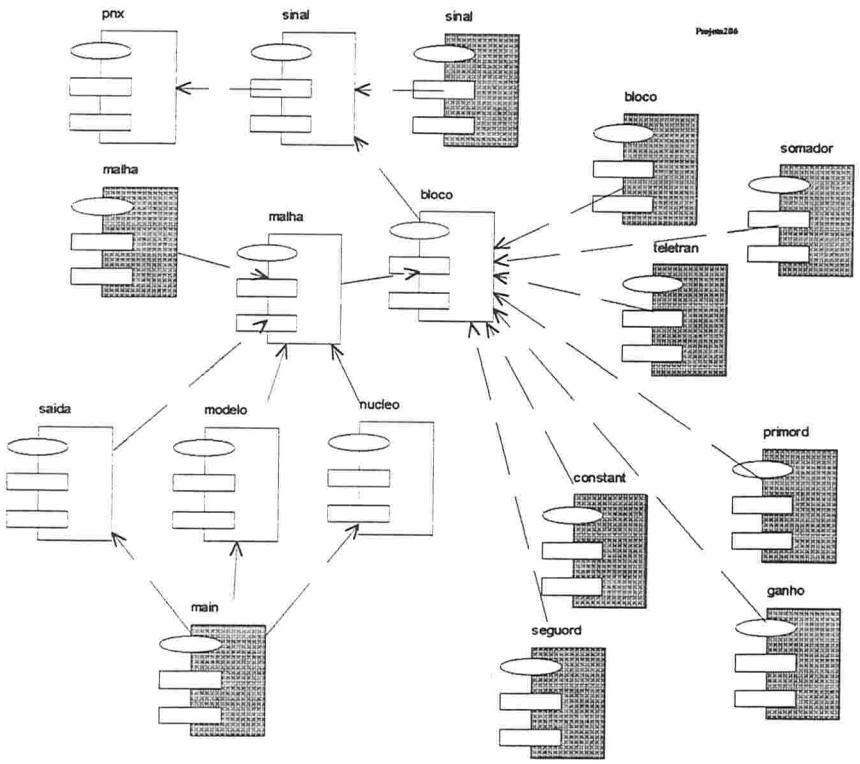
A documentação foi toda desenvolvida em uma ferramenta CASE, a *Rose*, para C++, e contém os diagramas apresentados neste trabalho, e descrições detalhadas das classes, operações, atributos, e considerações feitas.

A documentação compõe-se do conjunto de diagramas já apresentados nas seções anteriores, entre outros, auxiliares, construídos e armazenados no repositório da ferramenta CASE, em conjunto com comentários e descrições complementares.

Um dos diagramas disponíveis, a apresentado adiante, proporciona a visão de componentes da arquitetura.

Diagrama de Componentes

O diagrama abaixo representa a estrutura de implementação do protótipo de software. Em C++, os blocos escuros são os arquivos ".cpp", e os claros, os ".h".



4.6 Conclusões

Neste capítulo, foi desenvolvido um protótipo de sofware básico para reguladores capaz de criar e manipular *blocos*. Esses blocos são genéricos (podem ser um I/O digital ou analógica, por exemplo). Recebem dados em suas entradas, manipulam os mesmos com suas *funções de transferência*, e escrevem os resultados em suas saídas. Essas entradas e saídas são de um tipo qualquer, e são modeladas pelos *sinais*. Esses sinal é basicamente um buffer circular que permite manter registros da história de uma entrada ou saída de bloco.

Essa estrutura básica permite executar algoritmos de controle modelados por diagramas de blocos e ladder.

Associada a uma estrutura que manipula *Diagramas* (conjuntos de blocos e sinais inter-relacionados), e a um núcleo de tempo real, permite realizar modelos baseados em *estados*, onde para cada estado temos um Diagrama (seja ele sequencial ou concorrente).

Espera-se que este protótipo possa evoluir em ciclos subsequentes na direção de realizar um software completo, capaz de efetivamente realizar controle. Para isso, podemos observar que a implementação de alguns blocos além dos considerados aqui já permitiriam teste preliminares (por exemplo, os blocos de I/O referidos antes).

