UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA QUÍMICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA QUÍMICA

Desenvolvimento de um Simulador Genérico de Processos Dinâmicos

DISSERTAÇÃO DE MESTRADO

Rafael de Pelegrini Soares

PORTO ALEGRE
2003

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA QUÍMICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA QUÍMICA

Desenvolvimento de um Simulador Genérico de Processos Dinâmicos

Rafael de Pelegrini Soares

Dissertação de Mestrado apresentada como requisito parcial para obtenção do título de Mestre em Engenharia

Área de concentração: Simulação de Processos

Orientador:
Prof. Dr. Argimiro Resende Secchi

Porto Alegre

2003

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ENGENHARIA QUÍMICA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA QUÍMICA

A Comissão Examinadora, abaixo assinada, aprova a Dissertação *Desenvolvimento de um Simulador Genérico de Processos Dinâmicos*, elaborada por Rafael de Pelegrini Soares, como requisito parcial para obtenção do Grau de Mestre em Engenharia.

Comissão Examinadora:

Profa. Dra. Marla Azário Lansarin

Prof. Dr. Jorge Otávio Trierweiler

Profa. Dra. Roberta Chasse Vieira

“Most ideas come from
previous ideas...

The best way to predict the future
is to invent it.” (Alan Kay)

Agradecimentos

Todos meus colegas do Departamento de Engenharia Química da UFRGS merecem sinceros agradecimentos, foram grandes companheiros de trabalho e acima de tudo amigos.

Agradeço ao Prof. Dr. Argimiro R. Secchi para o qual o adjetivo de orientador foi perfeito, trazendo sempre uma opinião de bom senso e visão do global sem esquecer dos detalhes.

Agradeço em especial à minha família que forneceu todo o suporte e apoio, meus grandes conselheiros que tornaram a execução deste trabalho muito mais fácil.

Nada mais justo que agradecer àqueles que tornaram este trabalho possível, nada mais injusto que não citar todos os nomes que, de alguma forma, contribuíram para este objetivo... a todos estes ficam minhas desculpas.

Resumo

Simulador de processos é uma ferramenta valiosa, pois possibilita desde a validação de projetos e sua operabilidade prática até aumentos de produção e redução de custos. Devido a estes e outros fatores, o interesse industrial em técnicas e pacotes computacionais para a modelagem, simulação e otimização de processos tem crescido muito nos últimos anos. Juntamente com este interesse cresce a qualidade das ferramentas disponíveis no mercado para tal, mas estas ainda não satisfazem totalmente as expectativas de seus usuários. Este trabalho consiste no projeto de um novo simulador genérico para processos dinâmicos que satisfaça os usuários de forma mais completa do que os disponíveis atualmente no mercado. Para tanto, foram reunidas e, quando necessário, desenvolvidas novas técnicas relativas à descrição, análise e solução de problemas dinâmicos. Uma nova linguagem de modelagem orientada a objetos e um sistema de tradução da representação nesta linguagem para sistemas de equações foram propostos. Métodos de análise dos sistemas de equações provenientes da modelagem foram desenvolvidos com o intuito de auxiliar o usuário na detecção de erros de modelagem. Algoritmos para solução de sistemas dinâmicos e estacionários foram reunidos e uma arquitetura interna foi proposta. Por fim, o sistema como um todo foi testado através de sua aplicação em problemas típicos e outros que não podem ser resolvidos diretamente com os pacotes computacionais disponíveis no mercado. O teste com os problemas práticos provou que a estrutura proposta é adequada e apresenta uma série de vantagens quando comparada com *softwares* largamente utilizados na simulação de processos.

Palavras chave: simulação de processos, sistemas DAE, CAPE-OPEN, diferenciação automática, diferenciação simbólica.

Abstract

Process simulator is a valuable tool, once it can be used in applications ranging from project validation, plant control and operability to production increasing and costs reduction. These facts among others have made the industrial interest in software packages for modeling, simulation and optimization to grow up. Together with this interest, grows up the quality of the simulation software tools, but these tools are not considered totally adequate for the users yet. This work consists in the design of a new general simulator for dynamic process aiming to satisfy the users in a more complete way. Methods regarding to the description, analysis and solution of dynamic problems were gathered and developed when necessary. A new object-oriented modeling language and a method for converting from the representation on this language to systems of equations where proposed. Methods of analysis for the systems of equations coming from the model description where proposed in order to detect modeling errors. Algorithms for the solution of dynamic and stationary systems were gathered and a internal architecture was proposed. At last, the whole proposed system was tested by its application to typical problems and other which can not be solved directly with the commercially available packages. These tests have proved that the proposed structure is suitable and leads to a system with several enhancements when compared with the widely used software for process simulation.

Key-word: process simulation, DAE systems, CAPE-OPEN, automatic differentiation, symbolic differentiation.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1 Introdução

Simulador de processos é uma ferramenta valiosa, pois possibilita desde a validação de projetos e sua operabilidade prática até aumentos de produção e redução de custos. Devido a estes e outros fatores, o interesse industrial em técnicas e pacotes computacionais para a modelagem, simulação e otimização de processos tem crescido muito nos últimos anos. Juntamente com este interesse cresce a qualidade das ferramentas disponíveis no mercado para tal, mas estas não satisfazem grande parte das expectativas de seus usuários.

1.1. Histórico

As ferramentas para simulação de processos, embora tenham se tornado mais populares apenas nos últimos cinco anos, já acumulam uma história de mais de 50 anos.

Nos anos 50 foram desenvolvidos os primeiros modelos para operações de unidades os quais eram executados em computadores de baixíssima capacidade (para os padrões atuais). Em 1958 a M. W. Kellog Corp. (Kesler e Kessler, 1958) apresentou o sistema Flexible Flow. Este sistema calculava de forma seqüencial os equipamentos, passando a saída de um como entrada de outro e iterando para a solução de processos que apresentavam reciclos. Esta metodologia tornou-se conhecida como modular seqüencial (*sequential modular*).

Na década de 60 houve um grande esforço no sentido do desenvolvimento de ferramentas do tipo modular seqüencial, com um grande número de empresas desenvolvendo seus próprios pacotes, chegando-se ao número de mais de 200 pacotes diferentes (Westerberg, 1998). Este desenvolvimento próprio despendia de grandes grupos tanto para desenvolvimento quanto para manutenção. De forma paralela, diversos pesquisadores acadêmicos desenvolviam conceitos e métodos para o que chamamos hoje de sistemas baseados em equações (*equation-oriented*). A idéia básica por trás destes sistemas é que cada modelo ou subsistema compartilha apenas as suas equações e não mais a sua solução. Desta forma, um gerenciador agrupa as equações de todas as unidades do processo em um único sistema de equações para então obter a solução de forma direta. Nesta década, muitos dos conceitos da arquitetura dos simuladores atuais foram desenvolvidos e muitos dos métodos aproximados, largamente utilizados até então, começaram a ser descartados.

Nos anos 70, métodos mais avançados para a decomposição e solução de processos pelo método modular foram propostos, gerando os conceitos de solução modular simultânea (*simultaneous modular flowsheeting*). Segundo este conceito, as unidades são tratadas exatamente como no método seqüencial modular, porém a solução do processo como um todo é executada de uma forma global simultânea e não mais em seqüência. Novos algoritmos e modelos mais gerais foram incorporados juntamente com métodos numéricos mais sofisticados. Este desenvolvimento também foi motivado pelo projeto ASPEN do Massachusetts Institute of Technology.

Nas décadas de 80 e 90, os sistemas baseados em equações tiveram um considerável desenvolvimento, especialmente para sua utilização em otimização com a utilização de algoritmos seqüenciais quadráticos (*sequential quadratic programming*, SQP). Além disto, a utilização de novos conceitos da engenharia de *software* levaram ao desenvolvimento de interfaces gráficas amigáveis e algoritmos ainda mais poderosos. Finalmente, o rápido avanço dos sistemas de *hardware* levou a uma facilidade de acesso aos computadores pessoais tornando muito mais ampla as possibilidades de utilização das ferramentas computacionais para simulação.

Atualmente as principais ferramentas para simulação de processos são desenvolvidas por empresas especializadas, as quais produzem ferramentas baseadas tanto no paradigma modular quanto no baseado em equações. Embora as ferramentas modulares ainda dominem o mercado, se observa um movimento na direção das ferramentas orientadas a equações.

Nos dias de hoje não é difícil se deparar com problemas altamente acoplados que envolvem de 10.000 a 100.000 equações e, freqüentemente, mais do que isto (Biegler *et al.*, 1997). Claramente, ferramentas computacionais avançadas que se utilizem do paradigma baseado em equações são necessárias para o tratamento destes problemas.

1.2. Motivação

Pesquisas dirigidas aos usuários, mostram que os pacotes computacionais para simulação disponíveis atualmente ainda não satisfazem totalmente suas necessidades (76% dos participantes de uma enquête referenciam as ferramentas atuais como não adequadas, Che-Comp, 2002). Esta insatisfação deve-se, principalmente, às limitações na solução de problemas complexos ou fora dos padrões por falta de flexibilidade e a dificuldade ou lentidão no aprendizado (Hlupic, 1999). Juntamente com estes motivos caminham: alto custo, lentidão da inclusão de novas técnicas, heterogeneidade entre os pacotes e baixa modularidade de modelagem e estrutural.

Este contexto leva a crer que há espaço para uma nova ferramenta que venha a suprir algumas destas deficiências.

1.3. Objetivos

Este trabalho tem como objetivo gerar o projeto de um novo simulador genérico para processos dinâmicos que satisfaça os usuários de forma mais completa do que os disponíveis atualmente no mercado.

Para que este objetivo maior seja alcançado, é necessário reunir, e quando necessário, desenvolver técnicas relativas à descrição, análise e solução de problemas dinâmicos. De forma mais direta, isto significa produzir:

- Uma nova *linguagem de modelagem* de fácil aprendizado e utilização, porém sem o detrimento da flexibilidade;

- Um sistema de tradução da representação de processos dinâmicos na linguagem proposta para sistemas de equações;

- Métodos de análise dos sistemas de equações provenientes da modelagem que auxiliem o usuário na detecção de erros de modelagem;
- Uma biblioteca de algoritmos para solução de sistemas dinâmicos e estacionários;
- Uma arquitetura interna modular para o *software*;
- Aplicações para problemas típicos e outros.

1.4. Estrutura da Dissertação

Esta dissertação está dividida em sete capítulos, arranjados da seguinte forma:

O Capítulo 1 (este capítulo) trata da introdução, justificativa e objetivos deste trabalho.

No Capítulo 2 é proposta uma estrutura de avaliação para *softwares* de simulação de processos baseada em trabalhos encontrados na literatura. Após, esta estrutura é aplicada a alguns pacotes comerciais disponíveis no mercado, evidenciando assim os pontos fortes e deficiências dos mesmos.

O Capítulo 3 discorre sobre a *linguagem de modelagem* proposta para o simulador em projeto. Para tanto, primeiramente é feito um estudo sobre algumas linguagens utilizadas para modelagem e solução de processos dinâmicos.

No Capítulo 4 são apresentados a estrutura interna do simulador e os diagramas de classes conforme a UML (*Unified Modeling Language*) para a implementação da arquitetura proposta.

No Capítulo 5 consideram-se os problemas relativos à análise prévia e solução numérica dos sistemas de equações provenientes da modelagem de processos dinâmicos.

Diversos problemas típicos e outros que não podem ser resolvidos diretamente pelos pacotes de simulação disponíveis atualmente são tratados no Capítulo 6, visando validar a linguagem, estrutura e métodos numéricos reunidos e desenvolvidos neste trabalho.

No Capítulo 7 são apresentadas as conclusões e delineadas as perspectivas futuras para o *software* projetado neste trabalho.

Capítulo 2 Avaliação de Simuladores

Desenvolver uma nova ferramenta em um contexto no qual existe um parcial descontentamento dos usuários com as ferramentas disponíveis não é uma tarefa fácil. A construção de um sistema apenas semelhante aos existentes estará fadada ao fracasso, pois os concorrentes já estão estabelecidos. Por este motivo, uma análise detalhada dos *softwares* existentes para a detecção dos pontos fortes e falhos é crucial.

Neste capítulo é proposta uma estrutura de avaliação de pacotes computacionais para simulação de processos dinâmicos. Esta estrutura é baseada em trabalhos semelhantes encontrados na literatura, os quais propõem estruturas hierárquicas de avaliação. Finalmente, a estrutura de avaliação desenvolvida é aplicada a alguns *softwares* comerciais.

2.1. Critérios de Avaliação

Uma comparação entre diferentes pacotes de simulação ou simples avaliação de um *software* em particular é freqüentemente uma tarefa custosa e que consome tempo, o que torna clara a necessidade da utilização de uma metodologia de avaliação.

As técnicas normalmente utilizadas para uma comparação entre produtos em geral se resumem em um problema de identificação de critérios e sua estruturação. A literatura apresenta algumas metodologias para avaliação de *softwares* (Hlupic e Paul, 1995 e Nikoukaran *et al.*, 1999). Tais metodologias são apresentadas na forma de estruturas hierárquicas, iniciando-se de um ponto geral embora impreciso, o qual vai sendo gradualmente refinado em sub-critérios cada vez mais precisos. Como regra geral uma boa estrutura de avaliação deve assegurar uma caracterização completa do produto em questão, porém livre de redundâncias (Hulpic e Paul, 1995).

Alguns trabalhos relacionados à avaliação e seleção de *softwares* para simulação no ramo da produção podem ser encontrados na literatura (Hlulpic *et al.*, 1995 e Nikoukaran *et al.*, 1999). Tais publicações têm como principal objetivo a seleção do melhor produto entre um grupo de semelhantes. Esta seleção é feita de acordo com a futura aplicação do *software*, através da utilização de uma metodologia de avaliação. O presente trabalho difere dos citados por fazer uma avaliação dos *softwares* existentes visando o desenvolvimento de um novo produto, e por este motivo objetivando elucidar quais são as características bem implementadas e os pontos falhos que os pacotes apresentam. Devido a esta discrepância de objetivos e de área de aplicação (simulação de processos em geral e não apenas da área de produção), uma utilização direta de tais estruturas de avaliação não forneceria o resultado esperado.

Recomendações sobre o uso e seleção de *softwares* para simulação podem ser encontradas em livros de simulação (Law e Kelton, 1991 e Banks *et al.*, 1996) ou publicações de caráter introdutório (Gogg e Mott, 1993, Pidd, 1994 e Seila, 1995). Estes autores apresentam alguns comentários ou pequenas listas de critérios de avaliação de *softwares* de simulação, como apresentado em Marquardt (1995), mas não seguem alguma técnica ou metodologia de avaliação em específico e apresentam uma grande heterogeneidade de termos. Este fato dificulta a comparação direta e/ou compilação destes trabalhos.

Na Figura 2.1 estão demonstrados os grupos principais da estrutura de avaliação para *softwares* de simulação proposta neste trabalho. Esta estrutura é baseada no trabalho de Nikoukaran *et al.* (1999). Os blocos que aparecem em branco não sofreram alteração alguma, ao grupo *Modelagem e Entrada de Dados* foram feitas algumas modificações e o grupo de critérios *Ferramentas de Simulação* foi inserido na estrutura. Um maior detalhamento das modificações propostas é apresentado nas seções 2.1.1 e 2.1.2. Uma melhor explanação sobre os grupos inalterados pode ser encontrada no trabalho original.

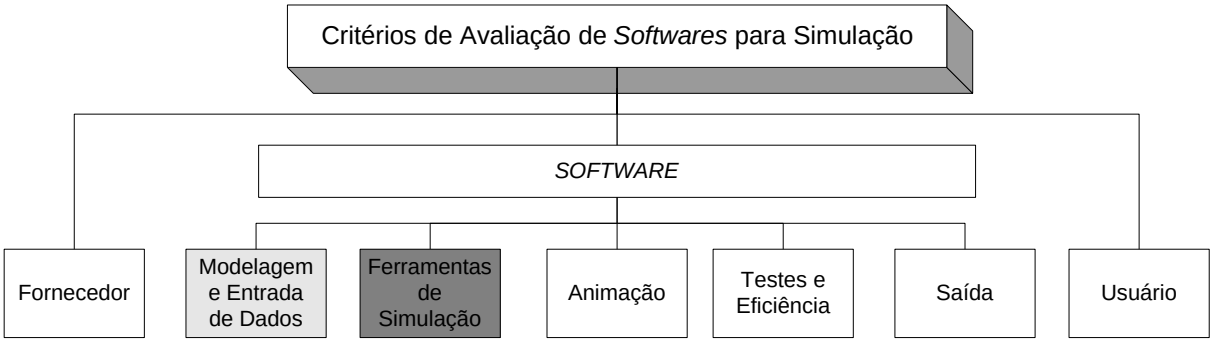


Figura 2.1 Grupos de critérios principais para avaliação de *softwares* de simulação propostos por Nikoukaran *et al.* (1999), com as modificações em destaque.

2.1.1. Modelagem e Entrada de Dados

Esta categoria de critérios inclui características relacionadas à modelagem do processo em estudo, desde a sua forma de descrição passando pelo seu desenvolvimento e tratamento interno. A representação gráfica deste grupo de critérios aparece na Figura 2.2. Esta estrutura é baseada no trabalho de Nikoukaran *et al.* (1999), com modificações visando atualização e adaptação aos objetivos deste trabalho. Uma breve explanação de seus subcritérios é apresentada a seguir.

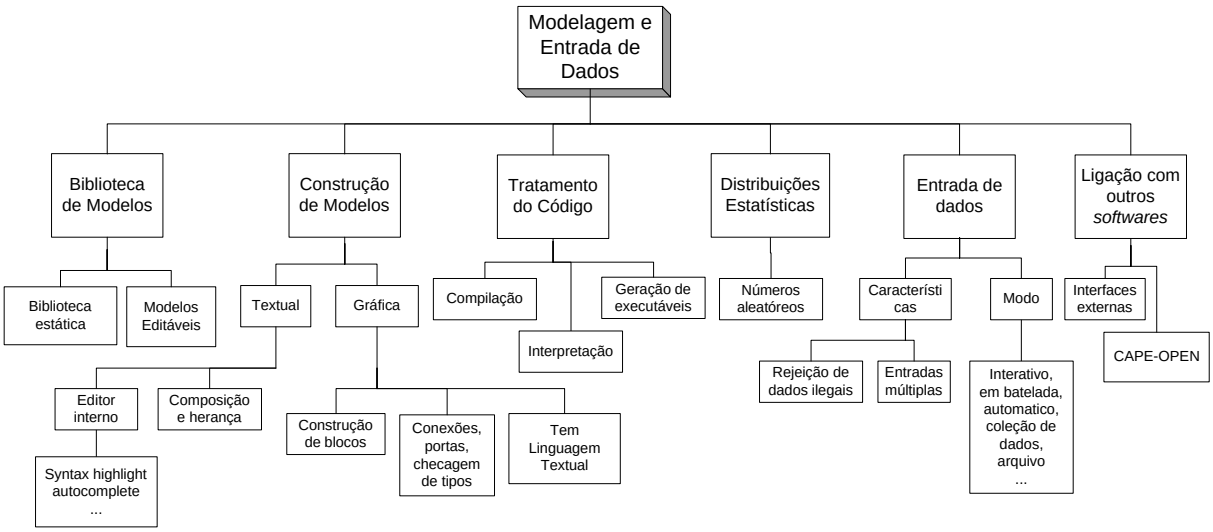


Figura 2.2 Subgrupo *Modelagem e Entrada de Dados* da estrutura de avaliação proposta e todas suas ramificações.

Biblioteca de Modelos:

Os pacotes de simulação podem prover facilidades relacionadas com a disponibilização e/ou criação de uma biblioteca de modelos, visando facilitar a modelagem de processos como um todo. Quando há disponibilidade de uma biblioteca de modelos, esta pode ser uma bibliotéca estática ou editável. Uma biblioteca estática é de grande valia quando o processo em estudo se enquadra nos modelos disponíveis. Já a possibilidade da edição de modelos e posterior agrupamento dos mesmos em uma biblioteca provê muito mais poder ao usuário.

Construção de Modelos:

Neste subgrupo são consideradas as características relacionadas com o desenvolvimento de novos modelos.

Um novo modelo é sempre construído através de algum formalismo, chamado de *linguagem de modelagem*, este formalismo pode ser gráfico ou textual.

Em uma linguagem textual, modularidade de desenvolvimento é uma característica muito importante, pois possibilita o desenvolvimento de forma independente de pequenas partes de código para uma posterior junção. Modularidade, em linguagens textuais, é mais eficientemente obtida através da utilização do paradigma orientado a objetos (*object-oriented paradigm*). Os conceitos deste paradigma, desejáveis na modelagem de processos, são composição (*composition*) e herança (*inheritance*). Onde, composição é a possibilidade da criação de um modelo que contenha um ou mais modelos internos e herança é a possibilidade da utilização de um modelo previamente escrito como base de um novo modelo. Quando é utilizado o conceito de herança, o novo modelo herda todas as características do modelo utilizado como base, possibilitando a adição de particularidades ao novo modelo que está sendo criado. No caso de descrição textual, o *software* pode disponibilizar um editor próprio ou fazer uso de um editor externo, tal editor pode fornecer facilidades como sintaxe destacada por coloração do código (*syntax highlight*) ou preenchimento automático de código (*autocomplete*).

No caso de construção de modelos de forma gráfica, modularidade também é uma característica importate. Na modelagem gráfica, modularidade é obtida pela possibilidade da criação de blocos e a conexão entre os mesmos, além da possibilidade de modelagem hierárquica. Onde, modelagem hierárquica significa a possibilidade do encapsulamento de um grupo de blocos para a formação de um bloco maior. Na modelagem gráfica a interface com o usuário é uma característica importante, podem existir facilidades tais como: *drag-and-drop*, caixas de diálogo, etc. Quando o software apresenta contrução de modelos de forma gráfica, a presença da possibilidade de uma descrição também através de código textual fornece um grande grau de flexibilidade.

Tratamento do Código:

A descrição de um sistema em um simulador de processos, seja gráfica ou textual, certamente necessitará de algum tipo de tratamento prévio que possibilite sua execução, e este é o assuntodo deste grupo de critérios. Este tratamento poderá ser a tradução para outra linguagem de nível mais baixo (ver níveis de linguagens Seção 3.1) e posterior compilação e *link* edição, ou algum método de interpretação sem a criação de arquivos intermediários ou utilização de um compilador. Pode ser feita também a conversão do modelo para um arquivo executável que não dependa mais do pacote de simulação, o que é uma característica interessante, principalmente para empresas que vendem soluções de engenharia a terceiros.

Estatística:

O *software* pode prover algumas distribuições estatísticas padrões, tais como normal, exponencial e retangular, assim como a possibilidade da definição de novas distribuições. Geradores de números aleatórios também podem estar presentes, assim como facilidades estatísticas em geral.

Entrada de Dados:

Simulações normalmente envolvem parâmetros ou necessitam do conhecimento do valor de algumas variáveis de entrada. Nestes casos é necessário que o usuário forneça de alguma forma estes dados. Esta entrada de dados pode ser feita de forma interativa, através de arquivos de dados ou coletada automaticamente de algum sistema externo. A rejeição de dados fora da sua faixa de validade ou de alguma forma ilegais pode prevenir grande parte dos erros que poderão ocorrer durante a simulação.

Ligação com outros Softwares:

Freqüentemente é necessária a utilização de rotinas ou mesmo pacotes externos na modelagem de um processo, exemplos típicos são o cálculo de propriedades termodinâmicas ou de correlações empíricas. Para facilitar a utilização da ferramenta para estes casos, a própria linguagem de modelagem pode prover facilidades na descrição de tais ligações com *softwares* externos. Estas ligações podem se dar por interfaces próprias ou utilizando o padrão CAPE-OPEN (CO, 2002).

2.1.2. Ferramentas de Simulação

Este grupo de critérios não faz parte da estrutura de avaliação proposta por Nikoukaran *et al.* (1999), e é proposto neste trabalho vislumbrando a avaliação das ferramentas numéricas disponíveis para solução, análise e otimização de processos em geral. Além disto, há critérios para classificação dos tipos de problemas que são passíveis de solução pelo pacote em avaliação. Uma representação gráfica deste grupo de critérios pode ser observada na Figura 2.3 e uma explanação sucinta dos critérios segue abaixo.

Processamento:

A solução dos problemas de simulação pode ser conduzida de forma seqüencial ou paralela. Quando há possibilidade de processamento paralelo, este pode ser feito por aplicativos pertencentes ao pacote, ou de forma heterogênea (aplicativos de terceiros). Uma forma de processamento paralelo ou computação distribuída é a utilização de interfaces de comunicação, como as propostas no projeto CAPE-OPEN (CO, 2000).

Estratégia:

Modularidade é uma característica importante na etapa de modelagem, como já discutido na seção 2.1.1. Mas a solução de um problema composto por vários subproblemas pode ser dada pela criação de um único problema formado pela junção de todos os subproblemas (solução direta), ou pela solução de cada subproblema de forma independente (solução modular).

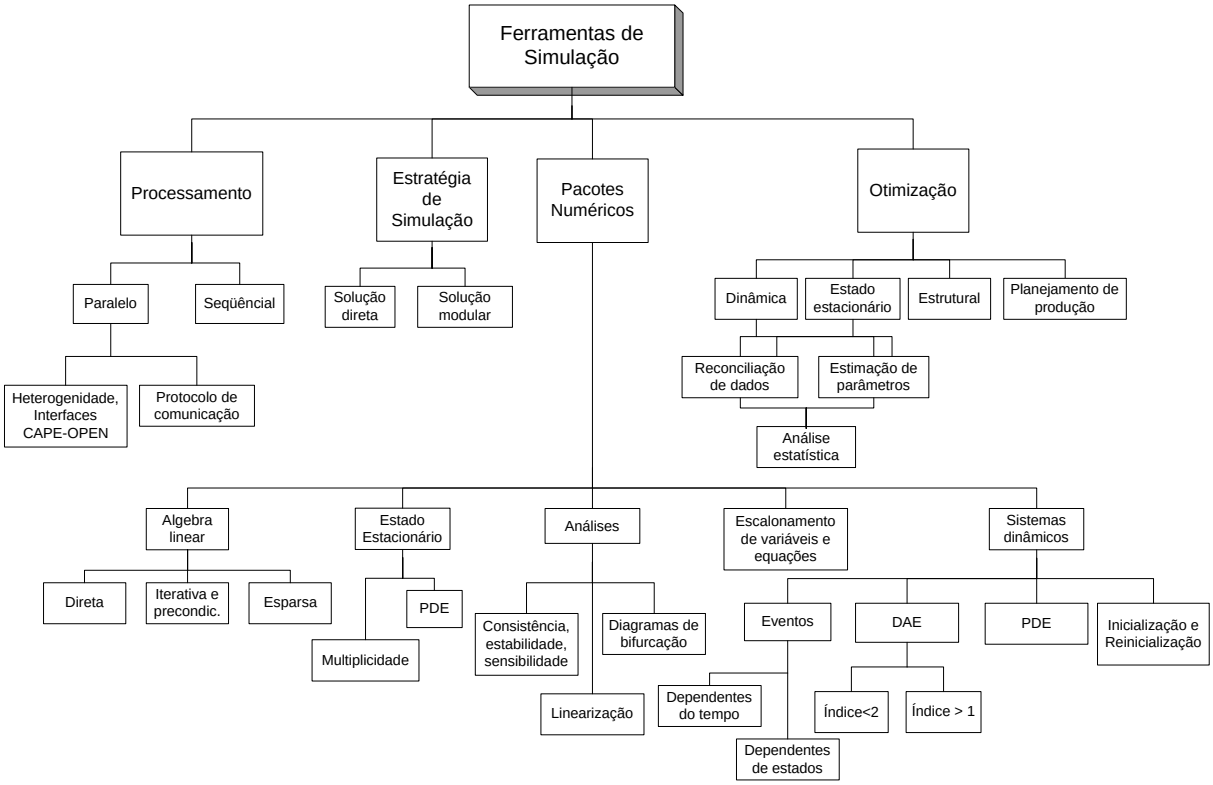


Figura 2.3 Subgrupo *Ferramentas de Simulação* da estrutura de avaliação proposta neste trabalho e todas suas ramificações.

Pacotes Numéricos:

Os pacotes numéricos contidos em um *software* de simulação formam o seu cerne, e deles depende grande parte da reputação do pacote de simulação como um todo.

Na solução de sistemas não-lineares, assim como na solução de sistemas algébrico-diferenciais, surgem problemas de álgebra linear. Estes problemas podem ser resolvidos de forma direta ou iterativa, no segundo caso pode-se fazer uso de pré-condicionadores, visando um aumento da taxa de convergência. A utilização de álgebra esparsa torna-se muito importante para sistemas de grande dimensão.

Para reduzir problemas de estabilidade numérica pode-se fazer um escalonamento das variáveis e/ou equações do sistema.

Em se tratando de soluções estacionárias, podem estar disponívies ferramentas para detecção da multiplicidade de soluções. Para soluções dinâmicas, pode-se fazer uma classificação da capacidade do integrador pelo índice diferencial dos problemas que o mesmo é capaz de resolver: sistemas de equações diferenciais ordinárias (índice 0), sistemas de equações algébrico-diferenciais (*Differential-Algebraic Equations*, DAE) até índice 1 ou problemas de índice elevado (índice maior que 1).

Durante uma simulação dinâmica podem ocorrer vários eventos que são de interesse do usuário. Estes eventos podem ser dependentes apenas da variável independente (normalmente o tempo), ou dependentes de algum dos estados do sistema, como por exemplo, o volume ou temperatura de um tanque. Há também pacotes numéricos capazes de resolver problemas de equações diferenciais parciais (*Partial Differential Equations*, PDE) tanto dinâmicos quanto estacionários.

Antes da solução numérica para simulação, pode-se fazer uso de análises de consistência (resolubilidade, condições iniciais, unidades de medida, etc.). Um simulador pode fornecer também uma série de tipos diferentes de análises, tais como: análise de estabilidade, sensibilidade, etc. Diagramas de bifurcação e a linearização do sistema também são de grande utilidade.

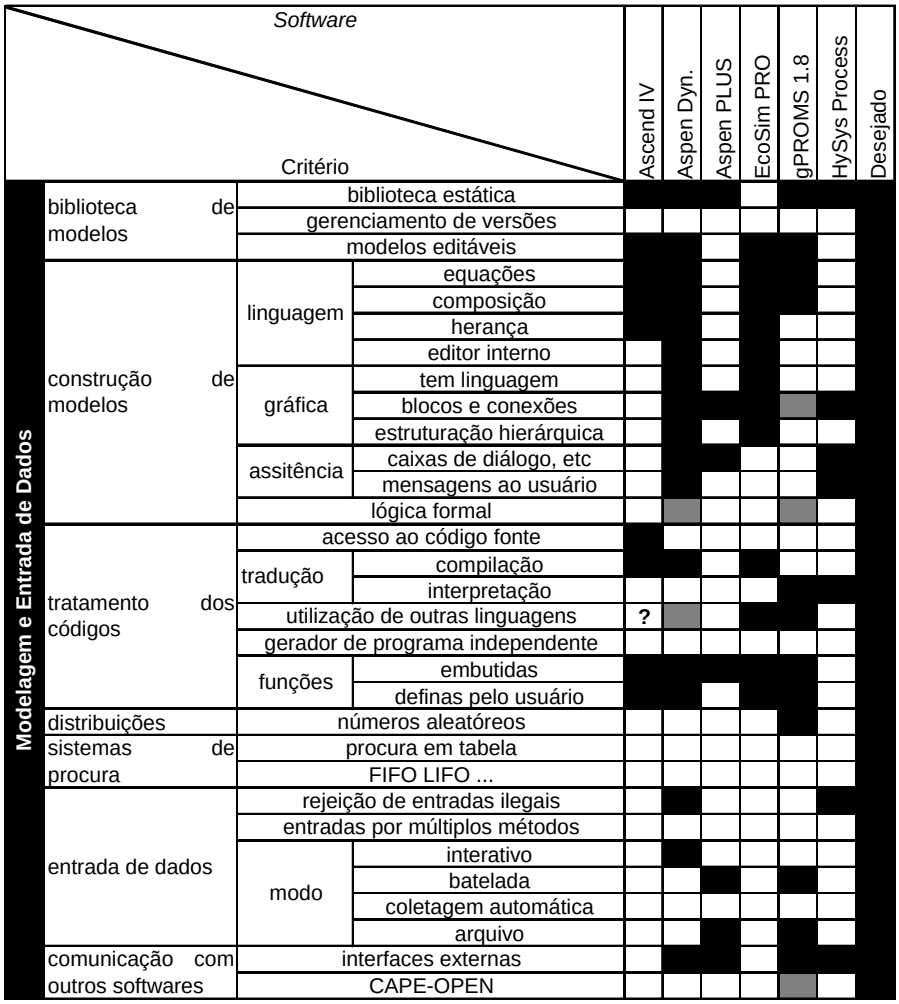
Otimização:

A otimização é um dos motivos da existência da simulação, e ela pode ser utilizada em problemas estacionários, dinâmicos, estruturais ou de planejamento. Pode haver facilidades para a solução de problemas de estimação de parâmetros e de reconciliação de dados, que não deixam de ser problemas de otimização dinâmica ou estacionária, dependendo do caso de aplicação. No caso destes problemas pode-se haver uma análise estatística das suas soluções para uma melhor avaliação dos resultados.

2.2. Aplicação da Estrutura de Avaliação

A estrutura de avaliação apresentada na Figura 2.1 é bastante ampla e trata aspectos relacionados com vendas, suporte técnico, relação usuário-provedor entre outros.Claramente, a avaliação quanto a todos estes aspectos não cabe ao escopo deste trabalho.

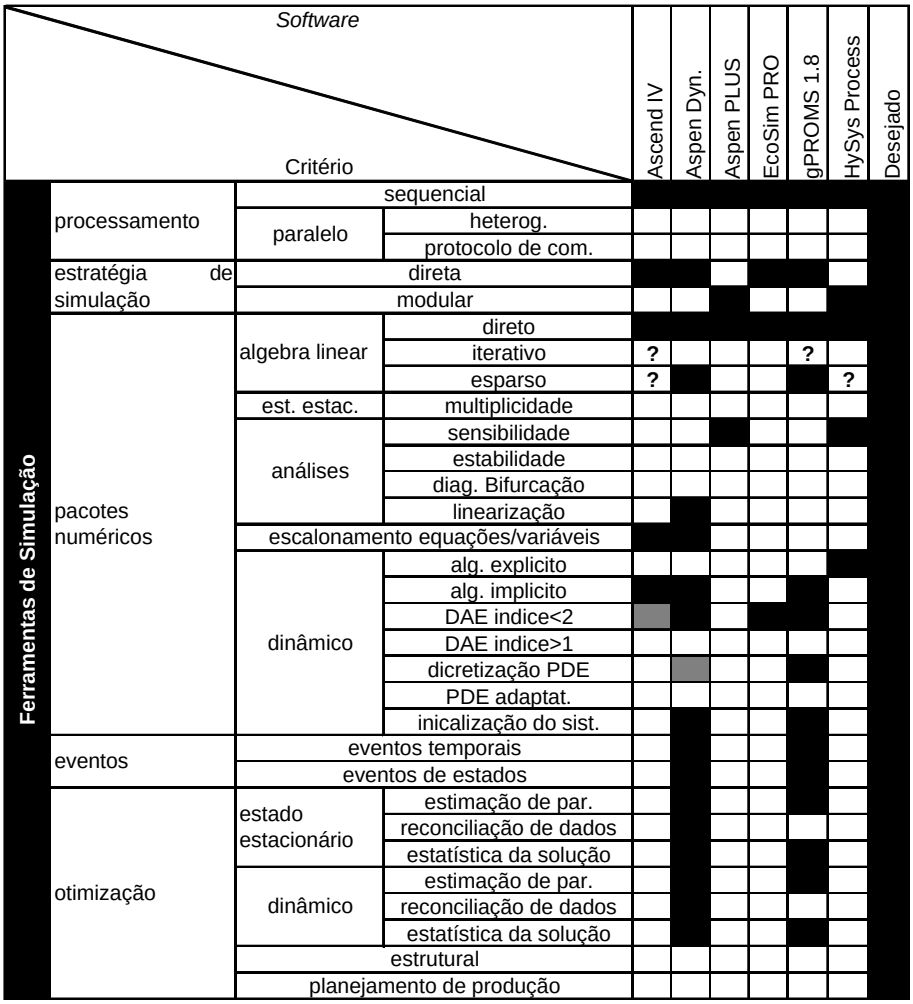
Nesta seção são apresentadas as avaliações de alguns *softwares* comerciais quanto aos grupos de critérios *Modelagem e Entrada de Dados* e *Ferramentas de Simulação*, demonstrados nas Figuras 2.4 e 2.5, respectivamente. Nestas figuras, juntamente com a avaliação dos *softwares* comerciais, são apresentados quais os critérios que se deseja obter com o simulador que venha a ser implementado com base neste trabalho.



contempla o critério totalmente; parcialmente; ? não se sabe ao certo; não contempla.

Figura 2.4 Avaliação de *softwares* comerciais, quanto ao grupo de critérios *Modelagem e Entrada de Dados.*

A última coluna da Figura 2.4 e Figura 2.5 corresponde aos critérios considerados, neste trabalho, como ideais. O projeto de uma base sólida, que possibilite a contemplação destes critérios, foi perseguido neste trabalho, conforme apresentado nos próximos capítulos.



contempla o critério totalmente; parcialmente; ? não se sabe ao certo; não contempla.

Figura 2.5 Avaliação de *softwares* comerciais, quanto ao grupo de critérios *Ferramentas de Simulação*.

Pode-se observar que nenhum dos pacotes avaliados contempla um bom número de critérios quando comparado ao desejado.

A observação das avaliações também permite concluir que existem duas diferentes *categorias* de simuladores:

- Sistemas de Modelagem de Processos (*Process Modeling Systems*): pacotes que primam por facilidades relativas à facilidade de utilização através de interfaces gráficas e biblioteca estática de modelos. Exemplos que poderiam ser citados são: Aspen PLUS (AspenTech, 2002) e HySys Process (HyProtech, 2002);

- Sistemas de Modelagem Genéricos (*General Modeling Systems*): pacotes que primam pela utilização de uma linguagem de desenvolvimento flexível juntamente com um bom conjunto de pacotes numéricos. O principal exemplo desta categoria é o software gPROMS (Oh e Pantelides, 1996).

2.3. Conclusão

Neste capítulo, uma estrutura hierárquica para avaliação de pacotes computacionais para simulação de processos dinâmicos, baseada em trabalhos encontrados na literatura, foi proposta. A principal diferença entre a estrutura proposta e as encontradas na literatura é a sua aplicação. Uma vez que os trabalhos encontrados tratam da escolha do melhor *software* para uma determinada aplicação e neste capítulo procurou-se detectar quais são os pontos fortes e falhos de um conjunto de programas computacionais.

A estrutura de avaliação proposta reúne, de forma hierárquica, as características mais relevantes de um simulador de processos dinâmicos. E sua aplicação permitiu detectar quais são as principais deficiências de alguns dos simuladores mais populares. Além disto, a estruturação de critérios, apresentada neste capítulo para fins de avaliação, também serviu como um conjunto de diretrizes a serem alcançadas neste trabalho.

Capítulo 3 Linguagem

Até a década de 80 a modelagem de sistemas dinâmicos era feita exclusivamente pela representação do sistema em estudo na forma de sistemas de equações diferenciais ordinárias. A partir da década de 90 a prática da utilização direta de sistemas algébrico-diferenciais, os quais surgem naturalmente na modelagem de problemas físicos assim como os de engenharia química, começou a tornar-se comum. Atualmente, a modelagem de sistemas dinâmicos tende a aproximar-se cada vez mais da própria descrição matemática do problema e esta é a linha adotada neste trabalho para a proposta de uma nova *linguagem de modelagem*.

É sabido que a reputação de um pacote computacional está diretamente ligada à forma com que o usuário se comunica com o mesmo, portanto o desenvolvimento de uma boa *linguagem de modelagem* é crucial para o sucesso de um *software* para simulação de processos. Neste capítulo, primeiramente desenvolve-se um estudo sobre algumas linguagens passíveis de utilização para simulação de processos dinâmicos. Com base neste estudo é concebida uma nova linguagem orientada a objetos para modelagem de processos dinâmicos.

3.1. Linguagens de Modelagem para Simulação

A tendência atual no campo da simulação é fazer um uso intensivo de interpretadores de código, os quais tornam possível a simulação de um sistema a partir de uma descrição de fácil compreensão. Tais ferramentas são capazes de aumentar bastante a produtividade do usuário. Muitos dos simuladores comerciais dispõem destas facilidades, que variam desde interfaces simples de tradução de texto até sofisticados sistemas gráficos de desenvolvimento.

Nestas ferramentas o modelo é descrito formalmente via texto ou gráfico, e esta descrição será denominada de *linguagem de modelagem*. Esta definição utiliza-se de uma extensão do significado da palavra linguagem, sendo aplicada para qualquer tipo de representação de um modelo, incluindo as representações gráficas (Lorenz, 1999).

As linguagens passíveis de utilização para a descrição de processos dinâmicos podem ser classificadas quanto a vários aspectos, entre eles ressalta-se o nível da linguagem. É possível definir diversos níveis de linguagens para simulação, embora estes não apresentem divisões muito claras, podendo praticamente ser considerados como um contínuo. O trabalho de Lorenz (1999) propõe uma divisão em quatro níveis, a aplicação desta classificação para algumas das linguagens utilizadas para modelagem e simulação de processos é apresentada na Figura 3.1.

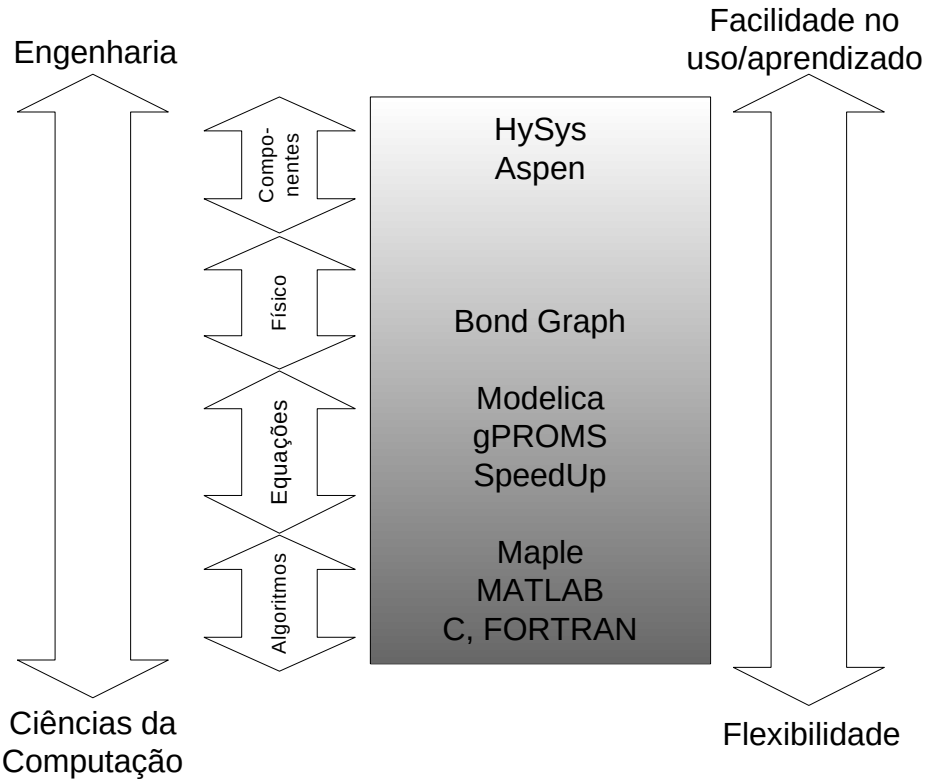


Figura 3.1 Classificação de algumas linguagens em níveis de abstração.

Certamente existem outros níveis de linguagens, mas estas não são comumente utilizadas para fins de simulação de processos. ASSEMBLY é um exemplo de linguagem que se encontra em um nível de abstração inferior ao FORTRAN. Abaixo segue uma breve descrição da caracterização em níveis utilizada neste trabalho.

3.1.1. Nível de Algoritmo

Uma linguagem deste nível é baseada em procedimentos ou algoritmos, e faz uso direto de entidades do computador, como por exemplo, a memória do computador. Exemplos típicos são as linguagens de programação como FORTRAN e C, que são puramente atribuitivas e não conseguem expressar diretamente algumas entidades do problema a ser resolvido, tais como equações de igualdade. Por este motivo uma linguagem pertencente a este grupo também pode ser chamada de atribuitiva ou de procedimentos.

3.1.2. Nível de Equações

Neste grupo encontram-se linguagens de um nível de abstração um pouco mais elevado. Tratam diretamente com conceitos matemáticos: variáveis (no sentido matemático) e equações (expressões de igualdade e não de atribuição). Permitindo assim a modelagem de um sistema através da sua representação na forma de conjunto de variáveis e equações. Podem ser citados como exemplos as linguagens utilizadas nos pacotes gPROMS (Oh e Pantelides, 1996) e Aspen Custom Modeler (AspenTech, 2002).

A diferença entre os níveis de algoritmo e de equações pode ser vislumbrada mais claramente através de exemplos. Considere-se a descrição matemática de um resistor:

$$V_1 - V_2 = i \cdot R \tag{3.1}$$

onde, V_1 e V_2 são os potenciais nos terminais do resistor, i é a corrente fluindo através do mesmo e R é a sua resistência.

Em um sistema baseado em equações a expressão de igualdade (3.1) pode ser escrita diretamente, independentemente de qual será a variável que se deseja determinar. Entretanto, em um sistema baseado em algoritmos, a descrição do problema na linguagem depende da variável que se deseja, uma vez que apenas operações de atribuição são permitidas. Por exemplo:

$$\begin{aligned} i &= (V_1 - V_2) / R \\ V_2 &= V_1 - i \cdot R \\ V_1 &= V_2 + i \cdot R \end{aligned} \tag{3.2}$$

devem ser utilizadas para a determinação de i, V_2 e V_1, respectivamente.

Outro exemplo pode ser a equação de conservação de massa de um sistema com uma alimentação F, saídas L e V e massa total M.

Em uma linguagem baseada em equações, este sistema pode ser representado diretamente como demonstrado em (3.3). Já em um sistema do tipo atribuitivo, com formulação implícita do sistema, seria necessário utilizar algo semelhante a (3.4), juntamente com algum algoritmo que anule (ou minimize) o resíduo *res* da igualdade original, para que a mesma seja satisfeita.

$$\frac{dM}{dt} = F - (V + L) \tag{3.3}$$

$$res = F - (V + L) - M' \tag{3.4}$$

Deve-se enfatizar que nesta seção estão sendo tratados aspectos relativos aos níveis de linguagens de descrição, embora internamente todos os sistemas recaiam no nível de algoritmos.

3.1.3. Nível Físico

Logo acima dos sistemas baseados em equações estão as linguagens baseadas em conceitos físicos, onde os elementos básicos são os fenômenos físicos.

O *Bond Graph* (Borutzky, 1992 e Broenink, 1997) é um exemplo típico de tal linguagem. Este sistema é baseado em duas considerações gerais: particionamento do processo sendo modelado em subsistemas os quais interagem através da troca de energia entre os mesmos e cada fluxo de energia entre os subsistemas ou elementos é associado a dois fluxos com sinais opostos os quais são as variáveis do problema.

3.1.4. Nível de Componentes

Na posição de mais alto nível estão as linguagens baseadas em componentes. Aqui a modelagem dos sistemas é feita simplesmente pela utilização, configuração e conexão de componentes pré-existentes. Estas são algumas das características do paradigma orientado a objetos, o qual é tratado mais especificamente na Seção 3.2.

Este tipo de linguagem tem como características a extrema facilidade de aprendizado e a rapidez na modelagem, em detrimento da flexibilidade. São exemplos típicos deste grupo pacotes como *HySys* e *ASPEN PLUS*.

3.1.5. Comparação entre os Níveis

Como comentado anteriormente não há rigidez nas fronteiras entre cada um dos níveis de linguagens aqui expostos e algumas linguagens podem contemplar, mesmo que parcialmente, mais de um nível. Sem dúvida não há como caracterizar um nível como melhor que outro, apenas pode-se dizer que há níveis de linguagem mais adequados a cada situação de utilização.

Deve-se notar que, como a Figura 3.1 sugere, uma linguagem de mais alto nível normalmente apresenta maiores facilidades no seu aprendizado e utilização, mas impõe restrições de flexibilidade ao usuário. Se este for o caso, é aconselhável a utilização de uma linguagem de mais alto nível possível, desde que esta proporcione a flexibilidade necessária à sua aplicação.

3.2. Programação Orientada a Objetos

A programação orientada a objetos (*object-oriented programming*, OOP) é parte do movimento em direção da utilização do computador como um meio de expressão ao invés de apenas máquinas de processamento de dados (Eckel, 2000).

Todos os tipos de linguagens de programação fornecem algum nível de abstração (Seção 3.1). A linguagem ASSEMBLY é uma pequena abstração da linguagem da máquina. Linguagens como FORTRAN, C e Basic são abstrações do ASSEMBLY. Essas linguagens já apresentam grandes avanços, mas seus níveis de abstração ainda estão relacionados com a estrutura do computador e não com a estrutura do problema que se quer resolver. A OOP está um passo a diante destas linguagens e fornece ferramentas ao programador que o tornam capaz de representar diretamente o problema em questão. Tudo se baseia na permissão ao programador da criação de variáveis de novos tipos (novas classes de objetos). Assim quando se lê o código descrevendo a solução de um problema, lê-se também a descrição do próprio problema. Isto é possível porque em OOP se descreve um problema nos termos do próprio problema e não nos termos da máquina onde a solução será obtida.