Capítulo 5

5.1 Conclusões do Trabalho

A arquitetura básica de blocos e sinais proposta parece contemplar as necessidades de um núcleo de execução dos algoritmos de controle dos reguladores de tensão, velocidade e estabilizador de sistema de potência, as necessidades de um sistema de monitoração e registro, e das suas interfaces de programação.

Os resultados obtidos até aqui precisam ser revisados e os testes complementados. É essencial garantir que as considerações de projeto estejam realmente satisfazendo as necessidades do cliente. Em particular, uma estrutura de controle completa deverá ser programada através do protótipo, para avaliação dos resultados.

As ferramentas de análise e projeto auxiliados por computador, em geral disponíveis para sistemas de software do tipo *de negócio*, dão suporte a um projeto bem documentado, e realizado de forma mais rápida, desde que a equipe de desenvolvimento esteja bem familiarizada com a notação que se utiliza.

As estruturas de software auxiliares criadas nesta etapa (classes auxiliares, Modelo, Saída, Entrada....) devem ser descartadas a princípio. Os elementos básicos, blocos e sinais, deverão passar por uma série de testes para avaliar se estão realmente satisfazendo requisitos do usuário, e então se deverá partir para um novo ciclo de desenvolvimento. Em particular, deve-se avaliar as sobrecargas de processamento das estruturas de controle, como o núcleo de tempo real, e os mecanismos de tabelas virtuais, etc, do C++.

Apêndice A

UML (Unified Modeling Language)

Resumo dos Elementos definidos em UML

"Casos de Uso"

O comportamento do sistema é como o sistema age e reage, e pode ser caracterizado por um conjunto de *casos de uso*. Um caso de uso é uma funcionalidade executada pelo sistema, em resposta a um estímulo proveniente de um *ator* externo. Eles estabelecem um veículo para a captura dos requisitos de um sistema através da ótica do usuário.

Um *ator* é algo ou alguém que deve interfacear com o sistema em desenvolvimento.

Um diagrama de caso de uso é uma descrição gráfica do sistema, que mostra seus atores e casos de uso identificados.

A documentação de um caso de uso consiste de uma breve descrição e um fluxo de eventos.

Exemplo de caso de uso: página 41.

Classes

Um objeto é algo que tem um estado, um comportamento e uma identidade.

O estado de um objeto é uma das situações possíveis em que o objeto pode existir. O comportamento determina como um objeto age e reage às solicitações de outros objetos. Cada objeto tem um identidade única, mesmo se o seu estado for idêntico ao de outro objeto.

Uma *classe* é uma definição abstrata de um conjunto de objetos que compartilham uma estrutura e um comportamento comuns. Devem possuir nomes que caracterizem bem a abstração. As classe focalizam o "que" e não o "como".

Uma classe é representada da seguinte forma: página 62.

Classe podem possuir *estereótipos*, que são classificações das classes. Estereótipos comuns são: fronteira, entidade, controle, exceção, metaclasse e utilitária.

Uma classe de fronteira modela a comunicação entre a visinhança do sistema e seus processos internos. Uma classe de entidade modela a informação e o comportamento associado. Uma classe de controle modela o comportamento específico de um caso de uso.

Interação entre Objetos

A interação entre objetos pode ser representada graficamente em um *diagrama de seqüência*, que mostra a existência de objetos e as interações entre os objetos identificados.

Exemplo nas páginas 56, 57.

A "linha de vida" de um objeto é representada pela linha tracejada vertical.

Mensagens são indicadas por setas horizontais direcionadas do objeto *cliente* (transmissor) para o objeto *servidor* (receptor). O nome das mensagens é indicado sobre esta linha.

Descrições adicionais podem ser acrescentadas para detalhar o diagrama.

Um *diagrama de colaboração* é uma representação alternativa da interação entre objetos.

Os objetos são representados por retângulos. Uma ligação de interação, representada por uma linha, é desenhada entre os dois objetos que se comunicam. A ligação é anotada com o nome da mensagem, direcionada do objeto cliente ao objeto servidor. Uma ligação também pode ser definida com uma seta de retorno de dados.