Alan Kay (1993), um dos criadores do Smalltalk (a primeira linguagem orientada a objetos a ter sucesso, e uma das linguagens sobre as quais o C++ é baseado) reuniu as principais características da OOP em cinco itens:

- **Tudo é um objeto.** Pode-se pensar em um objeto como um tipo de variável, ele guarda dados, mas pode-se fazer “pedidos” a este objeto, pedindo a ele que faça operações em si mesmo, ou em seus argumentos.

- **Um programa é uma reunião de objetos dizendo uns aos outros o que fazer, através do envio de mensagens.**

- **Cada objeto tem seu próprio espaço de memória que é construído por outros objetos (composição).** Pode-se criar um novo tipo de objeto através da união de outros objetos. Assim, pode-se "esconder" a complexidade de um programa atrás da simplicidade dos objetos.

- **Cada objeto tem um tipo.** Cada objeto é uma instância de uma classe, onde classe é um sinônimo de tipo. O que distingue uma classe da outra são os tipos de "mensagens" que se pode enviar a ela e os tipos de dados que ela guarda.

- **Todos objetos de um tipo em particular podem receber as mesmas mensagens (herança).** Por exemplo, um objeto do tipo "círculo" que também é do tipo "forma" poderá garantidamente receber todas as mensagens do tipo "forma". Isto significa que se pode escrever um código que "converse" com objetos do tipo "forma" e este código irá manipular automaticamente qualquer objeto que se ajuste ao tipo "forma". Este é substancialmente um dos conceitos mais poderosos da programação orientada a objetos.

Estes tópicos reúnem as principais características da OOP, a qual tem como principal objetivo o aumento na capacidade de criação do desenvolvedor ou programador. Certamente a leitura destes 5 itens não é o suficiente para uma compreensão completa deste paradigma, para tanto o leitor pode se utilizar de um dos diversos livros que tratam deste assunto, como por exemplo Eckell (2002).

3.3. Uma Nova Linguagem

Reutilização é a chave para o tratamento de complexidades. Em se tratando de *linguagens de modelagem*, a técnica mais bem sucedida para a reutilização de código é o paradigma orientado a objetos (Seção 3.2). Existem algumas linguagens que utilizam-se de alguns dos conceitos deste paradigma para a modelagem de problemas físicos, dentre estas destacam-se: Modelica (Otter e Elmqvist, 2000), gPROMS (Oh e Pantelides, 1996) e AspenDynamics (AspenTech, 2002).

O simulador em projeto neste trabalho poderia simplesmente adotar uma destas linguagens existentes como a sua *linguagem de modelagem*. Contudo, optou-se pelo agrupamento das melhores características de modelagem encontradas nestas linguagens juntamente com novas propostas, levando a uma linguagem mais próxima daquela usada como meio de expressão e comunicação e com maior capacidade de reutilização de código, apresentada a seguir.

A Figura 3.1 sugere um atrelamento entre o nível de abstração de uma linguagem e a flexibilidade que esta dispõe: a medida que caminha-se na direção de um maior nível perde-se em flexibilidade. Este paradoxo é um desafio não só para linguagens de modelagem de processos assim como para linguagens de programação em geral. Por este motivo, a tendência

atual na área de Ciências da Computação é fazer o uso intensivo de geradores automáticos de código e do paradigma orientado a objetos sempre que possível.

Este trabalho propõe como *linguagem de modelagem* para processos dinâmicos uma linguagem que contempla, simultaneamente, dois níveis: baseada em equações (Seção 3.1.2) e baseada em componentes (Seção 3.1.4) além de apresentar um alto nível de utilização do paradigma orientado a objetos. Tal linguagem apresenta três entidades principais: *Model*s, *Device*s e *FlowSheet*s.

Um *FlowSheet* é a abstração do problema em estudo o qual é composto por um conjunto de componentes, denominados *Device*s. Cada *Device* é baseado em uma descrição matemática a qual rege o comportamento de suas variáveis, à esta descrição chamamos de *Model*.

Esta proposta procura suprir a maior parte dos critérios de avaliação em relação à modelagem de processos (Seção 2.1.1), dentre os quais destacam-se: facilidades relativas à criação e edição de bibliotecas de modelos; modularidade; implementação de conceitos de composição e herança do paradigma orientado a objetos.

A Figura 3.3 apresenta uma visão global da estrutura proposta para o simulador em desenvolvimento. Nesta figura os conceitos de *Model* e *FlowSheet* podem ser visualizados.

Na parte inferior da Figura 3.3 podem ser observados alguns trechos de código na linguagem proposta, note-se que a linguagem de modelagem proposta não é uma linguagem de programação, mas sim uma linguagem descritiva. Cada um dos blocos do grupo *Biblioteca de Models* pode ser descrito em um arquivo de texto em separado, o que significa a possibilidade da criação de bibliotecas de modelos editáveis. Mais detalhes sobre *Model*s e *FlowSheet*s podem ser encontrados nas Seções 3.3.1 e 3.3.2, respectivamente.

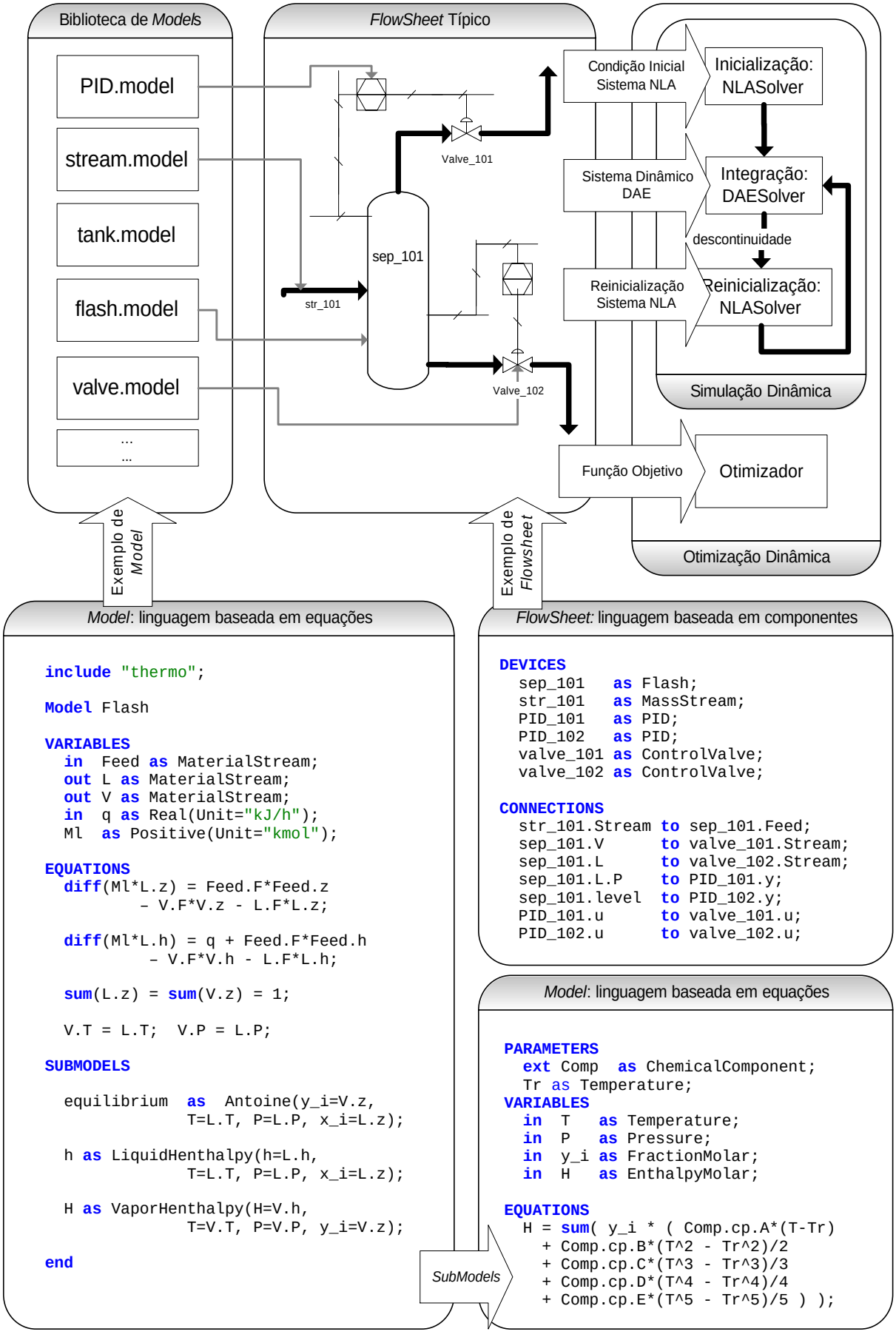


Figura 3.3 Visão geral da estrutura proposta para o simulador em desenvolvimento.

3.3.1. Model

Antes de apresentar a entidade chamada de *Model* é de grande valia uma consulta ao significado da palavra *model* (modelo). Segundo *Webster's New World Dictionary*: *model* é derivado do Latim *modus*, que significa *uma medida*. Utilizado como um substantivo, significa: representação aproximada de um objeto real ou imaginário. O *McGraw-Hill Dictionary of Scientific and Technical Terms* apresenta *model* como: um sistema matemático ou físico, o qual obedece a certas condições especificadas e o comportamento deste é utilizado para o entendimento de sistemas físicos, biológicos ou sociais os quais são análogos de alguma forma.

Na linguagem proposta, um *Model* consiste na abstração matemática de algum equipamento ou dispositivo real, parte de processo ou até mesmo de um *software*. Como exemplos de *Model*s podem ser citados: descrição matemática de um tanque, tubulação ou controlador PID.

Cada *Model* pode ter parâmetros, variáveis, equações, condições iniciais, condições de fronteira e *Model*s internos (modelagem hierárquica ou composição) os quais por sua vez podem ter *Model*s internos também. *Model*s podem ser baseados em outros preexistentes e novas características podem ser adicionadas (novas equações, variáveis, etc). Desta forma os principais conceitos do paradigma orientado a objetos de composição e herança são contemplados pela linguagem.

A Figura 3.5 apresenta a sintaxe para a descrição de um *Model* e segue a seguinte formatação: identificadores entre `[colchetes]` são opcionais sendo que os colchetes devem ser removidos no caso da utilização do comando; o sinal `+` significa que o grupamento que o precede pode aparecer uma ou mais vezes separados por vírgula; identificadores entre `"aspas"` são considerados texto; a linguagem diferencia letras maiúsculas e minúsculas tanto para os identificadores declarados pelo usuário quanto para as funções e palavras-chave do sistema; identificadores em **`negrito`** são palavras reservadas da linguagem, sendo que os em letras **`MAIULSCULAS`** definem a entrada em uma nova seção, os em letras **`minusculas`** são identificadores de comandos; com exceção dos identificadores de seção (**`VARIABLES`**, **`EQUATIONS`**, etc) e declaração de contextos, todos os comandos internos a um *Model* devem ser seguidos de ponto-e-vírgula. O caractere `#` demarca o início de uma linha de comentário.

A sintaxe apresentada na Figura 3.5 caracteriza uma linguagem descritiva, pois a mesma apresenta apenas declarações e não instruções a serem executadas. Além disto, esta linguagem utiliza-se do conceito de *seções* de declaração, onde cada seção tem como propósito a declaração de entidades diferentes. A entrada em uma nova seção é caracterizada por um identificador único, por exemplo, o identificador **`VARIABLES`** determina a entrada na seção de declaração de variáveis. Um maior detalhamento de cada identificador especial presente na Figura 3.5 segue.

Criando Models

Quando deseja-se criar um novo modelo de descrição *Model*, existem duas opções: desenvolver um modelo totalmente novo ou utilizar como base um ou mais modelos existentes. Em ambos casos inicia-se a declaração do modelo com o identificador **`Model`** (linha 1, Figura 3.5) o qual define o inicio da declaração de um novo modelo. O identificador **`end`** define o final da declaração do modelo e é obrigatório (linha 18).

Quando o modelo em desenvolvimento é muito semelhante a um modelo preexistente, porém com algumas particularidades, torna-se interessante a utilização do preexistente como base do novo. Um modelo pode utilizar-se de outros modelos previamente definidos como base pela utilização do comando **`as`** (linha1), isto faz com que todas as características dos modelos bases sejam herdadas automaticamente, dispensando a necessidade da utilização da técnica de copiar e colar.

Quando um ou mais modelos são utilizados como base na declaração de um novo modelo, este se torna um clone das suas bases, porém com características adicionais devido às declarações nas seções posteriores. Este método de utilização de bases é um conceito do paradigma orientado a objetos, conhecido como de herança (*inheritance*) e gera uma grande capacidade de reutilização de código, pois uma vez que os sistemas base são alterados ou seus defeitos corrigidos, estas modificações refletem-se automaticamente em todos os sistemas que derivam deste.

```
1   Model ModelName [as [BaseModel]+]

2     PRAMETERS
3     [ext]  ParameterName [as BaseType[([Member = Value]+)]];

4     VARIABLES
5     [in | out]  VariableName [as BaseType[([Member = Value]+)]];

6     EQUATIONS
7       [”equation name”] LeftExpression = RightExpression;

8     INITIAL
9       [”equation name”] LeftExpression = RightExpression;

10    SUBMODELS
11      subModelName  as  ModelName[(Member = Expression)+];

12    SET
13      ParameterName = Expression;

14    CONNECTIONS
15      OutputVariable to InputVariable;

16  end
```

Figura 3.5 Sintaxe para a descrição de um *Model*.

Adicionando parâmetros a um Model

Normalmente a modelagem de um sistema físico apresenta um conjunto de constantes que caracterizam o mesmo, estas constantes são denominadas parâmetros do modelo. Todo parâmetro deve ter seu valor designado em algum ponto após a sua declaração. O valor de parâmetro nunca será o resultado de algum cálculo do simulador, salvo o caso de estimação de parâmetros.

Na linha 2 da Figura 3.5 pode ser observado o identificador **`PARAMETERS`**, o qual define a entrada na seção de declaração de parâmetros. Cada comando válido dentro desta seção declara um novo parâmetro. Cada novo parâmetro deve ter como base um tipo predefinido `BaseType` (linha 3). O sistema tem como tipos embutidos: **`Real`** para representação de variáveis ou parâmetros contínuos, **`Integer`** para inteiros, **`Boolean`** para os valores lógicos (**`true`** ou **`false`**) e **`Text`** para parâmetros do tipo texto.

Assim, para a declaração de um parâmetro inteiro com nome `NumberOfComponents`, com limite inferior 0 e superior 10 o seguinte código poderia ser utilizado:

```
PARAMETERS

    NumberOfComponents as Integer (Lower=0, Upper=10);
```

Muitas vezes deseja-se compartilhar o valor de um ou mais parâmetros entre diversos componentes de um *FlowSheet*. Isto pode ser obtido pela utilização do prefixo **`ext`** na declaração do parâmetro, fazendo com que o mesmo seja um parâmetro externo ou *global*. Neste caso, quando o modelo for utilizado em um *FlowSheet*, este deve conter um parâmetro de mesmo nome o qual será o parâmetro *global* fonte. À exceção dos parâmetros externos, todo o parâmetro deve ter seu valor designado antes da utilização do modelo para fins de simulação e isto é feito na seção **`SET`**, apresentada adiante.

Como pode ser observado, um parâmetro é declarado com base em um tipo preexistente. Se um parâmetro é declarado sem um tipo base algum, este é considerado como do tipo embutido **`Real`**. Uma melhor explanação sobre os tipos segue.

Tipos de parâmetros e variáveis

Como visto, tipos são utilizados para a declaração de parâmetros de um modelo, além disto os tipos são utilizados de forma idêntica na declaração de variáveis. O conjunto de tipos embutidos já foi apresentado anteriormente, mas novos tipos derivados podem ser declarados pelo usuário. Estes devem ser declarados fora do contexto de qualquer *Model* ou *FlowSheet* e seguem a uma sintaxe muito semelhante a sintaxe de declaração de parâmetros e variáveis:

```
TypeName as BaseType[([Attribute = Value]+)];
```

Exemplos da declaração de alguns tipos são apresentados na Figura 3.7. Como pode ser visto nos exemplos desta figura, os tipos embutidos do sistema podem ser derivados para a criação de novos tipos e um ou mais *atributos* do tipo base podem ser modificados, tais como: limite inferior (`Lower`), limite superior (`Upper`), descrição breve (`Brief`), valor padrão por omissão ou estimativa inicial (`Default`), entre outros.

```
1   Positive as Real(Lower=0, Default=1e-3);
2   Negative as Real(Upper=0, Default=-1e-3);

3   EnergyHoldup as Positive(Unit="J");
4   Temperature as Positive(Unit="K",Upper=8000);
5   Pressure as Positive(Unit="Pa");

6   ChemicalComponentBase as structure (
7      Mw  as Positive(Brief="Molecular Weight",Unit="g/mol");
8      Pc as Pressure(Brief="Critical Pressure");
9      Tc as Temperature(Brief="Critical Temperature");
10     vc as Real(Unit="m^3/mol",Brief="Critical Volume");
11    );
```

Figura 3.7 Exemplos de declaração de tipos, base para parâmetros e variáveis.

Assim como em tipos base, na declaração de parâmetros e variáveis é possível modificar os *atributos* que são herdados do tipo base, tais como: limite inferior (`Lower`), superior (`Upper`), unidade de medida (`Unit`), descrição breve (`Brief`), etc. A utilização de unidades de medida é tratada com maior detalhe a seguir.

Além disto, como pode ser observado na Figura 3.7, existe a possibilidade de declarar tipos que na verdade são um agrupamento de tipos internos. Tipos pertencentes a esta categoria são conhecidos como estruturas. A vantagem da utilização de estruturas está no agrupamento de informações correlatas em uma única entidade. Este tipo de agrupamento gera uma modelagem mais enxuta e facilita muito a troca de informações entre *Devices*, pois com a conexão de apenas uma variável um conjunto inteiro de informações pode ser compartilhado.

Adicionando variáveis a um Model

Todo modelo contém um conjunto de variáveis as quais descrevem o comportamento do sistema. Na linguagem proposta, variáveis são declaradas de forma análoga aos parâmetros e tipos base, porém com a utilização do identificador de seção **`VARIABLES`**. Todo modelo deve conter ao menos uma variável, pois um modelo sem variável alguma não tem sentido de existir. No caso das variáveis, podem ser utilizados os prefixos **`in`** ou **`out`** (linha 6, Figura 3.5) que definem a variável como de entrada ou de saída, respectivamente. Uma variável de entrada consiste apenas em uma referência para a variável que é conectada a ela. Uma

variável de saída difere das variáveis normais apenas pelo fato de ser visível ao mundo exterior ao do modelo atual, podendo ser conectada a alguma variável de entrada de outro *Device*. Tais conexões se dão na seção **`CONNECTIONS`** detalhada mais adiante.

Adicionando equações a um Model

Para descrever o comportamento das variáveis de um modelo, um conjunto de equações deve ser declarado. Na linha 8 da Figura 3.5 pode ser observado o identificador **`EQUATIONS`**, este identificador define a entrada na seção de declaração de equações. Uma equação é uma expressão matemática de igualdade envolvendo qualquer das variáveis, ou parâmetros previamente declarados, incluindo aqueles declarados em *Model*s internos. A declaração de uma equação pode ser precedida por um texto o qual é o *nome* da equação (linha 9). Toda a equação deve conter obrigatoriamente um sinal de igualdade (futuramente sinais de desigualdade também podem permitidos principalmente para estudos de otimização).

Unidades de Medida

Nas seções anteriores pode ser observado que um tipo básico como o **`Real`** apresenta o *atributo* `Unit` o qual guarda a unidade de medida do mesmo. Quando variáveis ou parâmetros são criados estes herdam do tipo todas os seus *atributos*, incluindo a unidade de medida. Desta forma, todo parâmetro ou variável, que em última instancia seja derivado do tipo **`Real,`** apresenta uma unidade de medida.

Assim, sendo a unidade de medida algo inerente a parâmetros e variáveis, testes de consistência de unidades de medida podem ser executados em qualquer expressão da linguagem. Além disto, unidades de medida (expressões de unidades de medida entre aspas duplas) podem ser utilizadas livremente nas expressões da linguagem, por exemplo:

```
Pvap / "atm" = A * exp(B / (T + C));
```

Adicionando condições iniciais a um Model dinâmico

Modelos dinâmicos normalmente requerem um conjunto de condições iniciais para a sua solução, tais condições são equações válidas em um determinado valor da variável independente (o tempo inicial). As condições iniciais são declaradas na seção **`INITIAL`**, a qual é idêntica à seção **`EQUATIONS`**, diferindo apenas no fato de que as equações declaradas nesta seção são utilizadas apenas durante a inicialização do sistema.

Designando valores aos parâmetros

Como já mencionado, todos os parâmetros dos *Model*s de todos os *Device*s pertencentes a um *FlowSheet* devem ter seus valores designados antes de iniciar uma simulação. Isto é feito dentro da seção **SET** (linha 14, Figura 3.5) e pode ser feita diretamente no modelo onde foi declarado o parâmetro ou em qualquer nível superior a ele, e em última instância no próprio *FlowSheet*. A fixação dos valores dos parâmetros é feita conforme apresentado na linha (15). É importante notar que a expressão matemática `Expression` (linha 15) deve ter unidades de medida consistentes com o parâmetro, caso contrário o usuário deve ser advertido com uma mensagem de aviso. Quando o objetivo é designar o valor de parâmetros internos a *Device*s, estes podem ser acessados fornecendo o seu *caminho*, da seguinte forma:

```
DeviceName[ . SubDeviceName  ].ParameterName
```

Desta forma, parâmetros de *Device*s em qualquer nível interno podem ser acessados e então ter seus valores designados na seção **SET**. Esta forma de acesso a entidades internas não se restringe apenas para o caso de parâmetros pertencentes a *Device*s internos ou apenas à seção **SET**. Ela pode ser utilizada também para variáveis de *Model*s internos ou então no caso de estruturas (variáveis ou parâmetros que contém variáveis internas) e em outras seções (**EQUATIONS**, **INITIAL**, etc).

Especificando contextos

O acesso através da utilização de *caminhos* de variáveis ou parâmetros pode se tornar repetitivo, principalmente na seção **SET** quando são fixados os valores de um grande grupo de parâmetros. O comando **context** pode ser utilizado para contornar este problema, facilitando o acesso de entidades internas umas às outras pela utilização de contextos, da seguinte forma:

```
context ContextName
    ...
end
```

Com a utilização de um contexto como declarado acima, todo nome de variável que se encontra entre **context** e **end** será implicitamente precedido por `ContextName.`. Um contexto pode também ser declarado internamente a outro e não há limitação no número de níveis de contextos internos que podem ser utilizados. Note-se que o nome do contexto declarado deve ser o de um *Device* ou sub*Model* válido. Exemplos da utilização de contextos podem ser encontrados na Seção 3.4.

Conectando variáveis de entrada e saída

Variáveis podem ser declaradas como sendo de entrada ou saída (prefixos **`in`** ou **`out`**, respectivamente). Tais variáveis podem ser conectadas umas às outras seja entre *Models* internos ou *Devices* de um *FlowSheet*. Estas conexões se dão dentro da seção iniciada pelo identificador **`CONNNECTIONS`** (linha 16, Figura 3.5). Comandos dentro desta seção seguem a sintaxe demonstrada na linha 17 e formalizam uma nova conexão entre uma variável de saída e uma variável de entrada, onde estas devem ser compatíveis (unidades de medida consistentes e elementos internos compatíveis no caso de estruturas).

Deve ser lembrado que, diferentemente dos pacotes de simulação usuais, uma conexão não significa uma nova equação de igualdade implícita, mas sim que a variável de entrada é apenas uma referência para a variável de saída a ela conectada. Isto reduz tanto o número de variáveis quanto o de equações do sistema.

Composição com modelos internos

A utilização do identificador **`SUBMODELS`** (linha 12, Figura 3.5) é opcional e define a entrada na seção de declaração de *Model*s internos. Cada comando dentro desta seção declara um ou mais modelos que estão em um nível hierárquico abaixo do modelo atual. Esta seção é responsável pela implementação do conceito de composição do paradigma orientado a objetos. A seção **`SUBMODELS`** tem sintaxe idêntica a seção **`DEVICES`**, presente nos *FlowSheets.* Exemplos de utilização de ambas as seções de declaração são apresentados a seguir.