Diagramas de Classes

Classes são descobertas analisando-se os casos de uso e cenários desenvolvidos para o sistema. As classes de fronteira são acrescentadas ao modelo a partir da análise dos pares ator/caso de uso. As classes de entidade são acrescentadas a partir do exame dos substantivos e frases substantivas nas descrições dos casos de uso e cenários.

Classes de controle são acrescentadas na proporção de uma para cada caso de uso.

Cartões CRC podem ser usados para descobrir classes.

Um pacote é um mecanismo de uso geral para organizar elementos em grupos de elementos.

Um *diagrama de classe* é uma visão de algumas ou todas as classes e pacotes do modelo. Normalmente existem vários diagramas de classes.

Exemplo nas páginas 60, 61, 62.

Relacionamentos

Todos os sistemas sincronizam muitos objetos os quais colaboram entre si para produzir a funcionalidade necessária. Dois tipos importantes de relacionamentos são *associações* e *agregações*. Podem existir ainda os relacionamentos de *dependência*, e *herança*.

Uma associação é uma conexão entre duas classes que representa uma comunicação. Pode ter um nome. Nomes de papéis podem ser usados. Pode ser uni ou bidirecional (*default*).

Multiplicidade é o número de instâncias que participam de uma associação. É representada próximo ao final de uma linha de associação. Cada lado de uma associação pode ter um indicador de multiplicidade.

A agregação é uma forma especializada de associação, no qual o todo se relaciona com as suas partes. Cada lado de uma agregação pode ter uma multiplicidade.

Uma classe pode ter uma associação ou agregação reflexiva, quando dois objetos de uma mesma classe estão relacionados.

Herança define um relacionamento entre classes onde uma classe compartilha a estrutura/comportamento de uma ou mais classes. Define uma hierarquia de abstrações onde uma subclasse herda de uma ou mais superclasses. Pode ser simples ou múltipla.

Uma subclasse herda atributos, operações relacionamentos, e pode incluir atributos, relacionamentos e operações adicionais, além de refinar operações herdadas.

Exemplo nas páginas 58, 59.

Em alguns casos, a informação pertence à ligação entre os objetos. Então pode ser usada uma classe de ligação.

Os relacionamentos podem ser determinados examinando-se os cenários para descobrir as necessidades de comunicação entre os objetos.

Operações e Atributos

Uma classe incorpora um conjunto de responsabilidades que definem o comportamento dos objetos na classe. As responsabilidades são realizadas pelas operações definidas para a classe.

Cada operação deve realizar uma tarefa única e coesa. As operações mostram como um objeto age e reage ao receber uma mensagem.

Um atributo é uma característica de uma classe. Seu valor é o valor de um atributo para um objeto particular. Apenas atributos e operações relevantes (aplicáveis ao problema que está sendo resolvido) devem ser modelados.

Encapsular é esconder os detalhes da implementação do mundo exterior, protegendo o estado interno de um objeto, e protegendo código de *clientes* de mudanças na implementação do objeto.

Diagramas de Estados

Um diagrama de transição de estados representa o ciclo de vida do objeto em termos dos seus estados possíveis, e das transições entre estes estados.

O estado de um objeto pode ser identificado a partir dos valores de seus atributos, e de existência ou não de determinadas ligações.

Homogeneização

A medida que casos de uso vão sendo acrescentados, é necessário realizar a revisão e homogeneização do modelo. Classes podem ser combinadas/eliminadas, e os pacotes

podem ser reestruturados. Os nomes devem ser mantidos os mais coerentes possíveis com o domínio modelado.

Arquitetura

A visão lógica do sistema é capturada através dos diagramas de classes que contém pacotes, classes, e relacionamentos que representam as abstrações chave do desenvolvimento.

A visão de componentes (ou de desenvolvimento) se preocupa com a organização modular do software no ambiente de desenvolvimento. Um diagrama de componentes mostra a alocação de classes e objetos em componentes de implementação, assim como suas dependências de compilação.

Exmplo: página 70.

A visão de processo mostra a alocação dos componentes em processos.

A visão física mapeia os processos aos nós de processamento.

A visão de processo e física não foram utilizadas formalmente no projeto deste protótipo.

A visão de casos de uso procura integrar e validar as 4 visões da arquitetura do sistema.

♦

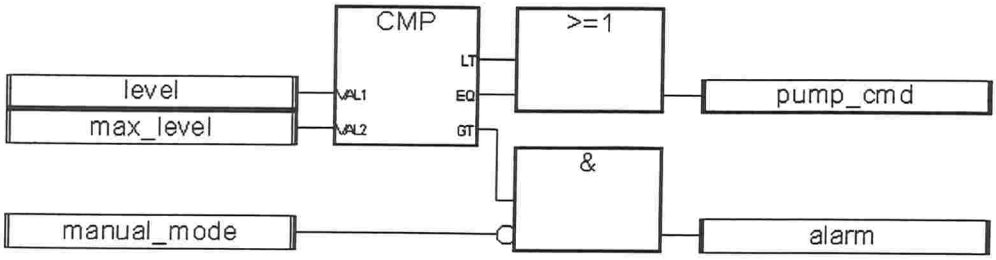
Apêndice B

FBD/Ladder/SFC

Exemplos das Linguagens Gráficas da IEC

FBD - Diagramas de Blocos de Função

Exemplo de programa usando FBD - bloco CMP (comparação) :



Equivalência em texto estruturado:

```
CMP1(level, max_level);
pump_cmd:= CMP1.LT OR CMP1.EQ;
alarm:= CMP1.GT AND NOT(manual_mode);
```

Equivalência em linguagem de instruções

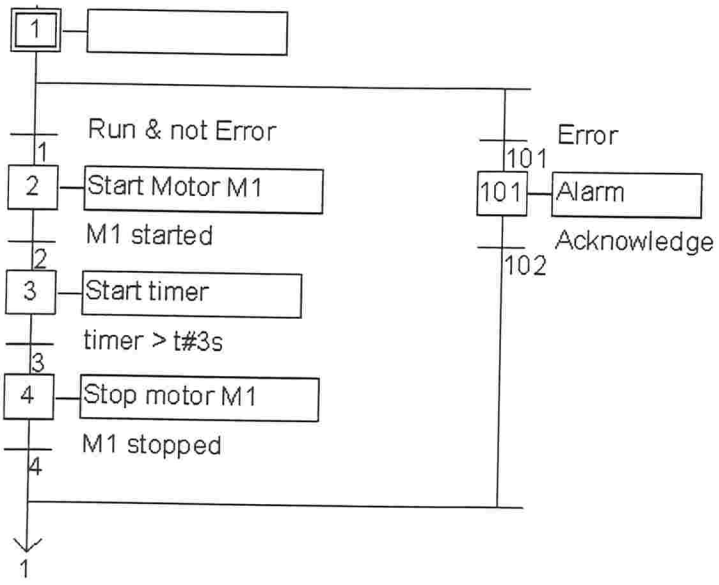
```
LD    level          ' carrega no "acumulador"
ST    CMP1.val1      ' joga em CMP1.val1
LD    max_level      ' idem para proximo
ST    CMP1.val2      ' parametro de entrada
CAL   CMP1           ' chama a funcao
LD    CMP1.LT        '
OR    CMP1.EQ        ' usa os parâmetros de saída no bloco or
ST    pump_cmd       ' joga para proximo bloco (uma variavel)
LD    CMP1.GT        '
ANDN  manual_mode    ' usa parametro de saida (no acumulador - ver linha anterior)
ST    alarm          ' joga numa variavel (segue o modelo).
```

SFC - Sequential Function Chart - Diagrama de Seqüência

Descreve lógica de alto nível (semelhante a Redes de Petri, GRAFCET, ou diagramas de estados).

Descreve a seqüência (ou o paralelismo) da execução de outros programas SFC, ou de programas nas outras 4 linguagens.