3.3.2. FlowSheet e Devices

Na seção anterior foi apresentada a linguagem de descrição da entidade denominada *Model*. Um *Model* pode ser considerado uma entidade abstrata uma vez que esta não está diretamente relacionada com um equipamento concreto, mas sim com uma *classe* de equipamentos. Para a representação de problemas concretos e seus componentes são utilizados *FlowSheet*s e *Devices*, respectivamente. A linguagem de descrição de um *FlowSheet*, que é análoga a linguagem de descrição de um *Model*, é apresentada nesta seção.

Em inglês o substantivo *device* advém do inglês antigo *devis*, *devise* e pode ser traduzido para o português como *dispositivo*, que por sua vez deriva do lat. *dispositus* e significa: mecanismo ou conjunto de meios com vista a um determinado fim (Aurélio Séc. XXI). *Flow sheet* (diagrama de processo) é um sinônimo de *flowchart* e tem como significado formal: um diagrama que representa um procedimento ou sistema em forma de passos utilizando-se de conexões por linhas e um conjunto de símbolos (Merrian Webster, 2002).

Na linguagem proposta um *Device* é a instância de um *Model* e representa uma peça **real** do processo em análise. Desta forma, um único *Model* pode ser utilizado como base para diversos *Device*s os quais tem a mesma estrutura, mas diferem em alguns aspectos (valores dos parâmetros, conexões ou especificações). *Device*s podem ser conectados uns aos outros para formar um *FlowSheet* (exemplificado pelo grupamento *FlowSheet Típico*, Figura 3.3) o qual é a abstração da linguagem do problema que se quer resolver.

Para a descrição de um *FlowSheet* desenvolveu-se uma linguagem baseada em componentes (Seção 3.1.4). Esta linguagem tem como objetivo ser simples o suficiente para ser totalmente manipulada por uma ferramenta gráfica, dispensando o aprendizado da mesma pelo usuário. A sintaxe proposta aparece na Figura 3.9.

```
1   FlowSheet FlowSheetName [as BaseModel +]

2     DEVICES
3     DeviceName as BaseModel;

4     CONNECTIONS
5     OutputVariable  to  InputVariable;

6     SPECIFY
7     VariableName = Expression;

8   end
```

Figura 3.9 Sintaxe resumida para a descrição de um *FlowSheet*.

A representação da Figura 3.9 segue a mesma convenção apresentada para a sintaxe da descrição de um *Model* (Seção 3.3.1). Embora esta figura apresente apenas três seções para a descrição de um *FlowSheet* (**DEVICES**, **CONNECTIONS**, **SPECIFY**), todas as seções presentes na descrição de um *Model* também são válidas e idênticas na descrição de um *FlowSheet* (com exceção da seção **SUBMODELS**, que tem como sua equivalente a seção **DEVICES**) e foram omitidas por simplicidade.

Criando FlowSheets

Da mesma forma que um *Model*, um *FlowSheet* pode utilizar-se de um ou mais modelos como base (linha 1, Figura 3.9) e sua declaração é terminada pelo identificador **end** (linha 6). Conceitualmente um *FlowSheet* e um *Model* são bem diferentes e estas diferenças vem sendo enfatizadas a todo momento. Mas quanto às suas descrições formais estas entidades mostram-se quase idênticas, a única diferença ocorre pela presença da seção **DEVICES** em substituição da seção **SUBMODELS**.

Adicionando Devices a um FlowSheet

O identificador de inicio de seção **DEVICES** inicia a seção onde são declarados os *Device*s do *FlowSheet*. Cada *Device* é declarado de forma análoga à declaração de variáveis e parâmetros, diferindo apenas pelo fato de que este não utiliza-se de um tipo como base mas sim um *Model* previamente declarado (linha 3). Uma vez declarado, um *Device* pode ser referenciado pelo identificador fornecido na sua declaração e o mesmo contém todas as informações contidas no *Model* utilizado como base (variáveis, parâmetros, equações, etc).

Conectando Devices

Todo o processo é formado por um conjunto de dispositivos, sendo que cada dispositivo também é formado por um subconjunto de dispositivos e assim por diante dependendo do grau de detalhamento utilizado. De forma totalmente análoga, a linguagem aqui proposta apresenta a descrição de um processo como um *FlowSheet* o qual contém um conjunto de *Device*s os quais podem conter sub*Device*s. Mas existem relações entre os dispositivos de um processo e estas são representadas na linguagem por conexões entre os *Devices.* As conexões entre *Devices* são idênticas as apresentadas na descrição de *Model*s.

Fixando variáveis através Especificações

Normalmente um *Model* apresenta um número de graus de liberdade maior do que zero. Assim, um *FlowSheet* que utilize tal modelo poderá escolher o modo de operação do sistema através da especificação de um determinado conjunto de variáveis de tais *Model*s. Estas especificações se dão dentro da seção iniciada pelo identificador **SPECIFY** (linha 4, Figura 3.9). Embora a utilização de especificações desta forma faça mais sentido quando se trata de um *FlowSheet*, este comando também pode ser utilizado na declaração de um *Model* e foi omitido propositadamente da Seção 3.3.1.

3.4. Exemplos de Aplicação da Linguagem

Problemas de simulação dinâmica de processos podem ser representados pela linguagem proposta neste trabalho de diversas formas, cada uma delas é mais adequada ao grau de complexidade do problema em mãos e ao nível de detalhamento da descrição.

Nas seções a seguir são apresentados alguns paradigmas ou formas de utilização da linguagem, os quais estão ordenados pelo grau de complexidade ou nível de detalhamento do problema que se quer resolver: variando desde a modelagem de sistemas simples com apenas um nível de detalhamento (não há *Devices*, mas apenas algumas variáveis e equações), até o

desenvolvimento de modelos para uma biblioteca e a sua utilização para modelagem de seções de uma planta industrial.

3.4.1. Modelos Simples – Apenas FlowSheet

Considere-se um problema descrito em apenas um nível de detalhamento, no qual o processo em estudo é composto por apenas um equipamento ou dispositivo o qual pode ser descrito por um pequeno número de variáveis, parâmetros e equações. Para estes casos, a criação de um *Model* e posterior utilização deste por um *Device* dentro de um *FlowSheet* pode significar uma burocracia desnecessária. Este trabalho adicional pode ser evitado através da declaração do modelo do sistema diretamente no *FlowSheet*, reduzindo a complexidade, que é desnecessária em casos de tal simplicidade. Exemplos da aplicação deste paradigma podem ser observados na Figura 3.11 e Figura 3.13.

O sistema modelado na Figura 3.11 representa um pêndulo com uma haste rígida e pode ser formulado pelo seguinte conjunto de equações:

$$\begin{aligned} w &= x' \quad (a) \\ z &= y' \quad (b) \\ T.x &= w' \quad (c) \\ T.y - g &= z' \quad (d) \\ x^2 + y^2 &= L^2 \quad (e) \end{aligned} \tag{3.5}$$

$$\begin{aligned} x(0) &= 0.5 \quad (a) \\ w(0) &= 0 \quad (b) \end{aligned} \tag{3.6}$$

onde, x e y representam as posições horizontal e vertical do pêndulo, w e z são as correspondentes velocidades nestas direções, T é a tensão na haste, L o seu comprimento e g é a aceleração da gravidade. Pode-se observar a total similaridade entro os dois modos de expressar o sistema.

```
1   FlowSheet Pendulum
2     PARAMETERS
3     g as Real(Unit="m/s^2", Brief="Gravity accell");
4     L as Real(Unit="m", Brief="Cable Length");

5     VARIABLES
6     x as Real(Unit="m",Brief="Position x");
7     y as Real(Unit="m",Brief="Position y");
8     w as Real(Unit="m/s",Brief="Velocity in x");
9     z as Real(Unit="m/s",Brief="Velocity in y");
10    T as Real(Unit="m/s",Brief="Tension on cable");

11    EQUATIONS
12    "Velocity in x" w = diff(x);
13    "Velocity in y" z = diff(y);
14    "Tension in x"  T*x = diff(w);
15    "Tension in y"  T*y -g = diff(z);
16    "Position constraint"   x^2 + y^2 = L^2;

17    INITIAL
18    "Initial x position"     x = 0.5*"m";
19    "Initial x velocity"   w = 0*"m/s";

20    SET
21    g = 9.8*"m/s^2";
22    L = 1*"m";
23  end
```

Figura 3.11 Declaração do *FlowSheet* para o sistema de um pêndulo com uma haste rígida, modelado em coordenadas cartesianas.

Este exemplo é amplamente referenciado na literatura que trata sobre o índice de sistemas algébrico-diferenciais (Pantelides, 1988, Mattsson *et al.* 2000 e Bachmann *et al.*, 1990) por uma série de motivos: é um sistema real, de modelagem simples e, modelado como apresentado na Figura 3.11, apresenta índice diferencial igual a três. A solução deste sistema e um maior detalhamento do mesmo podem ser encontrados na Seção 6.2.2.

Na Figura 3.11 pode ser observado o identificador de função **`diff`**. Na linguagem proposta, derivadas com relação à variável independente são obtidas pela utilização da função **`diff`** com apenas um argumento (a expressão que se deseja derivar, linhas 12-15 da Figura 3.11). No caso de haver mais de uma variável independente, o mesmo operador pode ser utilizado para a obtenção de derivadas parciais, porém com a utilização de dois argumentos:

```
diff(expression, independentVar)
```

Assim, sistemas com mais de uma variável independente podem ser representados e, a nível de linguagem, não é necessária discretização alguma. Embora, no momento de se obter a solução, o simulador deverá aplicar algum tipo de discretização (diferenças finitas, elementos finitos, volumes finitos, etc) a todas as variáveis independentes, exceto ao tempo, que será a variável independente de integração. A especificação mais detalhada deste tipo de problema,

assim como o estudo e implementação de métodos de discretização, serão objetivo de trabalhos futuros.

```
1   FlowSheet Galvanic
2     PARAMETERS
3     F as Real(Brief="Faraday constant");
4     R as Real(Brief="Ideal Gas constant");
5     T as Real(Brief="Temperature");
6     rho; W; V; phi1; phi2; i01; i02; iapp;

7     VARIABLES
8     y1 as Real(Unit="A",Brief="Mole fraction of NiOOH",
9           Default=0.05);
10    y2 as Real(Brief="Potential at solid-liquid interface",
11          Default=0.38);

12    EQUATIONS
13    rho*V/W*diff(y1) = i01*(2*(1-y1)*exp(0.5*F/(R*T)*(y2-phi1))-
14        2*y1*exp(-0.5*F/(R*T)*(y2-phi1)))/F;
15    i01*(2*(1-y1)*exp(0.5*F/(R*T)*(y2-phi1))-
16        2*y1*exp(-0.5*F/(R*T)*(y2-phi1)))
17        +i02*(exp(F/(R*T)*(y2-phi2))-
18         exp(-F/(R*T)*(y2-phi2)))-iapp = 0;

19    INITIAL
20    y1 = 0.05;
21    #y2 = 0.38;
22  end
```

Figura 3.13 Declaração do *FlowSheet* para um processo galvanostático de um filme fino de hidróxido de níquel.

O exemplo modelado na Figura 3.13, é apresentado no trabalho de Wu and White (2001) como um problema teste para códigos computacionais para a inicialização de sistemas de equações algébrico-diferenciais. O modelo na forma matemática, assim como a inicialização e integração deste sistema podem ser encontrados na Seção 6.2.1.

Na linha 6 da Figura 3.13 pode ser observada a declaração de alguns parâmetros sem a utilização de tipo base algum, nestes casos o tipo **`Real`** é utilizado automaticamente pelo sistema, como citado na Seção 3.3.1.

3.4.2. Utilizando Modelos Existentes

Quando se tem disponível uma biblioteca de modelos, ou mesmo algum modelo previamente desenvolvido, este pode ser utilizado diretamente (sem a necessidade de cópia dos textos descritivos) com auxílio do comando **`include`** (linha 1, Figura 3.15). Este

comando funciona exatamente como o seu nome sugere, faz com que o arquivo gravado com um determinado nome seja incluído no arquivo atual, fazendo com que todas as entidades declaradas neste arquivo (tipos base e *Model*s) possam ser utilizados. O comando **include** não se restringe apenas à arquivos, diretórios inteiros (todos os arquivos internos a este) podem ser incluídos com a utilização deste comando.

```
1   include "source.mso", "tank.mso";

2   FlowSheet ThreeTank;
3     DEVICES
4     feed as Source;
5     tank1 as Tank;
6     tank2 as Tank;
7     tank3 as Tank;

8     CONNECTIONS
9     feed.output to tank1.input;
10    tank1.output to tank2.input;
11    tank2.output to tank3.input;
12  end
```

Figura 3.15 *FlowSheet* com a utilização de *Model*s definidos em arquivos separados.

O exemplo da Figura 3.15 representa um sistema de três tanques em série com uma única alimentação no primeiro tanque. Ambos modelos (`Source` e `Tank`) estão descritos em arquivos separados e são incluídos com o comando **include** (linha 1). A descrição dos modelos `Source` e `Tank` pode ser observada na Figura 3.17.

```
1   Model Source;
2     VARIABLES
3     out output as Real(Brief="Vazão de saída",Unit="m^3/s");
4   end

5   Model Tank;
6     PARAMETERS
7     k as Real(Brief="Constante da válvula",Unit="m^2.5/s");
8     A as Real(Brief="Área da seção reta do tanque",Unit="m^2");

9     VARIABLES
10    in input as Real(Brief="Vazão de entrada",Unit="m^3/s");
11    out output as Real(Brief="Vazão de saída",Unit="m^3/s");
12    h as Real(Brief="Altura do nível do tanque",Unit="m");

13    EQUATIONS
14    output = k * sqrt(h);
15    diff(h)*A = input – output;
16  end
```

Figura 3.17 *Model*s utilizados pelo sistema modelado na Figura 3.15.

3.4.3. Construindo Novos Modelos

A construção de novos *Model*s é análoga à de *FlowSheet*s, descrita na Seção 3.4.1. A seguir são apresentados alguns exemplos nos quais são enfatizadas as principais características da *linguagem de modelagem* proposta neste capítulo.

O *Model* apresentado na Figura 3.19 é um modelo abstrato ou base. Sua utilização direta não é possível, pois não é fornecida uma equação para a taxa de reação `rate`. Como tal, este modelo pode ser utilizado como base para a construção de um modelo mais especifico, no qual a taxa de reação pode ser especificada. Na Figura 3.21 é apresentado um modelo que utiliza-se desta facilidade de herança (*inheritance*) da linguagem.

```
1   include ”stdtypes.mso”;

2   Model BaseIsothermalCSTR;
3       PARAMETERS
4       NoComp as Integer(Brief=”Number of Components”);

5       VARIABLES
6       rate(NoComp) as Real(Brief=”Reaction rate”);
7       C(NoComp) as Real(Brief=”Molar Concentration”);
8       T as Temparature;
9       V as Volume(Brief=”Total volume”);
10      out output as MaterialStream;
11      in input as MaterialStream;

12      EQUATIONS
13      ”Component Mass Balance” diff(C*V) = input.F*input.z
14                               – output.F*output.z + rate;

15      INITIAL
16      ”Inital Concentration” C = 0;
17  end
```

Figura 3.19 *Model* base de um reator CSTR isotérmico.

```
1   include ”baseisothermalcstr.mso”;

2   Model MethylBromideProduction as BaseIsothermalCSTR;
3       PARAMETERS
4       k as Real(Brief=”Reaction constant”);

5       EQUATIONS
6       rate(1) = rate(2) = k*C(1)*C(2);
7   end
```

Figura 3.21 *Model* de um reator CSTR isotérmico para produção de Brometo de metila, baseado no modelo da Figura 3.19.

O *Model* declarado na Figura 3.21 é um modelo *derivado* do modelo previamente declarado `BaseIsothermalCSTR`. A utilização deste método permitiu a modelagem de um reator CSTR isotérmico para um reator e reação em específico (produção de brometo de metila) com um código muito reduzido.

3.5. Relação com outras Linguagens

A nova linguagem de modelagem apresentada na Seção 3.3 foi inspirada em uma série de sistemas com propósitos semelhantes ao sistema em projeto neste trabalho. Estas linguagens são Modelica (Otter e Elmqvist, 2000), gPROMS (Oh e Pantelides, 1996) e AspenDynamics (AspenTech, 2002). Nesta seção é apresentada uma sucinta comparação entre a proposta deste trabalho e estas linguagens.

3.5.1. Modelica

Modelica é uma linguagem de modelagem orientada a objetos projetada para a modelagem de sistemas físicos. Modelica é um sistema livre, mas não provê uma ferramenta de *software*, é apenas a especificação de uma linguagem juntamente com uma biblioteca de modelos escrita nesta linguagem.

A Figura 3.23 apresenta a modelagem de um conjunto de componentes na linguagem Modelica. Pode-se observar uma grande semelhança desta linguagem com a linguagem de modelagem proposta.

```
model MotorDrive
  PID       controller;
  Motor     motor;
  Gearbox   gear   (n=100);
  Inertia   inertia(J=10);
equation
  connect(controller.outPort, motor.inPort);
  connect(controller.inPort2, motor.outPort);
  connect(gear.flange_a     , motor.flange_b);
  connect(gear.flange_b     , inertia.flange_a);
end MotorDrive;
```

Figura 3.23 Modelo de um grupo de componentes na linguagem Modelica.

Para os conhecedores das linguagens de programação C++ ou Java uma modelagem na linguagem Modelica parece natural. Porém, o objetivo deste trabalho é o desenvolvimento de

uma linguagem de descrição e não uma linguagem de programação e esta é a principal fonte da discrepância entre a linguagem proposta e a linguagem Modelica.

Um aspecto interessante da linguagem Modelica, é a possibilidade da declaração de tipos, como demonstrado na Figura 3.24. Esta característica foi introduzida de forma muito semelhante na linguagem proposta, porém sem a necessidade do identificador **type** e com o sinal **=** substituído pela palavra chave **as**, deixando mais claro a herança de propriedades que ocorre.

```
type Angle  = Real(quantity="Angle" , unit="rad", displayUnit="deg");
type Torque = Real(quantity="Torque", unit="N.m");
```

Figura 3.24 Declaração de tipos na linguagem Modelica.

Os exemplos acima citados apresentam as principais características da linguagem Modelica. Porém, se a modelagem de problemas mais complexos for analisada, como os presentes na biblioteca livre de modelos Modelica, poderá notar-se que estes confundem o usuário pela presença de comandos que nem sempre mantém o caráter descritivo. Talvez este seja o motivo pelo qual a linguagem Modelica ainda não obteve sucesso na aplicação a problemas de engenharia química ficando restrita praticamente a problemas de elétrica e eletrônica.

De forma geral a linguagem Modelica apresenta-se muito concisa, e foi inspiração para alguns conceitos utilizados na proposta de linguagem deste trabalho. Por outro lado, em alguns aspectos, a linguagem Modelica se aproxima muito das linguagens de programação usuais, fugindo do objetivo deste trabalho: gerar uma linguagem de modelagem descritiva.

3.5.2. gPROMS

gPROMS é uma ferramenta de *software* para modelagem, simulação e otimização de processos dinâmicos em geral, o qual apresenta uma linguagem de modelagem própria. Grande parte das características apresentadas na linguagem proposta são inspiradas no gPROMS ou tem como objetivo suprir suas deficiências.

A linguagem do gPROMS utiliza-se do conceito de seções de descrição exatamente como apresentado na linguagem proposta, fato que organiza e facilita o entendimento das informações contidas na descrição dos modelos.

Uma das grandes desvantagens da linguagem utilizada no gPROMS é o baixo grau de utilização dos conceitos do paradigma orientado a objetos (principalmente o conceito de herança). Este fato reflete-se na forma precária com que são descritos os tipos de variáveis. Além disto, no gPROMS os parâmetros não podem utilizar os mesmos tipos de variáveis. Isto

impossibilita um bom sistema de checagem de unidades de medida, uma vez que as unidades de medida das variáveis são apenas ilustrativas e os parâmetros não podem apresentar unidades de medida.

3.5.3. Aspen Dynamics

O Aspen Dunamics também é um sistema para modelagem, simulação e otimização de processos dinâmicos em geral, o qual apresenta uma linguagem própria. Embora ambos, Aspen Dynamics e gPROMS, tenham a mesma origem, atualmente as suas linguagens diferem bastante.

O Aspen Dynamics apresenta uma linguagem moderna, com alto grau de utilização de paradigma orientado a objetos. Esta linguagem é utilizada com sucesso para a descrição de uma grande biblioteca de modelos dinâmicos para engenharia de processos que é parte do *software*.

O maior problema relacionado com esta linguagem é o alto esforço necessário para o desenvolvimento de novos modelos. Isto ocorre, principalmente, devido à grande variedade de entidades não intercambiáveis presentes na linguagem. Por exemplo, um arquivo escrito na linguagem Aspen Dynamics podem conter as seguintes entidades: Variables, Parameters, Ports, Models, Streams, Generic types, Properties, Atomic, Solver properties, procedures, FlowSheets, Blocks etc. A disponibilidade desta grande quantidade de entidades poderia ser útil se as mesmas fossem permutáveis entre si, o que não ocorre, pois cada entidade tem um propósito diferente em tal sistema.

Os conceitos de herança presentes para os tipos, *Model*s e *FlowSheet*s da linguagem proposta neste trabalho foram fortemente influenciados pelos presentes no Aspen Dynamics. Já a grande variedade de entidades diferentes presentes no Aspen Dynamics foram substituídas pela simplicidade sem perda de generalidade. Por exemplo, não é necessário declarar uma *stream* ou *port* para se fazer conexões entre componentes de um diagrama de processo, isto é feito diretamente com a variável, declarando-a como de entrada ou saída.

3.6. Conclusões

Uma nova linguagem orientada a objetos para a descrição de sistemas dinâmicos foi desenvolvida. Sua utilização em problemas típicos tem provado que seu alto nível de abstração e utilização do paradigma orientado a objetos gera uma grande eficiência na reutilização de código.

Além disto a linguagem proposta permite a modelagem em diferentes níveis de detalhamento: um nível mais externo para a descrição topológica de processos e um nível

mais interno para a descrição de modelos matemáticos, o qual aproxima-se bastante da linguagem matemática usual e provê um alto grau de flexibilidade ao usuário.

Este capítulo apresentou uma linguagem de modelagem para processos dinâmicos, mas esta certamente não compreende todos os aspectos possíveis relacionados com a modelagem de sistemas dinâmicos e nem poderia. Portanto, são deixados como proposta de trabalhos futuros uma melhor investigação dos seguintes tópicos:

- Inclusão de equações íntegro-diferencais;
- Linguagem para compreender a solução de problemas de otimização;
- Estabelecer a possibilidade da definição de funções customizadas do usuário, assim como o formato de chamada de funções escritas em outras linguagens de programação, tais como C ou FORTRAN;
- Escrever e gerar modelos matriciais;
- Possibilidade da presença de modelagens aproximadas sendo utilizadas como estimativas iniciais para uma modelagem mais rigorosa para utilização juntamente com sistemas com falta de convergência;
- Modelagem de sistemas híbridos (contíno-discretos), seja pela presença de cláusulas condicionais ou outro método de descrição.

Capítulo 4 Estrutura e Diagramas de Classe

No Capítulo 3 foi proposta uma *linguagem de modelagem* para o simulador em projeto neste trabalho. A descrição de um problema nesta linguagem deve ser convertida em sistemas de equações para que os mesmos possam ser tratados por pacotes numéricos. Diagramas de classe (em conformidade com a UML, *Unified Modeling Language*) para a implementação desta conversão e da própria solução dos sistemas de equações são apresentados neste capítulo.

A linguagem escolhida neste trabalho para implementação de códigos protótipo foi o C++, por ser orientada a objetos, de alta portabilidade e grande velocidade de execução dos aplicativos gerados. Embora esteja sendo utilizado o C++, os diagramas de classes apresentados neste capítulo, podem ser mapeados diretamente para uma série de linguagens diferentes do C++.

4.1. Unified Modeling Language - UML

A modularidade intrínseca da programação orientada a objetos (Seção 3.2) torna natural a representação de um *software* na forma de diagramas. A *Unified Modeling Language* (UML) apresenta uma formalização para a representação de sistemas orientados a objetos por meio de diversos tipos de diagramas. Nesta seção são apresentados alguns conceitos de UML extraídos de OMG (2000).

UML é uma linguagem para especificação, visualização, construção e documentação de artefatos relacionados a sistemas de *software*, assim como para modelagem de outros sistemas não relacionados com *software*. Esta linguagem é composta pela coleção das melhores práticas de engenharia que tiveram sucesso na modelagem de sistemas de grande dimensão e complexidade.

A escolha dos modelos e diagramas criados para representar um problema tem uma profunda influência na forma como o problema é atacado e como sua solução é desenvolvida. Abstração e manter o foco no que é relevante, são as chaves para o aprendizado e a comunicação. Tendo em mente estes detalhes, a UML baseia-se nos seguintes itens:

- Todo sistema complexo é mais bem representado por um pequeno conjunto de modelos menores e relativamente independentes;
- Qualquer modelo pode ser expresso em diversos níveis de fidelidade;
- Os melhores modelos são aqueles intimamente ligados à realidade.