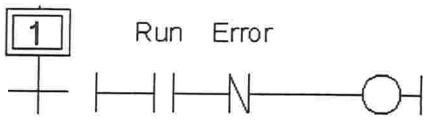
Por exemplo:



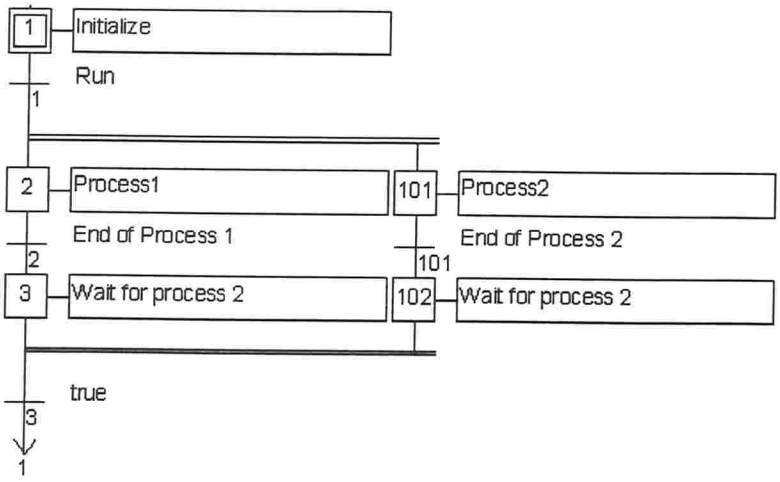
Cada quadrado é um passo (step), cada traço é uma transição. Pode-se associar uma chamada de função ou bloco, ou mesmo disparar um programa "filho" (descrito nas linguagens IEC, inclusive outro SFC) de acordo com uma entrada num "passo", ou na saída de um "passo".

O passo seguinte (ou próximo estado) só é executado quando a condição associada à transição que antecede o passo é verdadeira. A uma transição esta associado um valor booleano.

Se nenhum diagrama filho estiver presente, a condição é sempre TRUE. Por exemplo, na primeira transição, acima (1), no lugar de se usar texto estruturado (Run & not Error), poderia se usar ladder:

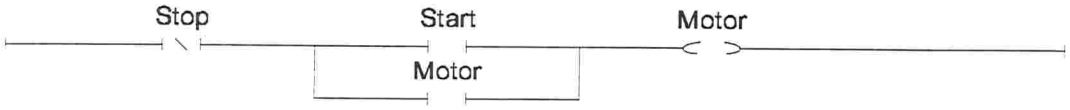


Usando-se divergências e convergências, podemos forcar execução concorrente (simultânea) de porções do modelo:



LD - Ladder Diagram - Diagrama de Escada

Exemplo:



Neste tipo de diagrama, adequado para representação de lógica de controle, grupos de contatos (Stop, Start, Motor, no exemplo) associados a solenóides (Motor, à esquerda), compões os *degraus*. Cada degrau é *avaliado* sequencialmente, de cima para baixo.

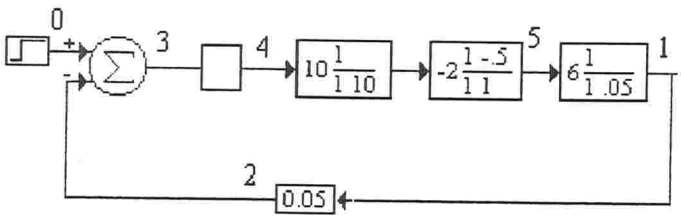
Blocos de funções podem ser associados a Ladder, desde que tenham uma entrada do tipo "habilita", ou uma saída do tipo lógica (verdadeiro ou falso), ou ambos.

◆

Apêndice C

Diagramas de Blocos - Modelo e Algoritmo

Várias considerações deste trabalho são realizadas sobre o seguinte modelo:



Neste modelo, a numeração dos blocos corresponde à ordem de execução, a partir do zero, de seus algoritmos correspondentes, para a realização de um resultado correto.

O número 5 compõe um único bloco, de 2ª ordem. O número 4 corresponde a um bloco de ganho, avaliado em função dos ganhos dos blocos 1 e 5.

O número do bloco, colocado à frente do mesmo, indica que o sinal correspondente à conexão com o(s) próximo(s) bloco(s) recebe o mesmo "nome" que o bloco, à saída do qual o sinal está relacionado. Esta "convenção" faz lembrar, ainda, que é responsabilidade do bloco "n" atualizar o sinal "n" (ver anexo II - definição dos pacotes de ASCII).

Chamamos *convenção* porque não existe restrição no protótipo que exija essa correspondência.

Este diagrama modela o funcionamento de uma turbina compensada, sem perturbação. A cada bloco corresponde um algoritmo. No caso dos ganhos, da constante e do somador, ele é evidente. Para os blocos dinâmicos, existem equações a diferença que os representam. Essas equações podem ser encontradas, por exemplo, pela aplicação de métodos de aproximação.

Para os blocos de 1ª e 2ª ordem do modelo acima, suas representações em forma de algoritmo (equação a diferença) são as seguintes:

Bloco 1:

$$\frac{120}{1+20S}$$

Para este bloco (T = 0,001), fazendo S = $(1-z^{-1})/T$, que corresponde à transformação retangular, temos uma equação em z, que é convertida em uma equação a diferença através da transformada Z inversa:

$$\frac{Y(z)}{U(Z)} = \frac{120T}{T+20(1-z^{-1})}$$

$$Y(z)[T+20(1-z^{-1})] = 120TU(Z)$$
$$YT+20Y-20z^{-1}Y = 120TU$$
$$y(n) = \frac{120T}{20+T}u(n) + \frac{20}{20+T}y(n-1)$$

Que resulta em:

y(n) = 0,00599985u(n) + 0,99995y(n-1)

Esta é uma forma de realizar a transformação. Neste trabalho, uma versão adaptada do SMO (produto da Reivax que possui simulador dinâmico) produz como resultado de um dado modelo a sequência em que os blocos devem ser executados e os parâmetros, já calculados, para cada bloco, segundo o método de integração escolhido e o passo de execução do modelo.

Essas informações serão então utilizadas para montar os pacotes ASCII que servirão de base para a construção dos modelos do protótipo.

Da mesma forma, se obtiveram os coeficientes do outro bloco dinâmico do modelo:

Bloco 2:

$$\frac{1}{1+0,1S}\,\frac{S-0,5}{1+S}$$

Esse bloco resulta, através de uma transformação retangular, numa equação na forma:

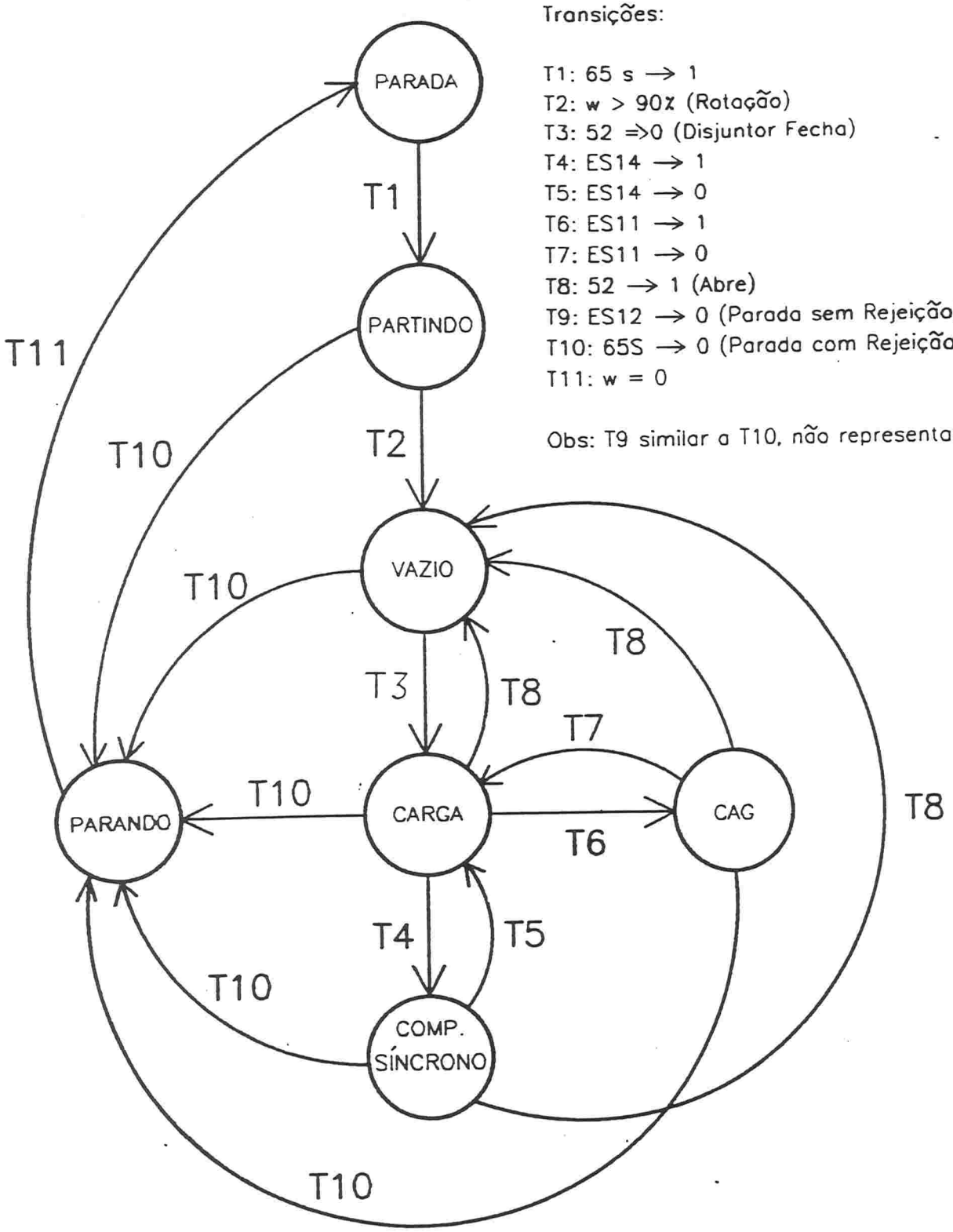
y(n) = 0,00988615u(n) - 0,0098911u(n-1) + 1,9891y(n-1) - 0,98911y(n-2)