Utilizando-se dos conceitos delineados acima, a UML provê vários tipos de visualizações de modelos através de diagramas gráficos. Estas visualizações são diagramas de casos de uso, de classes, de comportamento e de implementação.

4.1.1. Diagramas de Classes

Este capítulo faz uso intensivo de diagramas de classes. Estes diagramas consistem em uma representação gráfica de classes conectadas pelas suas relações, explicitando a estrutura estática do modelo. Por este motivo, um diagrama de classes pode ser chamado também de diagrama estrutural estático, mas diagrama de classes é um nome menor e já consagrado. Um exemplo de um diagrama de classes pode ser visto na Figura 4.1, uma breve explanação sobre a convenção utilizada nestes diagramas é apresentada a seguir, uma documentação completa pode ser encontrada em OMG (2000).

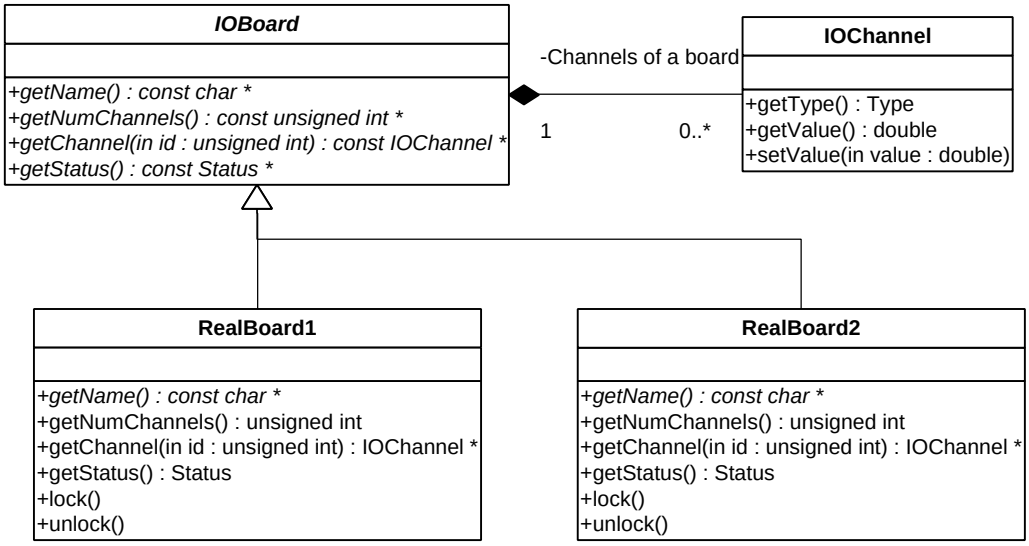


Figura 4.1 Diagrama de classes para um sistema de aquisição de dados.

O diagrama acima apresenta um diagrama de classes que modela, de forma simplificada, um sistema de entrada e saída de dados via uma placa de comunicação, freqüentemente encontrado nas indústrias. O problema é encarado como sendo composto por dois subsistemas: placa de comunicação (*IOBoard*) e canal de comunicação (*IOChannel*), que pode ser tanto um canal de entrada como de saída. Neste modelo uma placa pode retornar seu nome *getName()*, número de canais *getNumChannels()*, um canal em específico *getChannel()* ou seu status atual *getStatus()*. Cada canal tem informações sobre seu tipo (entrada ou saída) *getType()* e seu valor atual *getValue()*, e pode também ter seu valor ajustado *setValue()*.

Cada bloco retangular do diagrama da Figura 4.1 representa uma classe, e é dividido em três seções. A parte superior do retângulo mostra o nome da classe, a intermediária seus atributos ou dados e na inferior são colocadas suas operações (funções membro). Identificadores abstratos são representados em itálico e estáticos em sublinhado. As operações seguem a seguinte convenção:

visibility name (parameter- list) : return-type

- Onde *visibility* deve ser um dos símbolos: + (visibilidade pública, acessado livremente), # (visibilidade protegida, pode ser acessado apenas pela própria classe e suas derivadas) ou – (visibilidade privada, acesso permitido somente à própria classe);

- *name* é o texto identificador da operação;

- *parameter-list* é uma lista formal de parâmetros separados por vírgula, onde cada parâmetro deve ser declarado da seguinte forma:

kind name : type = default-value

- Onde *kind* é um de: **in** (parâmetro de entrada), **out** (parâmetro de saída) ou **inout** (parâmetro de entrada e saída), o padrão é **in** se estiver ausente;
- *name* é o nome do parâmetro;
- *type* é o tipo do parâmetro;
- *default-value* é um valor que é utilizado no caso de chamada da operação sem este parâmetro, este identificador é opcional

- *return-type* é o tipo que é retornado pela operação, se a operação não retornar valor algum o *return-type* deve ser omitido.

Os principais tipos de relações estáticas entre classes também podem ser observados no exemplo da Figura 4.1. Há uma relação de composição (*composition*, representada pela união com uma linha cheia e símbolo terminador ♦) entre uma placa e seus canais, onde uma placa pode ter zero ou mais canais (relação de 1 para 0..* - *Channels of a board*). Relações de herança (*inheritance*, representadas pela união com uma linha cheia e símbolo terminador Δ) estão demonstradas entre as classes *RealBoard1* e *RealBoard2* e a classe base *IOBoard*. A classe *IOBoard* é uma classe abstrata, e tem apenas uma relação indireta com o problema real, mas serve como interface base para qualquer placa de entrada e saída. As classes *RealBoard1* e *RealBoard2* representam placas de entrada e saída de dados de dois fabricantes diferentes. Utilizando este modelo onde cada placa diferente deriva da classe *IOBoard* qualquer código que se utilize desta classe pode também utilizar diretamente suas derivadas, sem nenhuma modificação no código original.

4.2. Estrutura

Com a representação de um sistema na linguagem descrita no Capítulo 3, o pacote em projeto neste trabalho deve ser capaz de conduzir uma simulação dinâmica ou estacionária. Além disto, com a inserção de alguns novos comandos na linguagem, deve ser possível a solução de problemas de otimização dinâmica e estacionária. A arquitetura interna básica do simulador para a solução destes tipos de problemas está representada na Figura 4.2.

Esta estrutura propõe a separação do sistema em estudo (*FlowSheet*) em subsistemas de dois tipos diferentes: sistemas de equações não-lineares NLA (*non-linear algebraic*) e sistemas de equações algébrico-diferenciais DAE (*differential-algebraic equations*). Este esquema vem ao encontro da estrutura proposta no pacote numérico do projeto CAPE-OPEN (CO, 1999), que é um esforço internacional na criação de interfaces padrões para a comunicação e interoperabilidade entre diferentes pacotes de simulação de processos. Esta estrutura interna modular contempla uma série de objetivos deste trabalho (modularidade

interna, fácil atualização) e faz com que seja conquistado um bom grupo de critérios de avaliação (Capítulo 2).

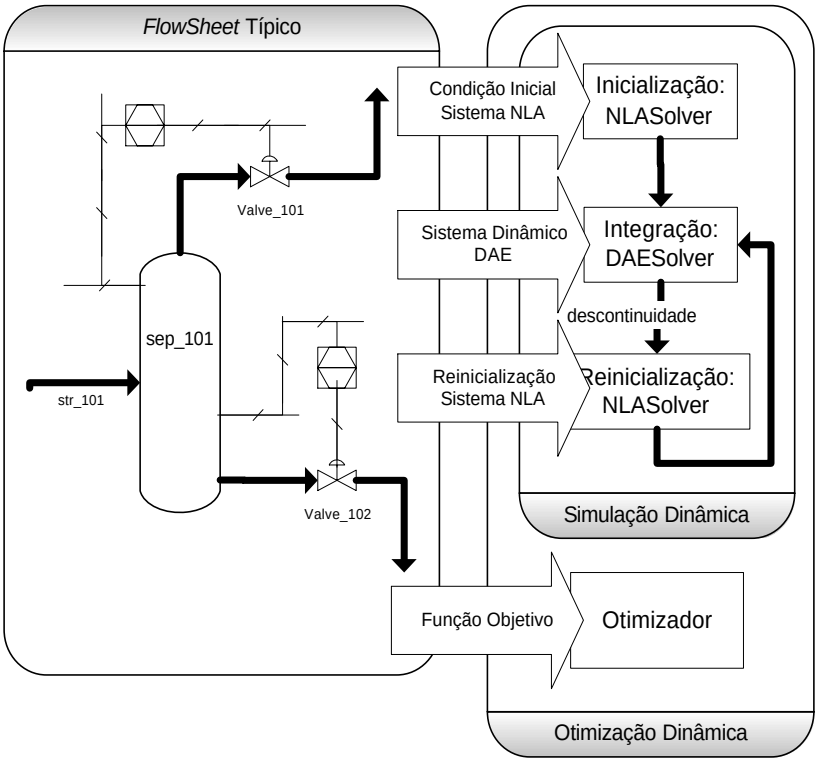


Figura 4.2 Esquematização da arquitetura interna do simulador.

O grupo denominado *Flowsheet Típico* na Figura 4.2 representa a entidade descrita como *FlowSheet* no Capítulo 3, os outros blocos desta figura são tratados conceitualmente e sua implementação é delineada a seguir. Alguns dos métodos necessários para a aplicação prática desta arquitetura são tratados no Capítulo 5.

4.3. Conversão de Equações e Derivação

Para a execução de tarefas diversas, tais como simulações, os sistemas descritos pela linguagem proposta no Capítulo 3 precisam ser convertidos de seu formato de entrada (texto descritivo) para uma outra forma que permita sua avaliação (cálculo de seu resíduo, ver Seção 3.1.2) e cálculo de derivadas.

O cálculo de derivadas é essencial para a solução de sistemas não-lineares e sistemas algébrico-diferenciais. Este cálculo pode ser feito por diferenciação automática, diferenciação simbólica ou aproximado por perturbação numérica, entre outras técnicas. Cada técnica de derivação tem suas vantagens e desvantagens, portanto a aplicação é que determina qual técnica deve ser preferida.

No caso de um simulador dinâmico, a disponibilidade de sistemas de derivação automática e simbólica é de grande interesse. A derivação automática pode ser utilizada na determinação de gradientes dos sistemas e a simbólica para geração de novas equações necessárias na redução de índice de sistemas algébrico-diferenciais (Seção 5.3.2). Neste trabalho foi desenvolvida uma classe para a representação de equações que é adequada tanto para a aplicação das técnicas de diferenciação automática quanto simbólica, apresentada a seguir.

4.3.1. Representação de Equações em Árvore

Com o intuito de permitir o cálculo de resíduos e de derivadas, uma conversão das equações de um modelo para a forma de uma estrutura em árvore é proposta.

A aplicação desta metodologia é apresentada por meio de um exemplo: considere-se a equação (4.1) a qual representa um balanço global de massa de um sistema, com uma entrada *F*, saídas *L* e *V* e uma massa total *M*. A representação em forma de árvore desta equação pode ser vista na Figura 4.3.

$$\frac{dM}{dt} = F - (L + V) \tag{4.1}$$

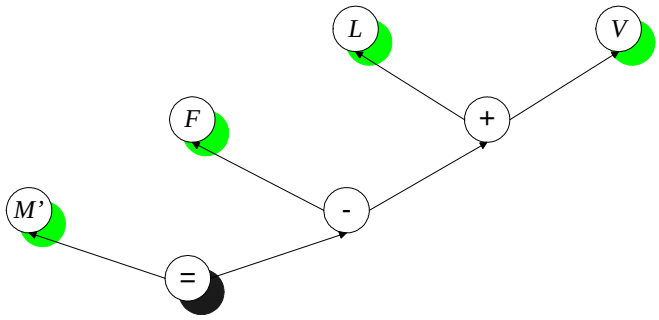


Figura 4.3 Representação da equação (4.1) em forma de estrutura em árvore.

Os vértices ou nós de uma árvore, como a apresentada na Figura 4.3, podem ser operadores (=, +, -, *, ^, etc), funções (*seno*, *cosseno*, *exp*, etc) ou terminadores (parâmetros, números ou variáveis). Desta forma existem três tipos diferentes de vértices: binários (dos quais emanam duas arestas para outros dois vértices, caso dos operadores); unários (dos quais emana apenas uma aresta, caso das funções unárias) e os terminadores ou nós folha, que determinam o final de um ramo da árvore (caso dos valores numéricos, parâmetros ou variáveis).

À exemplo da Figura 4.3, as equações convertidas pelo sistema têm sempre como primeiro nó (nó raiz) o sinal de igualdade. Ao nó raiz estão conectados os lados esquerdo e direito da equação que este representa.

Utilizando-se das facilidades da programação orientada a objetos o sistema de árvore pode ser facilmente representado por uma classe que contenha duas referências para objetos da mesma classe. O diagrama de classes deste objeto pode ser visto na Figura 4.4.

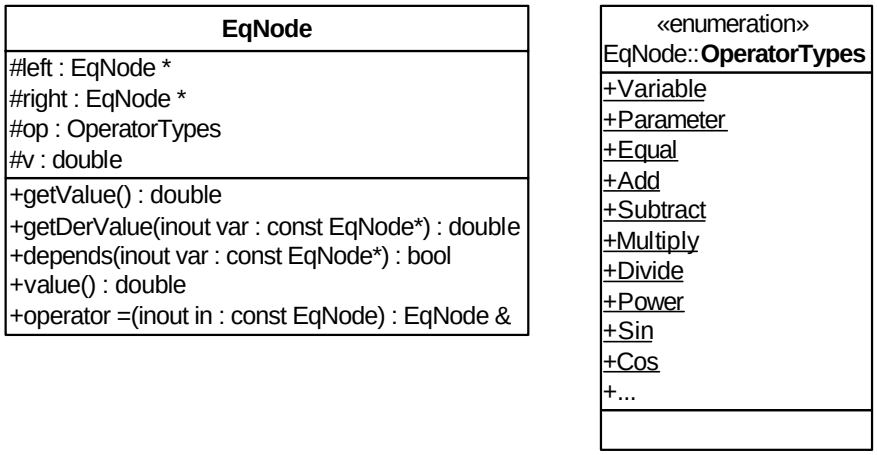


Figura 4.4 Diagrama de classes para a classe que representa um nó da estrutura de árvore.

O diagrama de classes da Figura 4.4 apresenta todos os requisitos necessários para a obtenção do valor do resíduo de um nó (Seção 4.3.2) e da derivada por derivação automática (Seção 4.3.4). Este mesmo sistema de representação para os vértices de equações é suficiente para a obtenção de derivadas simbólicas, como é visto na Seção 4.3.5.

Com a classe *EqNode* a estrutura de árvore é construída da seguinte forma: *left* e *right* são as referências para seus nós descendentes (para o caso de operadores unários *right* é uma referência nula e para nós folha *left* também é nulo); *op* é uma variável inteira de enumeração que identifica o tipo do nó e pode ser um dos valores representados pelo enumerador *EqNode::OperatorTypes* e *v* guarda o valor atual do nó.

4.3.2. Avaliação de Resíduo

A função *value()*, membro da classe *EqNode*, retorna o valor atual do nó que está guardado na variável *EqNode::v*. Já a função *getValue()* primeiramente atualiza o valor de *v* do nó (para refletir possíveis modificações nos nós folha) e então retorna o valor. Esta atualização é feita através da aplicação do Algoritmo 4.1.

Algoritmo 4.1 Atualização dos valores dos nós de equações.

EqNode::getValue( ) **Valor numérico de um nó.**

Saída: valor atualizado do nó.

1. Se este é um nó folha, retornar o valor atual do próprio nó.
2. Se este é um nó binário ou unário, aplicar a respectiva função, como por exemplo:
   - Se *op* é o operador *, fazer *v* igual à *left.getValue()*right.getValue* e retornar este valor*;*
   - Se *op* é a função exponencial, fazer *v* igual à *exp*(*left.getValue())* e retornar este valor*;*

4.3.3. Teste de Dependência

Durante a solução de sistemas de equações, é de grande valia o conhecimento da estrutura do jacobiano do sistema, principalmente para a sua utilização nos problemas de associação apresentados nas Seções 5.2 e 5.3.3.

Com a classe *EqNode*, a dependência de uma determinada equação com uma dada variável ou parâmetro pode ser obtida diretamente com a função membro *EqNode::depends()*. Esta função tem como argumento de entrada o nó que representa a variável ou parâmetro que se deseja testar a dependência e retorna verdadeiro se a equação depende do mesmo e falso no caso contrário. A implementação desta função pode ser feita com base no Algoritmo 4.2, apresentado abaixo.

Algoritmo 4.2 Teste de dependência de um nó de equação.

EqNode::depends(in var const EqNode *) **Teste de dependência com relação a um nó.**

Entrada: referência para o nó em relação ao qual se está sendo testada a dependência.

Saída: verdadeiro se depende do nó, caso contrário falso.

1. Se este é um nó folha e é igual a *var* retornar verdadeiro, caso contrário falso;
2. Se *left.depends(var)* ou *right.depends(var)* retornar verdadeiro, então retornar verdadeiro, caso contrário falso.

4.3.4. Derivação Automática

A diferenciação ou derivação automática (*automatic differentiation*, AD) é uma técnica para o cálculo do **valor numérico exato** de uma derivada, sem a utilização de manipulação simbólica alguma. A técnica de AD é superior à diferenciação simbólica quando se deseja apenas o valor numérico da derivada, pois seu custo computacional é bem inferior, tanto em termos de processamento como em relação à memória necessária (Luca e Musmanno,1997).

Os códigos computacionais para derivação automática dividem-se em dois grandes grupos: os que geram um novo código (baseado no código fonte que descreve as funções que se quer diferenciar) e os que utilizam-se das propriedades de sobrecarga de operadores presentes em algumas linguagens de programação, como o C++.

Em se tratando do primeiro grupo de códigos, o de maior relevância é o ADIFOR (Bischof *et al*., 1992), que é uma ferramenta para derivação automática de programas escritos em FORTRAN 77. Nesta ferramenta, dado um código em FORTRAN 77 e sendo especificadas quais são as variáveis dependentes e independentes, o ADIFOR gera um novo código estendido o qual fornece o cálculo das derivadas parciais de todas as variáveis dependentes com relação às variáveis independentes. Outras ferramentas similares que também podem ser citadas são: ADIC (Bischof *et al*., 1997) e DAFOR (Berz, 1987).

O segundo grupo de códigos para diferenciação automática não requer tipo algum de pré-processamento de código fonte, utilizando-se diretamente das propriedades de sobrecarga de operadores. O pacote mais conhecido pertencente a este grupo é o ADOL-C (Griewank *et al*., 1999). Este pacote utiliza-se da sobrecarga de operadores do C++, mas pode ser utilizado com códigos escritos em C. Além disto, são implementados os métodos direto e reverso de diferenciação automática de qualquer ordem. Pode-se citar ainda o FFADLib (Tsukanov e Hall, 2002) como uma eficiente biblioteca de diferenciação automática direta.

Em um simulador de processos, a utilização das ferramentas que geram códigos derivados fica inviabilizada, uma vez que seriam necessárias diversas etapas intermediárias para cada simulação (pré-processamento da *linguagem de modelagem*, pré-processamento do código gerado, compilação e *link* edição). A utilização de pacotes como o ADOL-C também fica prejudicada, pois este requer um código em C ou C++, tornando-se necessária a tradução da *linguagem de modelagem* e posterior compilação e *link* edição.

Desta forma, uma implementação própria de diferenciação automática é a alternativa mais adequada. E, em se tratando de um simulador de processos dinâmicos, uma implementação relativamente modesta já é o suficiente, com diferenciações apenas de primeira ordem e método de AD direto.

A técnica de AD é baseada na decomposição da função original (a qual se quer diferenciar) em funções elementares, para as quais se conhece a regra de derivação. A

estrutura de representação em árvore demonstrada na Seção 4.3.1 faz exatamente isto, pois cada nó da árvore sempre representa apenas uma função elementar.

A função *EqNode::getDerValue()* retorna o valor da derivada parcial através de um método direto de AD. Então, para o cálculo da derivada de uma função basta chamar esta função no objeto que representa o nó raiz da estrutura em árvore. A implementação da função recursiva *getDerValue()* é demonstrada pelo Algoritmo 4.3.

Algoritmo 4.3 Derivação automática para um nó de equação.

EqNode::getDerValue(in var const EqNode *) **Derivação automática.**

Entrada: referência para o nó em relação ao qual se está derivando.

Saída: valor da derivada em relação ao nó de entrada.

1. Se este é um nó folha, retornar 1 se o nó de entrada é igual a este nó e 0 no caso contrário.
2. Se este é um nó binário ou unário, aplicar a respectiva regra de derivação, por exemplo:
   - Se *op* é o operador *, retornar: *left.value()*right.getDerValue() + left.getDervalue()*right.value();*
   - Se *op* é a função exponencial, retornar: *left.value()*left.getDerValue();*

O Algoritmo 4.3 faz uso de *EqNode::value()* que retorna o valor atual do nó, portanto é possível que este valor esteja desatualizado. Por este motivo é importante que se atualize o valor do nó antes do cálculo das derivadas através deste algoritmo. Fato que sempre ocorre no sistema do simulador, pois primeiramente são calculados os resíduos das funções através de *EqNode::getValue()* (atualiza o valor dos nós) para o posterior cálculo do jacobiano.

Embora o método implementado na função *EqNode::getDerValue()* seja um método de AD direto, o mesmo pode ser convertido para um método reverso através da gravação prévia de uma "fita", como apresentado em Griewank *et al*. (1999).

4.3.5. Derivação Simbólica

A derivação simbólica consiste na manipulação simbólica de uma expressão para a obtenção da **expressão** de sua derivada. A derivação simbólica deve ser preferida quando não se deseja apenas o valor da derivada, mas sim sua expressão, como no caso da redução de índice de sistemas algébrico-diferenciais (Seção 5.3.2).

Existem alguns sistemas para manipulações de expressões algébricas tais como REDUCE (Hearn, 1999) e MAPLE (MapleSoft, 2002) que implementam métodos de

diferenciação simbólica. Estes sistemas são projetados para serem utilizados de forma interativa, onde o usuário entra com expressões sobre as quais pode fazer diversas operações (manipulação, diferenciação, integração, visualização, etc), embora o MAPLE forneça uma biblioteca para a utilização direta por programas externos. Estes sistemas apresentam a desvantagem de não serem de domínio público.

Por estes motivos, novamente uma implementação própria de diferenciação simbólica aparece como a melhor alternativa. Com uma equação representada na forma proposta na Seção 4.3.1 a implementação de um algoritmo de diferenciação simbólica se torna simples, como representado no Algoritmo 4.4.

Algoritmo 4.4 Derivação simbólica.

EqNode derive(in orig EqNode, in var const EqNode *) **Derivação simbólica.**

Entrada: nó original e referência para o nó em relação ao qual se está derivando.

Saída: nó da expressão derivada da equação original em relação ao nó *var*.

1. Se *orig* é um nó folha, retornar um novo nó representando a derivada de *orig* em relação à *var*.
2. Se este é um nó binário ou unário, criar um novo nó que represente a respectiva regra de derivação, por exemplo:
   - Se *orig.op* é o operador *, retornar um novo nó que represente a expressão: *orig.left****derive(****orig.right,var****)*** *+* ***derive(****orig. left,var****)*****orig.right;*
   - Se *orig.op* é a função exponencial, retornar um novo nó que represente a expressão: *orig.left****derive(****orig.left,var****);*

Assim, para a obtenção da derivada simbólica de uma determinada expressão, basta a aplicação do Algoritmo 4.4 ao nó raiz da árvore representando esta e o algoritmo recursivo retornará o nó raiz da expressão da derivada. Pode-se observar que o método de derivação simbólica é muito semelhante ao método de derivação automática, sendo que no primeiro método expressões simbólicas são propagadas na árvore, enquanto que no segundo apenas os valores numéricos se propagam.

4.3.6. Derivação Simbólico-Numérica

A obtenção da derivada por derivação automática ou simbólica, conforme apresentado nas Seções 4.3.4 e 4.3.5, requer que a equação que se quer derivar esteja na forma de árvore (Seção 4.3.1). Muitas vezes um simulador pode fazer uso de funções ou pacotes externos, os quais fornecem apenas o valor de resíduos de funções e algumas vezes valores de gradientes. Desta forma, a derivação de uma equação externa ao simulador pela aplicação dos algoritmos apresentados fica inviabilizada.

Para contornar este problema, a derivação automática deve ser substituída pela derivação numérica por perturbação e a derivação simbólico-numérica pode ser uma forma alternativa para a obtenção de uma expressão para a derivada. Este método utiliza-se da regra da cadeia e apenas do valor numérico de derivadas parciais (que são fornecidos diretamente pela rotina externa ou obtido por perturbação numérica). A descrição deste método é apresentada abaixo.

Considere-se um conjunto de funções externas ao simulador da seguinte forma:

$$F_{ext}(x,x') = 0 \tag{4.2}$$

Utilizando-se da regra da cadeia, a derivada com relação à variável independente da q-ésima equação deste sistema é:

$$f_q' = \sum_{i=1}^{n} \frac{\partial f_q}{\partial x_i} x_i' + \sum_{i=1}^{n} \frac{\partial f_q}{\partial x_i'} x_i'' \tag{4.3}$$

Assim, tem-se uma expressão para a derivada de uma função a qual não se conhece a estrutura, podendo ser externa ao simulador. Para sistemas compatíveis com CAPE-OPEN (CO 1999) a aplicação desta metodologia fica facilitada, uma vez que sistemas externos compatíveis com este padrão devem fornecer o valor das derivadas parciais.

Deve ser lembrado que o valor das derivadas parciais deve ser avaliado no ponto em que se quer conhecer a derivada e a expressão obtida pela aplicação de (4.3) é válida somente neste ponto. Derivadas de maior ordem podem ser obtidas pela subseqüente aplicação da regra da cadeia em (4.3).

Normalmente pacotes ou funções externos não envolvem a derivada das variáveis, reduzindo (4.2) a $F_{ext}(x) = 0$, anulando o segundo termo de (4.3). Este fato gera grandes simplificações principalmente se são necessárias derivadas de ordem superior.

4.4. Sistemas de Equações

A estrutura interna do simulador (Seção 4.1) propõe a conversão da descrição de um processo dinâmico em dois tipos de sistemas: sistema de equações não lineares (*non-linear algebraic*, NLA) e sistema de equações algébrico-diferenciais (*differential-algebraic equations*, DAE). A solução destes sistemas por métodos tradicionais requer que um terceiro tipo de sistema deva ser resolvido: sistema de equações lineares (*linear algebraic*, LA).

Estes sistemas de equações apresentam algumas características comuns e podem ser representados por uma estrutura hierárquica na qual todos os sistemas derivam de uma classe

base. Esta classe base representa um sistema abstrato de equações (*equation set object*, ESO). O diagrama de classes para os diferentes sistemas de equações é apresentado na Figura 4.5. Na Figura 4.6 pode ser visto o diagrama de classes para os *solvers* numéricos.

Esta estrutura é baseada no pacote numérico de interfaces do projeto CAPE-OPEN, com algumas alterações nas chamadas das funções objetivando tornar o sistema mais conciso e simplificado, porém sem perda de generalidade. Uma análise completa sobre o conjunto de interfaces para o compartilhamento de *solvers* entre simuladores de processo pode ser encontrada em Soares e Secchi (2002b, em anexo).

Os diagramas das figuras 4.6 e 4.7 apresentam algumas particularidades por se tratarem de diagramas de classes para interfaces. Interfaces são representações puramente descritivas e apenas apresentam quais são as funções que os objetos provêm aos seus clientes (OMG, 1999). Nos diagramas de classe as interfaces se diferenciam por apresentar um círculo na parte superior direita do bloco e por não apresentar a seção de atributos, uma vez que uma interface contém apenas funções e não dados.

Nesta arquitetura, dado um sistema de equações ESO, pode-se criar um *solver* para este através da função *ICapeNumericSolverManager::CreateSolverForProblem()*. Os objetos que representam sistemas de equações fornecem as informações para a sua solução, tais como número de equações, variáveis, valor dos resíduos e jacobiano. Este diagrama de classes facilita a implementação de sistemas de computação distribuída, pois o sistema de equações e seu respectivo *solver* podem ser executados em processos separados. A classe *ICapeDAESolver* faz uso de eventos os quais são apresentados na Seção 4.6.

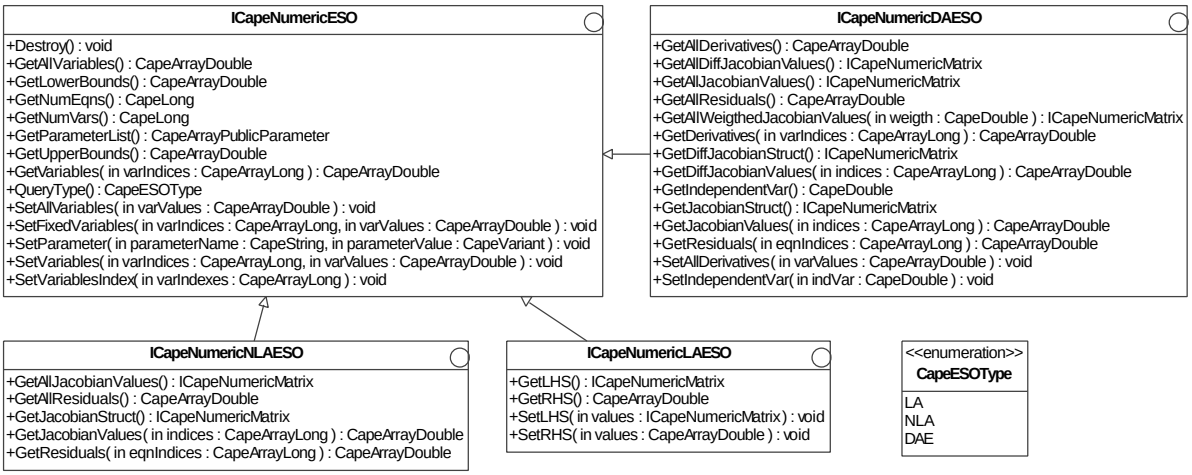


Figura 4.5 Diagrama de interfaces para representação de sistemas de equações.

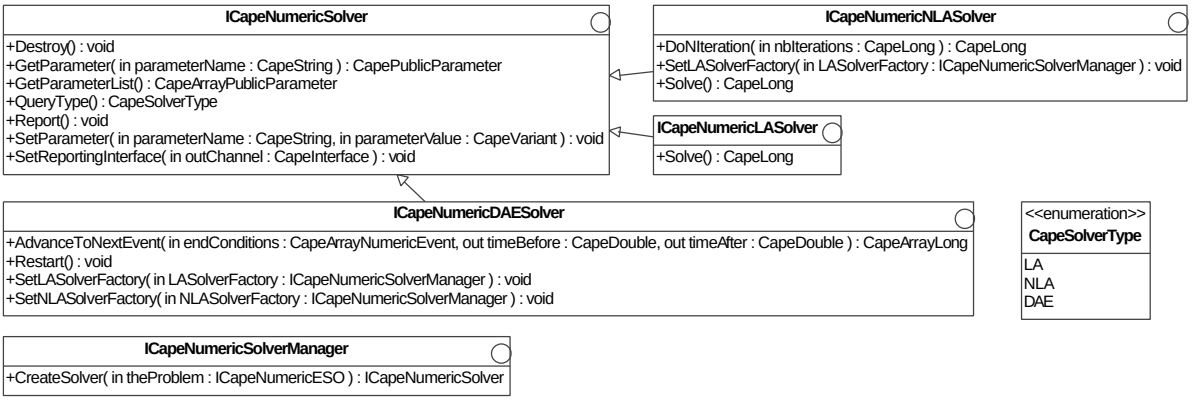


Figura 4.6 Diagrama de interfaces para representação de *solvers* numéricos.

4.4.1. Simulação Estacionária - Sistemas NLA

Sistemas estacionários são descritos por um sistema de equações não-lineares (NLA) e uma simulação estacionária consiste na sua solução. A solução de tais sistemas por métodos tradicionais (tipo Newton) requer o conhecimento do valor dos resíduos de todas as equações do sistema assim como o valor do jacobiano.

Na estrutura proposta na Figura 4.5, a solução de problemas NLA se dá pela utilização de duas classes: *ICapeNLAESO* (representando o problema) e *ICapeNLASolver* (representando a rotina matemática capaz de resolver um problema NLA). Assim, tendo-se em mãos um objeto de cada uma destas classes pode-se fazer com que o *ICapeNLASolver* resolva o *ICapeNLAESO* por meios da função *ICapeNLASolver::Solve()*. Durante a solução, o *solver* fará uso das funções membro *GetAllResiduals()* e *GetAllJacobianValues()* do objeto que representa o problema NLA.

A implementação da função *ICapeNLASolver::solve()* pode ser feita através da utilização de métodos tipo Newton. Existem diversos *solvers* deste tipo em domínio público, tais como: BRENT, HYBRD and DSOS (http://www.netlib.org).

É importante lembrar que, na estrutura proposta, a solução de sistemas NLA não ocorre apenas em simulações estacionárias, mas também na inicialização e reinicialização de sistemas DAE, como demonstrado a seguir.

4.4.2. Simulação Dinâmica – Sistemas DAE

Uma simulação dinâmica (representada pelo grupamento *Simulação Dinâmica* na Figura 4.2) nada mais é do que a integração de um sistema algébrico-diferencial em relação à sua variável independente. Para a integração deste tipo de sistema de equações de um ponto

inicial até outro é necessário conhecer o valor de todas as variáveis e de suas derivadas no ponto inicial.

A etapa, que precede a integração, na qual se determina o valor de todas as variáveis e de suas derivadas no ponto inicial, é denominada de inicialização. A determinação destes valores pode ser feita de diversas maneiras, as mais relevantes são: aproximações BDF (*Backward difference formulae*) (Brenan *et al*.,1989); métodos implícitos tipo Runge-Kutta (Hairer e Wanner, 1991); extrapolações baseadas no método de Euler (Deuflhard *et al*., 1987); diferenciações e posterior solução de um sistema de equações não-lineares formado pelo sistema original, derivadas deste e as condições iniciais (Soares e Secchi, 2002a, em anexo; Costa Jr. et al., 2001).

Todos estes métodos, com exceção do último, envolvem aproximações e apresentam dificuldades quando do tratamento de sistemas DAE de índice elevado. Este trabalho se utilizará do último método citado (solução de um problema NLA, como apresentado na Seção 4.4.1) por ser um método mais robusto e ser mais adequado à sistemas DAE de índice elevado (Soares e Secchi, 2002a, em anexo).

Uma vez resolvido o problema de inicialização do sistema, pode-se conduzir a integração do mesmo. Se durante a integração, vier a ocorrer qualquer evento que gere uma descontinuidade no sistema os valores das variáveis e suas derivadas podem ter deixado de ser consistentes com o sistema, fato que impede o avanço do integrador.

No caso de uma descontinuidade (detalhada na Seção 4.6), a integração de um sistema algébrico-diferencial só deve ser prosseguida após a reinicialização do mesmo. Também para este caso, o método escolhido foi a solução de um sistema NLA constituído por todas as equações de descrição do processo juntamente com equações de junção que relacionam o sistema antes e depois da descontinuidade.

Então, para a simulação dinâmica de um processo, o simulador deve construir um sistema não-linear (*ICapeNLAESO*) para a inicialização e um sistema algébrico-diferencial para a integração (*ICapeDAESO*), além da geração de novos sistemas NLA no caso da ocorrência de descontinuidades.

De forma análoga à solução de problemas NLA, o diagrama de classes da Figura 4.5 propõe a solução de problemas DAE pela utilização de duas classes: *ICapeDAESO* (representando o problema) e *ICapeDAESolver* (representando a rotina matemática capaz de resolver um problema DAE). Assim, tendo-se em mãos um objeto de cada uma destas classes pode-se fazer com que o *ICapeDAESolver* resolva o *ICapeDAESO* por meio da função membro *advanceToNextEvent()*. Durante a solução, o *solver* fará uso das funções membro *GetAllResiduals()* e *GetAllJacobianValues()* do objeto que representa o problema DAE.

Para a implementação da função *ICapeDAESolver::advanceToNextEvent()*, algum pacote de integração tal como DASSLC (Secchi e Pereira, 1997) pode ser utilizado, sendo necessária apenas a implementação de um sistema para a detecção e determinação do ponto de ocorrência de eventos (Seção 4.6).

4.4.3. Sistemas Lineares - LA

Sistemas lineares (*linear algebraic*, LA) surgem durante a solução de sistemas NLA e DAE. A solução destes sistemas é parte da álgebra linear e divide-se basicamente em dois grandes grupos: métodos diretos e iterativos.

A álgebra linear é uma área já bem compreendida pela comunidade científica e diversas implementações de domínio público que tratam deste assunto estão disponíveis. Estas podem ser encontradas tanto para linguagens orientadas a procedimentos: LAPACK (http://www.netlib.org/lapack/), BLAS (Lawson *et al.*, 1979, http://www.netlib.org/blas), como para linguagens orientadas a objetos: TNT (http://math.nist.gov/tnt/), SparseLib++ (Pozo *et al.*, 1996), IML++ (Dongara *et al.*, 1996), etc.

Neste trabalho o tratamento de sistemas LA ficará restrito à utilização de bibliotecas já consagradas e bem estabelecidas, necessitando apenas a sua adaptação à interface proposta no diagrama de classes da Figura 4.5.

4.4.4. Otimização Estacionária

Embora este trabalho trate do projeto de um *software* para a utilização preferencial de problemas dinâmicos, a solução de problemas estacionários assim como a otimização de tal categoria de problemas também é tratada.

Sistemas estacionários são descritos por um sistema de equações NLA (Seção 4.4.1) e uma otimização estacionária envolve a minimização de uma função objetivo sujeita à restrições. Em um simulador, dentre estas restrições está o próprio sistema NLA proveniente da descrição do processo em estudo (*FlowSheet*, Seção 3.3.2), da seguinte forma:

$$\min_{v} S(v) \tag{4.4}$$

sujeito a:

$$h(v) = \begin{bmatrix} F(y,v) \\ \tilde{h}(v) \end{bmatrix} = 0 \tag{4.5}$$

$$g(v) \geq 0 \tag{4.6}$$

onde, v é o vetor de parâmetros de otimização, y é o vetor das variáveis do modelo, S é a função objetivo a ser minimizada, h é o conjunto de restrições de igualdade (contendo o modelo do sistema, F, e demais restrições de igualdade, $\tilde{h}$) e g é o conjunto de restrições de desigualdade.

A solução numérica do problema (4.4)-(4.6) pode ser obtida pela utilização de alguma biblioteca numérica de otimização para problemas não-lineares com restrições. Na seção a seguir são apresentados alguns pacotes de domínio público que podem ser utilizados com este intuito.

4.4.5. Otimização Dinâmica

Um problema de otimização dinâmica resume-se na minimização de uma função objetivo sujeita à um conjunto de restrições, envolvendo equações diferenciais. No caso da otimização dinâmica em um simulador, entre as restrições impostas à função objetivo está o modelo do sistema (sistema DAE proveniente do *FlowSheet*, Seção 4.4.2). Uma demonstração estrutural das entidades envolvidas em uma otimização dinâmica pode ser observada no grupamento *Otimização Dinâmica* da Figura 4.2 e uma definição formal do problema segue:

$$\min_{x,v,t_f} S(x, v, t_f) \tag{4.7}$$

sujeito a:

$$h(x, v, t) = \begin{bmatrix} F(t, y, y', x, v) \\ \tilde{h}(x, v, t) \end{bmatrix} = 0 \tag{4.8}$$

$$g(x, v, t) \geq 0 \tag{4.9}$$

onde, $t \in [t_0, t_f]$ é a variável independente, t_0 seu valor inicial e t_f seu valor final, x é o vetor de variáveis de otimização (dependentes de t), v é o vetor de parâmetros de otimização (independentes de t), S o funcional a ser minimizado, h é o conjunto de restrições de igualdade, do qual faz parte o modelo do problema F (sistema DAE proveniente do *FlowSheet*, Seção 4.4.2), e g são restrições de desigualdade.

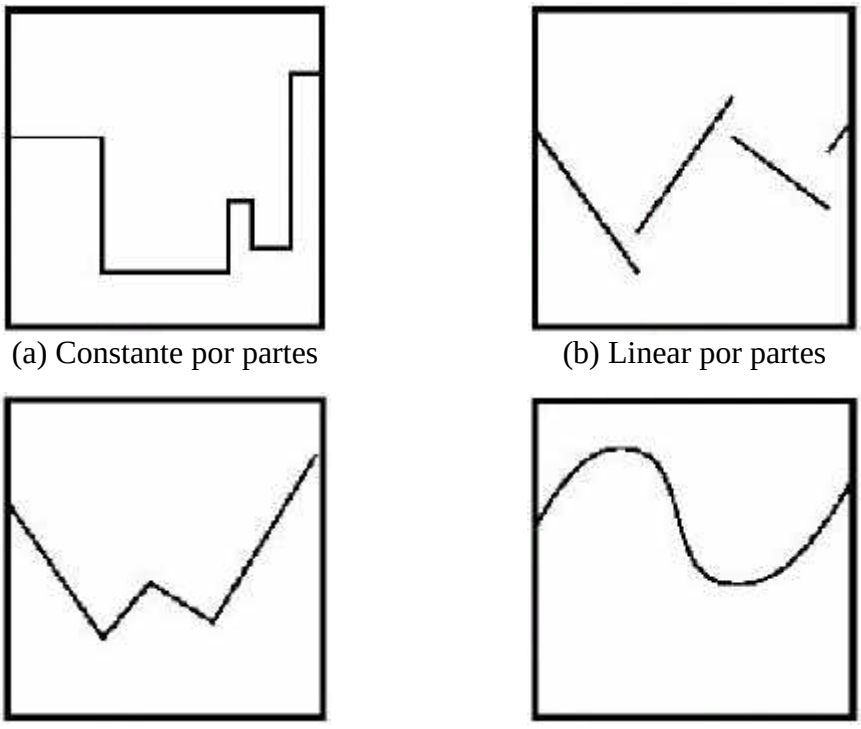
Em problemas de otimização dinâmica o vetor x é uma função de t e S é um funcional de x. Para os casos em que x envolve apenas variáveis operacionais do problema em estudo, o problema de otimização dinâmica é denominado de *controle ótimo* (Secchi, 2002).

É comum a presença de termos integrais sobre todo o intervalo $[t_0, t_f]$ no funcional S, da seguinte forma:

$$\Phi(x, v, t_f) = \int_{t_0}^{t_f} j\,(t, y, y', x, v)dt \tag{4.10}$$

Um sistema genérico de solução deve permitir que termos da forma de (4.10) estejam envolvidos no funcional *S* por meio de qualquer operação algébrica.

Para que seja possível a obtenção da solução do problema (4.7)-(4.9), primeiramente é necessário que se escolha uma forma para as variáveis de controle *x*, uma vez que estas são uma função de *t*. Os principais tipos de variações podem ser vistos na Figura 4.7, sendo que os tipos (a) e (b) são mais freqüentemente utilizados em *controle ótimo* de processos.



(a) Constante por partes (b) Linear por partes

(c) Linear por partes contínuas (d) Polinomial

Figura 4.7 Tipos de variações para variáveis de controle em problemas de otimização dinâmica.

Método Single-Shooting

O método conhecido como método seqüencial ou *single-shooting* consiste na aproximação do problema de otimização dinâmica em um problema de programação não-linear (*Nonlinear Programming*, NLP) através da parametrização das variáveis de controle *x* (Binder *et al.*, 2000). Esta parametrização se dá pela escolha da forma das variáveis de controle (Figura 4.7) fazendo com que $x = f(\tilde{v})$, onde $\tilde{v}$ são novos parâmetros de otimização invariantes com o tempo. Desta forma, o problema passa a ter apenas parâmetros de otimização (v e $\tilde{v}$) e a função objetivo e os gradientes podem ser avaliados através de integração numérica do sistema *F*, juntamente com perturbações nos parâmetros de otimização ou análises de sensibilidade.

Então, a solução de um problema de otimização dinâmica através do método *single-shooting* se resume na aplicação dos seguintes passos:

Passo 1: Escolher um valor inicial para todos os parâmetros de otimização v e $\tilde{v}$;

Passo 2: Integrar o sistema F de t_0 até t_f;

Passo 3: Calcular a função objetivo S e as restrições h e g;

Passo 4: Corrigir os parâmetros de otimização v e $\tilde{v}$ através da utilização de algum método NLP;

Passo 5: Verificar a convergência de v e $\tilde{v}$. Caso seja satisfeita a otimização está terminada, caso contrário retornar ao Passo 2.

Para que o método *single-shooting* possa ser utilizado em casos nos quais a função objetivo envolve termos do tipo (4.10) é necessário que se crie, para cada um dos termos integrais, um novo estado Φ e que a seguinte equação seja adicionada ao sistema:

$$\dot{\Phi} = j\,(t, y, y', x, v) \tag{4.11}$$

juntamente com a condição inicial $\Phi(t_0) = 0$. Desta forma, ao final de cada integração $\Phi(t_f)$ fornece diretamente o valor do termo integral que este representa.

Embora o método *single-shooting* não gere problemas de dimensões elevadas, é sabido que este método sofre deficiências de estabilidade e robustez (Ascher *et al.*, 1995).

Método da Discretização

Uma alternativa para a solução de problemas de otimização dinâmica é a discretização do sistema no intervalo [t_0, t_f] por métodos como: diferenças finitas, elementos finitos, volumes finitos, etc. Devido à discretização do sistema, a presença de termos integrais no funcional S não gera a necessidade da criação de novos estados, uma vez que seu valor pode ser calculado pela utilização de alguma fórmula de quadratura.

Embora este método seja mais estável e robusto, quando comparado ao método de *single-shooting*, o mesmo requer a solução de um problema de otimização que pode ter dimensões inviáveis se o problema F for de grande dimensão. Além disto, a utilização de discretização adaptativa fica inviabilizada para a utilização neste método (Gill *et al.*, 2000).

Método de Multi-Shooting

No método conhecido como *multi-shooting* o intervalo $[t_0, t_f]$ é dividido em subintervalos $[t_i, t_{i+1}]$ ($i = 1,\ldots,N$), e o sistema algébrico-diferencial F é resolvido independentemente em cada um dos subintervalos, onde são introduzidas as variáveis intermediárias y_i.

Em cada subintervalo a solução y de F, com valor inicial y_i em t_i é denominada $y(t, t_i, y_i, v)$. A continuidade é garantida pela adição de *restrições de continuidade* $y_{i+1} = y(t_{i+1}, t_i, y_i, v)$ entre os subintervalos. Desta forma, o problema de otimização dinâmica pode ser escrito como o seguinte problema NLP:

$$\min_{y_i, x, v, t_f} S(y_i, x, v, t_f) \tag{4.12}$$

sujeito a:

$$y_{i+1} - y(t_{i+1}, t_i, y_i, v) = 0 \tag{4.13}$$

$$\tilde{h}(v, t_i) = 0 \tag{4.14}$$

$$g(v, t_i) \geq 0 \tag{4.15}$$

Este problema pode ser resolvido por algum código de otimização não-linear. Estes códigos normalmente requerem o cálculo de gradientes da função objetivo e de suas restrições. O valor destas derivadas pode ser obtido por algum método de cálculo de sensibilidade de sistemas algébrico-diferenciais (Maly e Petzhold, 1996).

Embora o método de *multi-shooting* seja mais estável, robusto e apresente uma forma passível de implementação paralela, este requer a solução de problemas de sensibilidade da ordem do número de variáveis y vezes o número de subintervalos N. Já para o método *single-shooting*, estes cálculos são da ordem do número de parâmetros de otimização $v + \tilde{v}$ (Gill *et al.*, 2000). Assim, o método de *multi-shooting* deve ser considerado para a solução de problemas de otimização de dimensões pequenas a moderadas, quando implementado de forma seqüencial.

Métodos para a solução de problemas NLP

Foi visto que, tanto na solução de problemas de otimização estacionária (Seção 4.4.4) quanto nos três métodos apresentados para a solução de problemas de otimização dinâmica, recaiu-se em um problema NLP. Esta categoria de problemas pode ser resolvida por diversos métodos, dentre os quais destacam-se os quadráticos seqüenciais (*Sequential Quadratic*

Programming, SQP). Existem diversos pacotes de otimização em domínio público, os quais podem ser utilizados para a solução de problemas NLP, tais como OPT++ (Meza, 1994), COOOL (Deng *et al.*, 1994) e HQP (Franke, 1998).

4.5. Matrizes Multidimensionais

Cada linguagem de programação apresenta um diferente nível de abstração (Seção 3.1). No Capítulo 3 foi apresentada uma nova linguagem de alto nível de abstração, a qual tem como suas entidades básicas, variáveis (no sentido matemático), equações e dispositivos. Além disto, sistemas descritos na linguagem projetada podem envolver matrizes destes tipos básicos. A implementação de tais matrizes é feita por uma classe de matrizes multidimensionais, conforme o diagrama de classes da Figura 4.8.

No caso de um simulador, são necessárias matrizes de variáveis, equações e dispositivos. Uma forma de implementação seria a criação de uma classe diferente para a representação de cada um destes tipos de matrizes. Outra forma, bem mais eficiente, do ponto de vista de desenvolvimento de código, é a criação de uma única classe representando uma matriz de um tipo indeterminado, uma *template class*. Na Figura 4.8 este tipo indeterminado é indicado pelo identificador “T”, que pode ser substituído por qualquer outra classe.