Com os coeficientes acima calculados através do SMO.

◆

Anexo I

Diagrama de Estados - Regulação de Velocidade



Transições:

T1: 65 s → 1
T2: w > 90% (Rotação)
T3: 52 =>0 (Disjuntor Fecha)
T4: ES14 → 1
T5: ES14 → 0
T6: ES11 → 1
T7: ES11 → 0
T8: 52 → 1 (Abre)
T9: ES12 → 0 (Parada sem Rejeição)
T10: 65S → 0 (Parada com Rejeição)
T11: w = 0

Obs: T9 similar a T10, não representada

RIVAX AUTOMAÇÃO E CONTROLE	EQUIPAMENTO: REGULADOR DE VELOCIDADE RVX100				
	TITULO: DIAGRAMA DE TRANSIÇÕES DE ESTADO				
	PROJETADO: JMS	DESENHADO: RPM	APROVADO:	NUMERO:	ARQUIVO: C1DTED01
ESCALA: 3.97	DATA:	DATA: 04/01/93	DATA:	FOLHA:	REVISAO: 0

Anexo II

Pacotes definidos para o teste do protótipo

Significado	Valor
Número de diagramas	6
Nome do n-ésimo diagrama	diagrama1.txt
...	diagrama2.txt
...	diagrama3.txt
...	diagrama4.txt
...	diagrama5.txt
...	diagrama6.txt
Número de pares de conxão entre diagramas	3
Número do diagrama de origem 1	0
Número do sinal de origem 1	0
Número do diagrama destino 1	1
Número do bloco de transporte destino 1	1
etc	2
...	0
...	3
...	1
etc	4
...	0
...	5
...	1
Aqui começam as informações sobre as saídas:	
Número de sinais que serão registrados	6
Número do diagrama que contém sinal 1	0
Sinal de saída 1	1
etc	1
...	1
etc	2
...	1
etc	3
...	1
etc	4
...	1
etc	5
...	1
Aqui começam as informações sobre cada diagrama de	