Nos diagramas de classes, um retângulo tracejado sobreposto na parte superior direita do bloco da classe (Figura 4.8) indica que esta é uma *template class* e os identificadores internos a este retângulo são os parâmetros do *template.* Uma *template class* tem parâmetros indeterminados que fornecem uma grande facilidade na reutilização de código.

O modelo proposto na Figura 4.8 para a representação de matrizes tem um tipo base chamado *ArrayAny*, uma classe abstrata que define uma interface básica para qualquer outro tipo de matriz multidimensional. Todas as outras classes são derivadas desta classe base: *Array* que representa realmente uma matriz de dados do tipo T e *ArrayRef* que apenas contém referências à variáveis do tipo T, sem conter realmente os dados. Uma matriz de referências é gerada por uma matriz real através da função membro *operator()* que utiliza-se da classe *Range* como parâmetro de entrada. Este conjunto de classes forma um sistema que, na sua utilização, se assemelha ao Matlab (Mathworks, 2002).

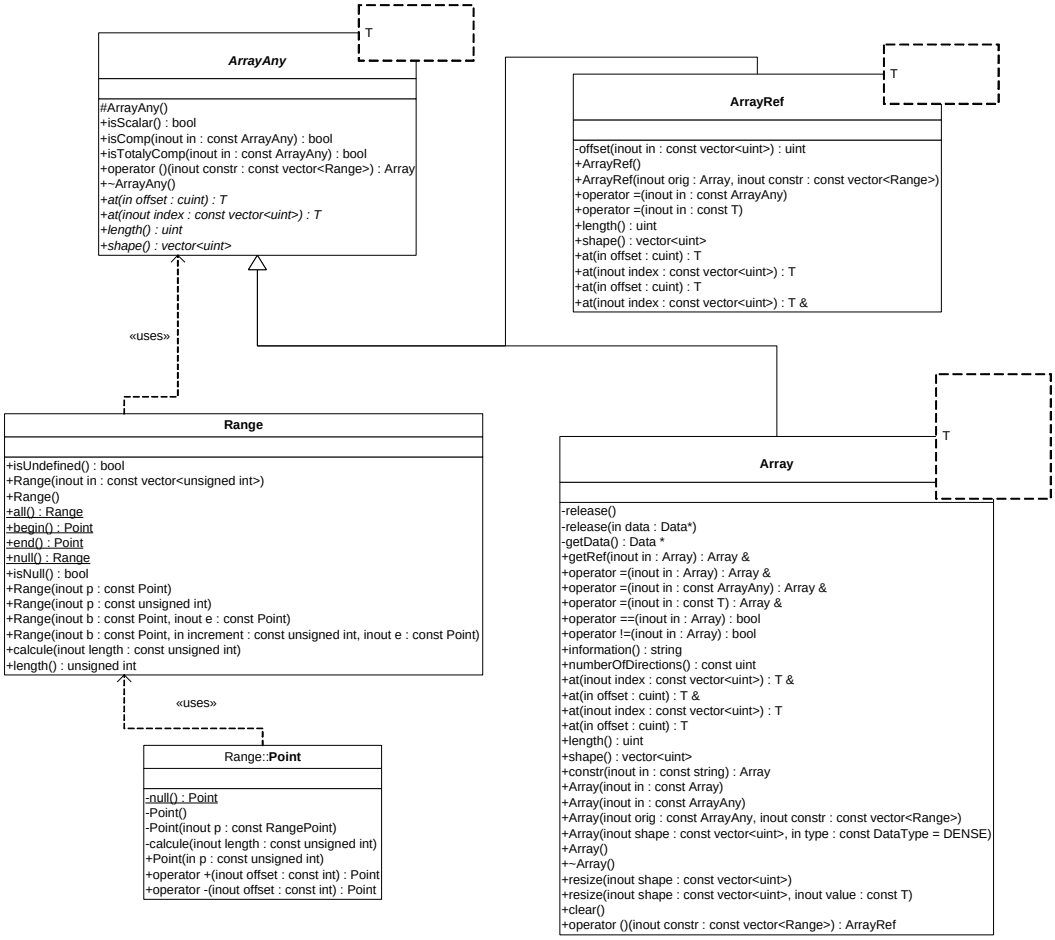


Figura 4.8 Diagrama de classes para representação de matrizes multidimensionais.

4.6. Eventos

No transcorrer de uma simulação dinâmica (solução de um DAE) podem surgir situações nas quais ocorre uma modificação no sistema de equações. Aos fatos que determinam estas situações chama-se de eventos. Os eventos podem ser caracterizados como: dependentes do tempo (variável independente) ou dependentes dos estados do sistema (variáveis dependentes).

Um exemplo simples de um evento dependente do tempo seria o fechamento de uma válvula após determinado tempo de operação. Um exemplo de um evento dependente dos estados seria a ruptura de uma válvula de segurança em determinada pressão.

A importância no tratamento dos eventos vem da descontinuidade que os mesmos geram no sistema de equações. Há muitos exemplos de fenômenos que são comumente descritos de uma forma descontínua, ou seja, que envolvem eventos. Entre eles pode-se citar:

- Aparecimento e desaparecimento de fases termodinâmicas;

- Transição entre regimes de fluxo (por exemplo, de laminar para turbulento);
- Mudanças na direção de fluxo e suas conseqüências;
- Descontinuidades na geometria dos equipamentos (por exemplo, abertura de uma linha de fluxo alternativa).
- Saturação de equipamentos ou a sua quebra;
- Ações discretas de controle ou outros tipos de distúrbios impostos por agentes externos.

Em se tratando de sistemas algébrico-diferenciais uma descontinuidade pode gerar uma inconsistência entre os valores das variáveis, suas derivadas e o sistema de equações. Esta inconsistência impossibilita o avanço da solução numérica em um integrador de passos múltiplos (caso do integrador DASSLC, Secchi e Pereira, 1997). Assim, sempre da ocorrência de um evento, este deve ser detectado e a reinicialização do sistema deve ser feita antes do avanço da solução.

Para a manipulação de eventos é proposta uma estrutura composta por duas classes, uma para eventos dependentes do tempo e outra para dependentes de estados. O diagrama de classes desta estrutura pode ser visto na Figura 4.9.

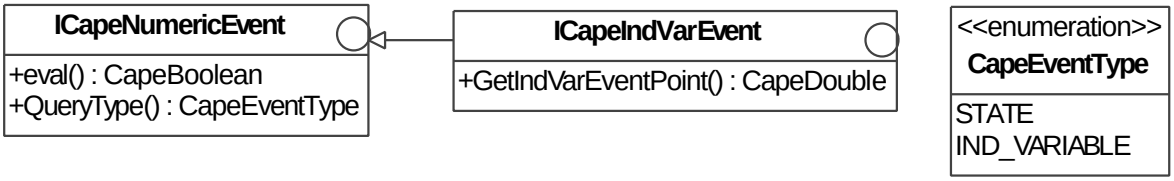


Figura 4.9 Diagrama de classes para eventos.

4.6.1. Eventos dependentes de Estados

No sistema em desenvolvimento os objetos para representação de eventos dependentes de estados são do tipo *ICapeNumericEvent*. Esta classe oferece uma interface bem simples, composta por apenas duas funções:

- *QueryType():* retorna o tipo de evento (um dos valores do enumerador *CapeEventType*);
- *eval():* avalia o evento nas condições atuais, retorna verdadeiro se o evento ocorreu e falso no caso contrário;

A manipulação dos eventos se resume à utilização destas funções. Cada evento tem relacionado a si alterações a serem feitas no sistema. O integrador deverá monitorar o valor retornado pela função *eval()* durante a integração do sistema, enquanto este valor for falso (*false*) nada precisa ser feito. No momento que for detectado um valor verdadeiro (*true* retornado pela função *eval()*) para algum dos eventos, significa que o mesmo ocorreu. A função *eval* apenas notifica o *solver* de que o evento ocorreu, mas não aplica a descontinuidade ou ação a ele associada.

Uma vez detectada a ocorrência de um evento dependente de estados, o integrador deverá localizar o ponto exato (dentro da tolerância ajustada pelo usuário) onde este ocorreu. Quando encontrado o ponto onde se dá o evento, o integrador deverá retornar o controle ao seu cliente para que as ações relativas ao evento sejam tomadas.

Existem poucas implementações de integradores que apresentam algum sistema de detecção de eventos. O código mais conhecido é o DDASRT (Netlib, 2002), que é uma derivação do já consagrado DASSL (Brenan *et al*., 1989).

Neste trabalho, foi proposta a utilização do DASSLC como código para a integração de sistemas DAE (Seção 4.4.2). Esta biblioteca numérica não apresenta um sistema de detecção de eventos, portanto um novo esquema de localização de eventos foi desenvolvido e implementado para este integrador, como apresentado a seguir.

Em integradores baseados em BDF as fórmulas de interpolação utilizadas são válidas em todo o intervalo interior a dois passos de integração executados com sucesso. Desta forma, a cada passo na evolução da solução pode-se testar a ocorrência de algum evento. Se algum evento ocorreu, este está garantidamente no intervalo entre os dois últimos passos. Assim, a detecção de eventos pode ser feita através da utilização de métodos como a bisseção, seção áurea ou busca de Fibonacci, pela aplicação do seguinte procedimento:

Passo 1: Gravar como t_1 o valor da variável independente no último passo de integração e t_2 o seu valor atual. Se nenhum evento foi detectado em t_2 pular para o Passo 3, caso contrário seguir no Passo 2.

Passo 2: Se o intervalo (t_1, t_2) é menor que a precisão desejada, o ponto de ocorrência dos eventos está determinado (t_2) e o procedimento terminado. Caso contrário, escolher um valor para t_{in} interno ao intervalo (t_1, t_2) (bisseção, seção áurea ou Fibonacci). Interpolar o valor para as variáveis dependentes do sistema em t_{in} utilizando-se das formulas de interpolação do próprio método de integração. Testar a ocorrência de eventos em t_{in}, se algum evento foi detectado fazer $t_2 = t_{in}$, caso contrário fazer $t_1 = t_{in}$. Retornar ao Passo 2.

Passo 3: Avançar na solução do sistema dando mais um passo de integração. Retornar ao Passo 1.

4.6.2. Eventos dependentes do Tempo

Os eventos dependentes do tempo são representados na Figura 4.9 pela classe *ICapeIndVarEvent*. Esta classe é derivada da classe *ICapeNumericEvent* e oferece a função extra *GetIndVarEventPoint()* a qual fornece diretamente o valor da variável independente (o tempo) em que o evento ocorre.

A vantagem dos eventos dependentes da variável independente é a simplicidade no seu tratamento, uma vez que o tempo em que ocorrem já é conhecido. Estes eventos têm comportamento análogo ao explicado na Seção 4.6.1, com a simplificação de que não é necessária a utilização de método algum para determinação do ponto em que eles ocorrem. Para estes eventos o integrador pode, antes de iniciar a integração, verificar em que tempos terá que parar para a reportagem de eventos ao seu cliente.

4.7. Conclusão

Este capítulo apresenta conceitualmente as metodologias e os diagramas de classes necessários para a implementação de um simulador genérico de processos dinâmicos, envolvendo: conversão da linguagem proposta no Capítulo 3 para equações executáveis e diferenciáveis (avaliação de resíduo, teste de dependência, diferenciação automática e simbólica), simulação e otimização estacionária e dinâmica. A estrutura de classes proposta apresenta uma grande modularidade, possibilitando o compartilhamento de objetos com sistemas externos e o desenvolvimento de forma concorrente de cada um dos módulos, além de propiciar uma grande facilidade de atualização independente de cada um dos subsistemas.

A estrutura é muito semelhante ao padrão original de interfaces numéricas do projeto CAPE-OPEN e se apresenta na forma ideal para a sua implementação em alguma linguagem orientada a objetos. Sendo este o caso, objetos tais como *ICapeNLAESO*, *ICapeDAESolver*, entre outros, podem ser compartilhados entre processos sendo executados em diferentes máquinas de diferentes arquiteturas através da utilização de um *middleware* como o omniORB (Grisby *et al.*, 2002).

A arquitetura dos pacotes numéricos só não é idêntica à proposta pelo projeto CAPE-OPEN por motivos de consistência e simplicidade (Soares e Secchi, 2002b, em anexo). Espera-se que as modificações propostas sejam adotadas como padrão pelo projeto CAPE-OPEN.

Capítulo 5 Análises de Consistência

No Capítulo 4 foi demonstrado que, de forma simplista, um simulador pode ser encarado como uma ferramenta de tradução aliada a um conjunto de pacotes numéricos. Tal ferramenta faz a tradução da descrição de um processo em alguma linguagem de modelagem (Seção 3.3) para sistemas de equações (Seção 4.4). Após a conversão, o simulador utiliza-se de pacotes matemáticos, próprios ou externos, para a análise e obtenção da solução dos sistemas de equações gerados.

É sabido que a aplicação de análises antes da solução numérica dos sistemas de equações revela grande parte das falhas que viriam a ocorrer. Existem diversos tipos de análises de consistência que podem ser aplicadas aos sistemas de equações envolvidos em um simulador de processos dinâmicos, entre eles pode-se citar a consistência das unidades de medida, resolubilidade estrutural e consistência de condições iniciais. Este capítulo apresenta uma série de análises de consistência, juntamente com os métodos utilizados para sua aplicação.

5.1. Consistência de Unidades de Medida

Em problemas de engenharia, a tarefa de conversão de unidades de medidas é tediosa e fortemente sujeita a erros. Além disto, inconsistências nas unidades de medida de uma ou mais equações de um sistema podem revelar erros na descrição do problema. Por estes motivos uma ferramenta que verifique a consistência de unidades de medida e execute automaticamente as conversões necessárias torna-se valiosa.

Para equações convertidas para a forma proposta na Seção 4.3.1 a análise de consistência pode ser feita pela aplicação do Algoritmo 5.1 ao nó raiz da equação. Este algoritmo retorna a unidade de medida de um nó e é análogo ao Algoritmo 4.1 que retorna o valor numérico de um nó.

Algoritmo 5.1 Análise de consistência de unidades de medida de um nó de uma equação.

EqNode::getUnit( ) **Unidade de medida de um nó.**

Saída: unidade de medida do nó.

1. Se este é um nó folha retornar a sua unidade de medida (por exemplo, um valor numérico retorna unidade adimensional e uma variável retorna sua unidade de medida);
2. Se este é um nó binário ou unário, testar a consistência e retornar a unidade resultante, por exemplo:
    - Se *op* é o operador +, testar se as unidades retornadas por *left.getUnit()* e *right.getUnit()* são consistentes e então retornar a resultante da soma*;*
    - Se *op* é a função exponencial, testar se *left.getUnit()* é adimensional e retornar uma unidade adimensional*;*

A implementação do Algoritmo 5.1 requer testes de consistência e operações algébricas entre unidades de medida. Isto pode ser feito de forma enxuta se for utilizada uma biblioteca para manipulação das unidades de medida. Uma implementação em C++ de uma classe para testes de consistência, manipulações e conversões entre unidades que pode ser utilizada para este fim é o RUnits (Soares, 2002).

5.2. Consistência de Sistemas NLA

Em um simulador de processos, sistemas não-lineares surgem na descrição matemática de modelos estacionários (Seção 4.4.1), na solução de sistemas algébrico-diferenciais (Seção 4.4.2) e em problemas de otimização.

Uma análise de consistência ou resolubilidade de sistemas não lineares $H(x) = 0$ resume-se em testar a singularidade do sistema através da checagem da singularidade do jacobiano $H_x(x)$. A singularidade desta matriz pode ser testada de forma estrutural ou numérica.

No caso de teste estrutural, pode-se resolver um problema de associação entre as variáveis e equações do problema: se todas as variáveis puderem ser associadas a equações únicas então o sistema é dito estruturalmente não singular. Quando detectada uma situação de singularidade estrutural, o simulador deve comunicar ao usuário não só a singularidade do problema, mas a razão e preferencialmente a ação corretiva que deve ser tomada. Por exemplo, comunicar ao usuário que uma das variáveis não está envolvida no sistema.

Para os casos em que o sistema é estruturalmente não singular, mas numericamente singular (jacobiano não inversível) perturbações podem ser feitas ao sistema na tentativa de que este deixe a condição de singularidade numérica.

5.3. Consistência de Sistemas DAE

Sistemas de equações algébrico-diferenciais (*differential algebraic equations,* DAE) surgem naturalmente da modelagem matemática de processos dinâmicos. Sistemas DAE são compostos por equações algébricas acopladas a equações diferenciais e podem ser equacionados por:

$$F(t, y, y') = 0 \tag{5.1}$$

onde $t \in \Re$ é a variável independente (normalmente o tempo); $F \in \Re^m$, y e $y' \in \Re^n$ são respectivamente as variáveis dependentes e suas derivadas. Nesta representação não é feita a distinção das variáveis que aparecem somente como algébricas no sistema (5.1).

O primeiro teste de consistência a ser feito em um sistema DAE é a verificação se o sistema é quadrado (graus de liberdade igual a zero), isto ocorre quando o número de variáveis *n* é igual ao número de equações *m.* Se o sistema não tem o número de graus de liberdade nulo, este não é passível de solução e tal fato deve ser reportado ao usuário.

Se um sistema DAE apresenta um número de graus de liberdade igual a zero, então pode-se prosseguir com um teste do número de graus de liberdade dinâmicos. Para sistemas DAE de índice até 1 (definido a seguir), este teste pode ser feito através da utilização do seguinte teorema proposto.

***Teorema* 5.1**: O número de graus de liberdade dinâmicos para um sistema de DAE de índice até 1 é igual ao número de equações diferenciais do sistema, que, portanto, deve ser igual ao número de condições iniciais.

Como prova deste teorema, considere-se o sistema algébrico-diferencial abaixo:

$$\begin{aligned} F(x, x', y, t) &= 0 \quad (a) \\ g(x, y, t) &= 0 \quad (b) \end{aligned} \tag{5.2}$$

onde, x são as variáveis que aparecem no sistema na forma diferencial, y as variáveis que aparecem apenas na forma algébrica, F é um sistema de n equações diferenciais e g um sistema de m equações algébricas.

Para que o sistema (5.2) tenha um número de graus de liberdade igual a zero o número de variáveis $[x, y]^T$ deve ser igual ao número de equações $n + m$. Se o sistema for de índice 1 a derivação de g faz com que se tenha $2(n + m)$ variáveis $[x, y, x', y']^T$ e $n + 2m$ equações $[F, g, g']$, pois g_y tem obrigatoriamente *rank* cheio se o sistema é de índice 1. Restando assim, n graus de liberdade a serem fornecidos como condições iniciais do problema. Note-se que não foi feita nenhuma consideração quanto a igualdade do número de variáveis y e de equações algébricas m, nem quanto a igualdade do número de variáveis diferenciais x e de equações diferenciais n.

Para o caso de sistemas de índice elevado (índice maior que 1) a análise de graus de liberdade dinâmicos não é tão simples devido à restrições que podem estar "escondidas" no sistema. O estudo para estes casos é apresentado a seguir, juntamente com os conceitos necessários à sua compreensão.

5.3.1. Índice

A propriedade conhecida como índice é de grande importância na classificação de um sistema DAE (Brenan *et al.*, 1989). A título de ilustração, considere-se o sistema DAE semi-explícito:

$$\begin{aligned} x' &= f(x, y, t) \quad (a) \\ 0 &= g(x, y, t) \quad (b) \end{aligned} \tag{5.3}$$

Se a equação algébrica (5.3b) for diferenciada com respeito à *t*, pode-se formar o novo sistema:

$$\begin{aligned} x' &= f(x,y,t) && (a) \\ g_x(x,y,t)x' + g_y(x,y,t)y' &= -g_t(x,y,t) && (b) \end{aligned} \tag{5.4}$$

Note-se que, se g_y for não singular o sistema (5.4) será um sistema ODE semi-implícito e o sistema (5.3) terá índice um, uma vez que este foi diferenciado uma vez. No caso em que g_y for singular, pode-se dizer que o sistema (5.3) tem índice elevado (superior a um), e serão necessárias manipulações algébricas e novas diferenciações para que se obtenha um sistema ODE (Brenan *et al.*, 1989). A formalização do método aplicado ao exemplo da equação (5.3) é apresentada na definição abaixo:

Definição 5.1 O **índice diferencial**, v_d, é o número mínimo de vezes que todo ou um subgrupo de equações de um sistema DAE $F(t, y, y') = 0$ precisa ser diferenciado, em relação à variável independente *t*, até ser transformado em um sistema ODE.

Definição 5.2 O **índice estrutural singular**, v_{ss}, é o número mínimo de vezes que todo ou um subgrupo de equações de um sistema DAE $F(t, y, y') = 0$ precisa ser diferenciado, em relação à variável independente *t*, para que a matriz $\partial\tilde{F} / \partial\tilde{y}'$ seja não estruturalmente singular ($\tilde{F}$ corresponde ao sistema *F* modificado pela adição das derivadas necessárias e $\tilde{y}'$ corresponde ao conjunto formado por y' e suas derivadas de ordem superior que venham a surgir nas diferenciações).

A definição de índice singular, v_s, é equivalente à Definição 5.2, porém com a substituição do critério $\partial\tilde{F} / \partial\tilde{y}'$ não estruturalmente singular por apenas não singular. A literatura apresenta outras definições de índice, todas elas têm a mesma essência, mas cada definição é baseada no seu próprio critério de resolubilidade. Referências para diferentes definições de índices de sistemas algébrico-diferenciais podem ser encontradas em Unger *et al.* (1994).

Deve-se observar que, embora o índice de um sistema DAE esteja intimamente ligado com a dificuldade na sua solução, este parâmetro de caracterização não representa uma propriedade intrínseca do sistema. Pois, qualquer sistema de índice elevado que seja passível de solução pode ser representado em outra forma de índice inferior, que, para as mesmas condições iniciais, gera uma solução idêntica.

5.3.2. Redução de Índice

Na seção anterior foi visto que, para um sistema de índice até um, a tarefa da verificação do número de graus de liberdade dinâmicos é simples. Também foi apresentada,

na forma de exemplo, uma nova metodologia para a redução de índice de sistemas DAE através da adição de novas equações (derivadas das equações originais).

A aplicação desta metodologia em forma estrutural pode ser resumida em alguns passos:

Passo 1: gerar uma lista de variáveis contendo todas as derivadas das variáveis do sistema original;

Passo 2: se cada uma das variáveis da lista puder ser associada com uma equação única (ainda não associada a outra variável), então o sistema tem índice estrutural zero e o número máximo que uma mesma equação foi derivada até o momento é o índice estrutural do sistema original e o procedimento está terminado;

Passo 3: tentar obter as variáveis da lista que não foram associadas através da diferenciação de alguma das equações. Se esta diferenciação gerar novas variáveis (derivadas de maior ordem das variáveis originais), estas devem ser adicionadas à lista. Retornar ao Passo 2.

É importante notar que, a metodologia proposta age diretamente no sentido de tornar a matriz $\partial F / \partial y'$ estruturalmente não singular e de fato este é o resultado ao final da aplicação do procedimento (caso o sistema DAE seja resolvível). Este critério é diferente do utilizado no algoritmo proposto por Pantelides (1988), o qual age no sentido de remover os conjuntos de equações minimamente estruturalmente singulares. Também deve ser lembrado que o algoritmo de Pantelides não torna a matriz $\partial F / \partial y'$ estruturalmente não singular, pois este é incapaz de reduzir o índice de sistemas DAE até zero.

A Aplicação da metodologia descrita acima determina o índice singular estrutural de sistemas DAE. As novas equações, geradas por diferenciações no Passo 3, representam restrições adicionais ao sistema. Estas equações são as responsáveis pela redução do número de graus de liberdade dinâmicos do sistema original.

Uma vez reduzido o índice de um sistema DAE até zero, pode-se determinar facilmente o número de graus de liberdade dinâmicos como sendo: o número de total de variáveis (variáveis originais, suas derivadas e as variáveis novas provenientes da redução de índice) menos o número total de equações (equações originais e as derivadas geradas na redução). O número de condições iniciais fornecidas pelo usuário deve ser igual ao número de graus de liberdade dinâmicos. Se este for o caso, o sistema é estruturalmente consistente e pode-se seguir com a análise da consistência do problema de valor inicial, apresentado na próxima seção.

De acordo com o método de redução de índice apresentado, os sistemas reduzidos podem apresentar *novas* variáveis. Mas, *a priori*, não é sabido se estas variáveis *novas*

deverão ser consideradas como novas variáveis dependentes (*y*) ou novas derivadas (*y'*). Então, a fim de determinar à qual conjunto as variáveis *novas* farão parte, o seguinte procedimento deve ser aplicado à lista de variáveis *novas* geradas pelo método de redução de índice:

- Se uma variável *nova* foi adicionada à lista e a sua derivada não: a variável será na verdade uma derivada de variável nova e deverá ser adicionada ao conjunto *y'* enquanto que a sua integral deverá ser adicionada a *y* como uma nova variável, e a equação $\frac{d}{dt}(\text{variável original}) = (\text{variável nova})$ deverá ser adicionada ao sistema de equações;

- Se uma variável *nova* foi adicionada à lista e a sua derivada também: esta variável deve ser adicionada como uma nova variável de *y* enquanto sua derivada deve ser adicionada a *y'*.