blocos, ou seja, seus blocos e sinais:	
Número de blocos do diagrama1	6
Número de sinais do diagrama1	6
Passo de execução do diagrama1	0.1
Profundidade (memória) do sinal 0	3
Profundidade (memória) do sinal 1	3
Profundidade (memória) do sinal 2	3
Profundidade (memória) do sinal 3	3
Profundidade (memória) do sinal 4	3
Profundidade (memória) do sinal 5 (6 sinais)	3
Bloco tipo 6 (transporte) - bloco número 0	6
(esse diagrama "observa" seu par: diagrama 2)	
Numero de entradas (transporte tem ZERO)	0
Numero de saidas	1
Numero de parametros	0
Saída do bloco realizada no sinal número 0	0
Bloco tipo 4 (primeira ordem) - bloco número 1	4
Uma entrada	1
Uma saída	1
Três parâmetros	3
Entrada 0 é o sinal 5	5
Saída 0 é o sinal 1 (igual o número do bloco: 1)	1
Parâmetro 0 - coef. u(n) da eq. a diferença	0.00599985
Parâmetro 1 - coef. u(n-1) da eq. a dif.	0.0
Parâmetro 2 - coef. y(n-1) da eq. a dif.	0.99995
Bloco tipo 2 (ganho) - bloco 2	2
uma entrada	1
uma saída	1
um parâmetro	1
entrada é o sinal 1	1
saída é o sinal 2 (= número do bloco...)	2
parâmetro (ganho) = -0,05	-0.05
Bloco tipo 1 (somador) - bloco 3	1
duas entradas	2
uma saida	1
zero parametros	0
primeira entrada (0) é o sinal 0	0
segunda entrada (1) é o sinal 2	2

saída é o sinal 3	3
Tipo de bloco: 2 (ganho)	2
uma entrada	1
uma saída	1
um parâmetro	1
entrada é o sinal 3	3
saída é o sinal 4	4
parâmetro (ganho) de valor igual a -2	-2
Tipo de Bloco: 5 (sedunda ordem)	5
uma entrada	1
uma saída	1
cinco parâmetros	5
entrada é o sinal 4	4
saída é o sinal 5	5
parametro 0 - coef. de u(n)	0.00988615
parametro 1 - coef. de u(n-1)	-0.0098911
parametro 2 - coef. de u(n-2)	0.0
parametro 3 - coef. de y(n-1)	1.9891
parametro 4 - coef. de y(n-2)	-0.98911
(continua - diagrama 2, ...)	

Bibliografia

[Bonfatti 97] Bonfatti, Flavio; et alli. *IEC1131-3 Programming Methodology*. CJ Internacional, 1997.

[Brown 93] Brown, John Forrest. *Embedded Systems Programing in C and Assembly*. Van Nostrand Reinhold, New York, 1993

[Ellis 95] Ellis, Margaret A. & Carroll, Martin D. *Designing and Coding Reusable C++*. Addison-Wesley Publishing Company, Massachusetts, 1995.

[Ellis 93] Ellis, Margaret A. & Stroutrup, Bjarne. *C++: Manual de Referência Comentado*. Editora Campus, Rio de Janeiro, 1993.

[Fuchs 83] Fuchs, Rubens Dario; et alii. *Centrais Hidro e Termelétricas*. Edgar Blücher, São Paulo, 1983.

[Houpis 92] Houpis, Constantine H. & Lamont, Gary B. *Digital Control Systems*. MacGraw-Hill, Singapore, 1992.

[IEC1131-1] *IEC1131-1 International Standart*, International Eletrotechnical Comission, Genève, 1992.

[IEC1131-3] *IEC1131-3 International Standart*, International Eletrotechnical Comission, Genève, 1993.

[Kruglinski 96] Kruglinski, David J. *Inside Visual C++*. Microsoft Press, Washington, 1996.

[Labrosse 95] Labrosse , Jean J. *Embedded Systems Building Blocks*. R&D Publications, Kansas, 1995.

[Lee 97] Lee, Richard C. & Tepfenhart, Willian M. *UML and C++: a Practical Guide to Object-Oriented Development*. Prentice-Hall do Brasil, Rio de Janeiro, 1997.

[Martin 95] Martin, James & Odell, *James J. Análise e Projeto Orientados a Objeto*. Makron Books, São Paulo, 1995

[Pressman 95] Pressman, Roger S. *Engenharia de Software*. Makron Books, São Paulo, 1995.

[Rational] Rational Software Corporation. *Análise e Projeto Orientados a Objetos usando UML (Unified Modeling Language)*. - Apostila de Curso. Rational Software Corporation, 1997.

[Young 82] Young, Stephen J. *Real Time Languages*. Ellis Horwood, England, 1982