Se este procedimento não for aplicado (considerando todas as derivadas de alta ordem como variáveis novas) o sistema resultante pode não apresentar índice zero, quando analisado isoladamente.

5.3.3. Consistência de Condições Iniciais

O sistema (5.1), se analisado de forma independente, pode apresentar infinitas soluções, mas para fins de simulação apenas uma é de interesse. Esta solução é aquela que satisfaz além do sistema (5.1), um outro conjunto de restrições que o torna de solução única. Quando estas restrições são todas fornecidas para um mesmo valor da variável independente (normalmente o tempo inicial), o problema formado é denominado de problema de valor inicial (*initial value problem*, IVP), e pode ser representado por:

$$\begin{aligned} \hat{F}(t, y, y') &= 0 \quad (a) \\ I(t_0, y_0, y_0') &= 0 \quad (b) \end{aligned} \tag{5.5}$$

onde, $\hat{F}$ e I são sistemas DAE, embora I seja válido apenas para um ponto específico da variável independente (t_0).

O teste do número de graus de liberdade dinâmicos (Seção 5.3.2) é de grande importância à esta classe de problemas, mas além disto é crucial o teste de consistência entre os subsistemas (5.5a) e (5.5b), denominado de teste de consistência de condições inicias.

O trabalho de Costa Jr. *et al*. (2002) trata da caracterização de sistemas DAE e da consistência de condições iniciais. Isto é feito pela redução do índice do sistema DAE através do algoritmo de Pantelides. Assim, tendo-se reduzido o índice do sistema DAE até 1 a

consistência de condições iniciais simplifica-se, pois, como apresentado anteriormente, para sistemas de índice 1 o número de condições iniciais deve ser igual ao número de equações diferenciais do sistema.

Neste trabalho é proposta uma forma semelhante à apresentada em Costa Jr. *et al*. (2002) para o teste de condições iniciais. Porém, com a utilização da redução total de índice. Considere-se que $\hat{F}$ é um sistema DAE de índice reduzido a zero, como apresentado na Seção 5.3.2. Para este caso, um teste de condições iniciais pode ser executado de forma simples, pois o sistema (5.5) reduz-se a um sistema NLA e a sua consistência pode ser testada exatamente como descrito na Seção 5.2.

5.3.4. Índice 1 ou zero?

Existe uma grande dúvida sobre a real necessidade ou sobre as possíveis vantagens ou desvantagens da redução do índice de um sistema algébrico-diferencial até zero (redução total) ou da sua redução apenas até 1. O trabalho de Soares e Secchi (2002a, em anexo) é bastante elucidativo quanto a estes aspectos e algumas de suas conclusões são reproduzidas nesta seção.

De fato quando um sistema sofre uma redução de índice total este tende a apresentar dimensões bem maiores do que se o mesmo fosse reduzido apenas até índice 1. Tal fato é observado para a maioria dos sistemas de índice elevado e é uma grande desvantagem da redução total de índice. Em contrapartida, a solução da condição inicial de um sistema de índice zero mostra-se muito mais robusta quando da redução total, pois não há a necessidade da utilização de métodos aproximados. Além disso, também são observados casos em que a evolução da solução do problema reduzido até índice 1 falha por falta de convergência enquanto que para o mesmo problema com redução total de índice progride sem problemas.

De forma geral, a redução prévia de um sistema DAE até índice zero aumenta a robustez tanto na inicialização do problema quanto na evolução da solução quando comparada com a redução até índice 1. Embora, por questões de eficiência computacional durante a evolução na solução, a utilização de um sistema reduzido apenas até índice 1 seja vantajosa, devido as maiores dimensões do sistema com redução total de índice.

5.4. Conclusão

Falhas na solução de problemas de simulação ocorrem basicamente por dois motivos: erro na própria descrição do problema ou falha do algoritmo numérico utilizado.Neste capítulo foram apresentados métodos que vêm de encontro com estas deficiências.

Primeiramente, foram apresentados métodos para a detecção e conseqüente advertência ao usuário de problemas de descrição. Estes métodos baseiam-se em inconsistências de unidades de medida ou inconsistências relativas ao sistema como um todo, tais como singularidades estruturais ou numéricas. Por fim, foi proposto um novo método robusto para a inicialização de problemas DAE (que é a etapa responsável pela maior parte das falhas na solução numérica) fazendo com que as falhas na solução se tornem mais raras. Este método utiliza-se de um novo algoritmo de redução de índice, capaz de reduzir o índice estrutural de sistemas DAE até zero.Assim, tal metodologia permite, também, a solução direta de problemas de índice elevado utilizando-se rotinas projetadas para sistemas de baixo índice.

Capítulo 6 Estudo de Casos

Neste capítulo são apresentados alguns problemas típicos de simulação dinâmica de processos químicos e outros que não podem ser resolvidos diretamente com os simuladores disponíveis no mercado. O principal objetivo deste capítulo é a demonstração da viabilidade e das vantagens das propostas e métodos apresentados e desenvolvidos nos capítulos anteriores.

Os resultados apresentados neste capítulo foram obtidos utilizando-se o *software* EMSO – *Environment for Modeling, Simulation and Optimization* (Soares e Secchi, 2002c, em anexo), o qual é capaz de interpretar modelos em conformidade com a Seção 3.3 e executar simulações dinâmicas utilizando-se do pacote DASSLC (Secchi e Pereira, 1997). Além disso, o EMSO dispõe de sistemas próprios de derivação automática e simbólica, implementados conforme discutido na Seção 4.3.

6.1. Problemas Típicos

A Engenharia Química é uma área rica em problemas dinâmicos. A modelagem e simulação destes problemas é utilizadas para diversos fins, tais como a obtenção de conhecimento profundo do processo, treinamento de pessoal, projeto, controle ou otimizações em geral.

Como problemas típicos podem ser citados: separação em um estágio de equilíbrio, reatores do tipo CSTR e reatores do tipo PFR. Estes problemas são tratados individualmente a seguir com o objetivo de avaliar se a estrutura proposta para um simulador de processos dinâmicos nos capítulos anteriores é adequada.

6.1.1. Separação em um Estágio de Equilíbrio

A separação de dois ou mais componentes em um único estágio é uma operação comum em plantas industriais. Nestes sistemas uma mistura, a uma pressão acima do seu ponto de bolha, portanto no estado líquido, é alimentada a um vaso, chamado de vaso de *flash* ou tambor de *flash*. Este vaso opera a uma pressão inferior à pressão de ponto de bolha da mistura, fazendo com que parte desta vaporize.

A separação por *flash* vem sendo utilizada como representação esquemática de problema dinâmico neste trabalho, e pode ser observada na Figura 3.3 e Figura 4.2.

O modelo simplificado de um *flash* pode ser visto na Figura 6.1, este modelo apresenta duas variáveis de entrada (corrente de alimentação *Feed* e o calor fornecido ao vaso q) e duas variáveis de saída (as correntes de vapor V e líquido L). Tais variáveis poderão ser conectadas à outros *Device*s para formar um *FlowSheet*. Como pode ser visto nas linhas 16-18 da Figura 6.1 o modelo do *flash* utiliza-se de *SubModel*s para o cálculo de equilíbrio e de entalpias. Estes modelos internos são apresentados na Figura 6.2, sendo que o modelo para o cálculo da entalpia do líquido foi omitido por ser análogo ao modelo para o cálculo da entalpia do vapor.

A simulação de um sistema de separação como o modelado na Figura 6.1 é relativamente simples quando as condições de alimentação e um calor q têm seus perfis especificados, sendo que a maior complicação fica relacionada com o modelo termodinâmico utilizado.

```
include "Thermo";

Model FlashSimplified
    VARIABLES
    in  Feed as MaterialStream(Brief="Feed Stream");
    out L as MaterialStream(Brief="Liquid Stream");
    out V as MaterialStream(Brief="Vapor Stream");
    in  q as EnthalpyRate(Brief="Specified duty");
    Ml as HoldupMolar(Brief="Total liquid molar holdup");

    EQUATIONS
    "Component Material balance" diff(Ml*L.z)=Feed.F*Feed.z-V.F*V.z-L.F*L.z;
    "Energy balance"             diff(Ml*L.h)=q+Feed.F*Feed.h-V.F*V.h-L.F*L.h;
    "Molar Fraction constraint"  sum(V.z) = sum(L.z) = 1;
    "Temperature equilibrium"    V.T = L.T;
    "Pressure equilibrium"       V.P = L.P;

    SUBMODELS
    equilibium  as  Antoine(y_i=V.z, T=L.T, P=L.P, x_i=L.z);
    LEnth   as  LiquidEnthalpy(h=L.h, T=L.T, P=L.P, x_i=L.z);
    VEnth   as  VapourEnthalpy(H=V.h, T=V.T, P=V.P, y_i=V.z);
end
```

Figura 6.1 Modelagem simplificada de um *flash* com controle de pressão e nível.

```
    include "StdTypes";

    Model Antoine
    PARAMETERS
    ext  Components as ChemicalComponent;

    VARIABLES
    in T as Temperature; in P as Pressure;
    in  x_i as FractionMolar; in  y_i as FractionMolar;

    EQUATIONS
    P*y_i = x_i * "kPa" * exp(Components.Pvap.A*ln(T/"K") +
      Components.Pvap.B/T + Components.Pvap.C +
      Components.Pvap.D*T^2);
    end

    Model VaporEnthalpy
    PARAMETERS
    ext  Components as ChemicalComponent;
    Tr as Temperature;

    VARIABLES
    in T as Temperature; in P as Pressure;
    in  y_i as FractionMolar; in  H as EnthalpyMolar;

    EQUATIONS
    H = sum( y_i * ( Components.cp.A*(T-Tr)+
      Components.cp.B*(T^2-Tr^2)/2+Components.cp.C*(T^3-Tr^3)/3
      +Components.cp.D*(T^4-Tr^4)/4+Components.cp.E*(T^5-Tr^5)/5 ) );
    end
```

Figura 6.2 Modelo termodinâmico tipo gás ideal com equação de Antoine e correlação para cálculo da entalpia da fase vapor de uma mistura.

Um caso mais complexo pode ser imaginado, quando o calor *q* não é especificado mas sim um perfil de temperatura. A solução deste caso pode ser útil no estudo de controladores (ótimo, preditivo, etc.). Mas esta solução não é de fácil obtenção, pois com tal configuração o

modelo do processo forma um sistema DAE de índice elevado e sua solução não é possível com os pacotes computacionais tradicionais.

Com o método apresentado na Seção 5.3.2 este sistema pode ser resolvido. Os resultados de uma simulação dinâmica para o caso de três componentes, utilizando um modelo termodinâmico tipo gás ideal com equação de Antoine e controle de nível e de pressão podem ser vistos na Figura 6.3. Os dados utilizados para esta simulação encontram-se no Apêndice A. A solução de problemas com índice elevado é tratada na Seção 6.4.

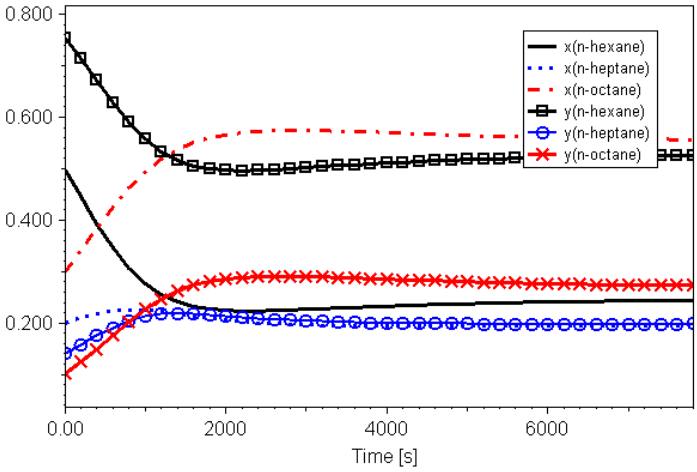


Figura 6.3 Simulação dinâmica de um *flash* com controle de pressão e nível.

6.1.2. Reator CSTR

O reator contínuo de mistura é facilmente encontrado na indústria química. Este tipo de reator é referenciado como CSTR (*Continuous-Stirred Tank Reactor*) e normalmente é operado de forma a garantir uma boa mistura em seu interior.

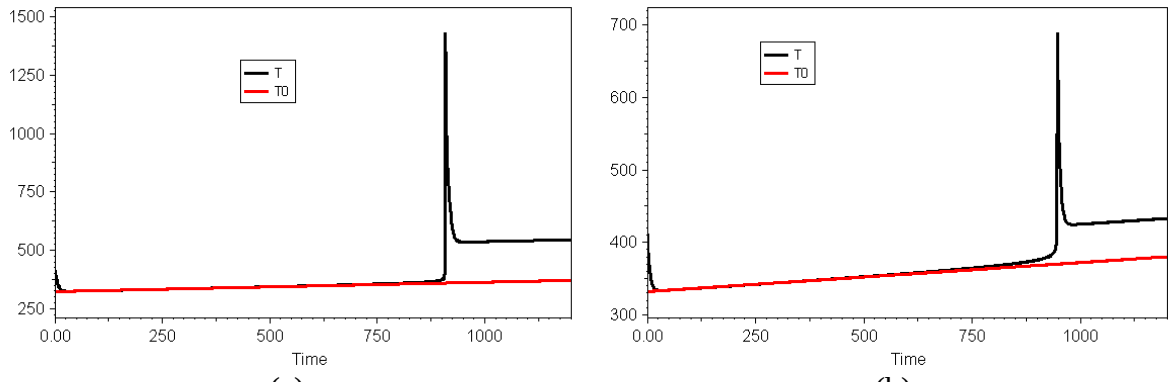
A modelagem de um reator CSTR na linguagem proposta na Seção 3.3 é simples e já foi delineada na Figura 3.21. O modelo de um reator CSTR não-isotérmico com reação de primeira ordem e camisa de troca térmica pode ser visto na Figura 6.4. A simulação estacionária destes reatores, salvo casos de múltiplas soluções, é trivial. Já simulações dinâmicas podem ser úteis em uma série de tarefas: projeto de controladores; determinação das curvas de *ignição* e *extinção*; projeto de partida, etc.

A solução do problema hipotético de um reator CSTR não-isotérmico com reação de primeira ordem com um reagente A formando um produto B pode ser vista na Figura 6.5. Este problema apresenta multiplicidade de estados estacionários e se a temperatura de alimentação T_0 for elevada até um determinado patamar ocorrerá a *ignição* do reator.

No caso (a) da Figura 6.5 a *ignição* do reator gera um pico de temperatura *T* que beira os 1500 K, fato que inviabiliza uma partida nestas condições (dependendo do material do reator). O caso (b) já apresenta condições mais seguras para a partida do reator e picos de temperatura ainda mais baixos podem ser obtidos por meio de outros experimentos. Os dados utilizados para estas simulações encontram-se no Apêndice A.

```
1       FlowSheet CSTRNaoIsotermico
2       PARAMETERS
3       Fe;Fin; Fout; CA0; CB0; FA0; FB0; dens; Cps;
4       # Troca térmica
5       Tw; U; A;
6       # Reação
7       k0; Ea; R; HR;

8       VARIABLES
9       T as Real(Default=300);
10      k; CA; CB;
11      V as Real(Default=1);
12      rA; rB; T0; q;

13      EQUATIONS
14      q=U*A*(T-Tw);
15      k=k0*exp(Ea/R*(1/436.15-1/T));
16      rA=-k*CA;
17      rB=-rA;
18      diff(T)=((FA0*Cps*(T0-T)+HR*rA*V-q)/(dens*V*Cps));
19      dens*diff(V)=(Fin*dens-Fout*dens);
20      diff(V*CA)=(FA0-Fout*CA+V*rA);
21      diff(V*CB)=(FB0-Fout*CB+V*rB);

22      T0= (60+273.15) + (time*40/1000); #temperatura da corrente de alimentação
23      end
```

Figura 6.4 Modelo de um CSTR não-isotérmico com reação de primeira ordem e camisa de troca térmica.



(a) (b)

Figura 6.5 Perfil de temperatura na *ignição* de um reator CSTR com reação de primeira ordem.

Na Figura 6.6 são demonstrados os perfis de concentração para o caso (b). Nesta figura pode ser observado o ponto em que ocorre a *ignição* do reator, quando há uma queda brusca na concentração de reagente C_A e o inverso ocorre com a concentração do produto C_B.

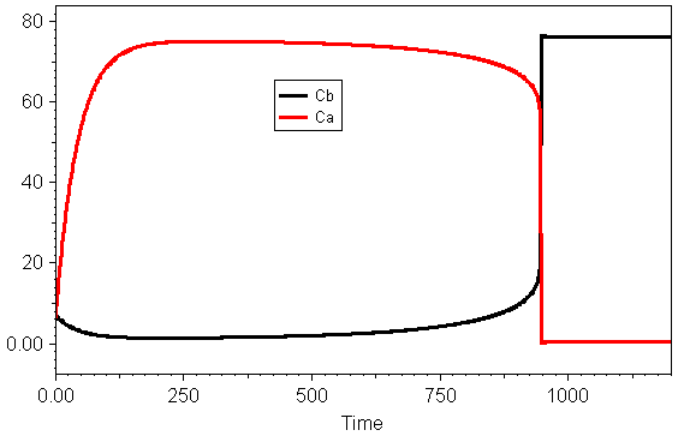


Figura 6.6 Perfil de concentração de reagente (C_A) e de produto (C_b) durante a *ignição* de um reator CSTR com reação de primeira ordem.

6.1.3. Reator PFR

Outro tipo de reator também facilmente encontrado é o reator tubular. Quando pode-se considerar que não há variações na concentração e temperatura na direção radial do tubo, refere-se a este reator como um PFR (*Plug-Flow Reactor*).

A representação de um reator PFR na forma de um sistema DAE pode ser feita de duas maneiras: modelagem estacionária com a consideração do comprimento do reator como variável independente; e modelagem dinâmica onde o comprimento do reator é discretizado e a variável independente é o tempo, este método é conhecido como MOL (*Method of Lines*).

A modelagem estacionária de um reator PFR conforme a linguagem proposta na Seção 3.3 não apresenta dificuldades, assim como a sua solução. A única particularidade deste caso é que a variável independente não é mais o tempo, mas sim o comprimento ou volume do reator. Sendo assim, a função **`diff`** é utilizada como derivada em relação ao comprimento ou volume do reator.

Como exemplo, considere-se o balanço de massa em um reator PFR: $\frac{dX}{dV} = \frac{-r_A}{F_{A0}}$, onde X é a conversão, V o volume, r_A é a taxa de reação e F_{A0} é a vazão de alimentação. Esta equação deve ser escrita como apresentado na linha 9 da Figura 6.7, ficando implícito que o volume V é a variável independente.

```
1   FlowSheet PFREmcamisado
2      PARAMETERS
3      CA0; FA0; T0; UA; Tw;

4      VARIABLES
5      T as Real(Brief=”Temperature”,Default=300);
6      X as Fraction(Brief=”Convertion”);
7      rA; CpA; dCp; therm; dh;

8      EQUATIONS
9      diff(X) = -rA/FA0;
10     diff(T) = (ua*(Ta-T)+ra*dh)/(fa0*(cpa+x*dcp));
11     cpa = 26.6+0.183*T-0.0000459*T^2;
12     dcp = 6.8-0.0115*T-3.81e-6*T^2;
13     term = -1.27e-6*(T^3-298^3);
14     dh = 80770+6.8*(T-298)-0.00575*(T^2-298^2)+term;
15  end
```

Figura 6.7 Modelo de um PFR estacionário não-isotérmico com camisa de troca térmica.

Resultados da simulação de um PFR estacionário com uma reação endotérmica e camisa de troca térmica, conforme modelo da Figura 6.7 e dados do exemplo 8-7 apresentado por Fogler (2001) podem ser vistos na Figura 6.8. Os círculos presentes nas curvas desta figura apenas explicitam os pontos de reportagem da solução e não têm relação com o passo de integração, uma vez que o integrador utilizado foi o DASSLC o qual utiliza passo e ordem variáveis.

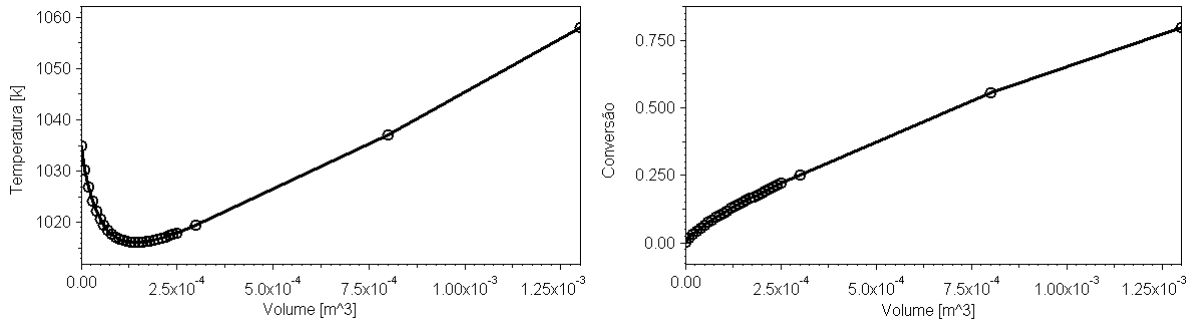


Figura 6.8 Resultados da simulação de um reator PFR com uma reação endotérmica e camisa de troca térmica.

A simulação dinâmica de um reator PFR em simuladores que utilizam-se de *solvers* projetados para sistemas DAE requer a discretização do comprimento do reator. Pois, neste caso, o sistema apresenta duas variáveis independentes: o tempo e o comprimento do reator.

Na linguagem proposta na Seção 3.3, derivadas com relação à variável independente são obtidas pela utilização da função **`diff`** com apenas um argumento (a expressão que se deseja derivar). No caso de haver mais de uma variável independente, o mesmo operador pode ser utilizado para a obtenção de derivadas parciais, porém com a utilização de dois argumentos:

```
diff(expression, independentVar)
```

Assim, sistemas com mais de uma variável independente podem ser representados e, a nível de linguagem, não é necessária discretização alguma. Embora, no momento de se obter a solução, o simulador deverá aplicar algum tipo de discretização (diferenças finitas, elementos finitos, volumes finitos, etc.) a todas as variáveis independentes, exceto no tempo, que será a variável independente de integração.

A especificação mais detalhada deste tipo de problema, assim como o estudo e implementação dos métodos de discretização serão objetivo de trabalhos futuros.

6.2. Dificuldades na Inicialização de Sistemas DAE

A maior parte das falhas na solução numérica de sistemas algebrico-diferenciais ocorre durante a sua inicialização (Wu e White, 2001). Estas ocorrem principalmente por dois motivos: inconsistência de condições iniciais ou pela falha do próprio método numérico utilizado, inadequado ao sistema.

Por este motivo, na Seção 5.3.3 foi proposta uma metodologia para a inicialização de sistemas DAE. Esta metodologia tende a ser mais robusta na inicialização de sistemas que apresentem índice 1 além de possibilitar a solução de sistemas com índice elevado com rotinas projetadas para sistemas com índice até 1.

6.2.1. Sistema Galvanostático

O trabalho de Wu e White (2001) trata especificamente da inicialização de sistemas DAE, onde são comparados os métodos de inicialização de diversos *solvers* para sistemas DAE e um utilizado pelos autores. Esta comparação é feita através de um problema teste, descrito a seguir.

$$\begin{aligned} \frac{rV}{W} y_1' &= \frac{j_1}{F} \quad (a) \\ j_1 + j_2 - i_{app} &= 0 \quad (b) \end{aligned} \tag{6.1}$$

onde,

$$\begin{aligned} j_1 &= i_{01}\left[2(1-y_1)\exp\left(\frac{0.5F}{RT}\left(y_2 - f_{eq,1}\right)\right) - 2y_1\exp\left(-\frac{0.5F}{RT}\left(y_2 - f_{eq,1}\right)\right)\right] \quad (a) \\ j_2 &= i_{02}\left[2(1-y_1)\exp\left(\frac{F}{RT}\left(y_2 - f_{eq,2}\right)\right) - \exp\left(-\frac{F}{RT}\left(y_2 - f_{eq,2}\right)\right)\right] \quad (b) \end{aligned} \tag{6.2}$$

Os valores dos parâmetros são: F = 96487, R = 8.314, T = 298.15, $f_{eq,1}$ = 0.420, $f_{eq,2}$ = 0.303, r = 3.4, W = 92.7, V = 1.10^{-5}, i_{01} = 1.10^{-4}, i_{02} = 1.10^{-10} and i_{app} = 1.10^{-5}. As unidades de medida dos parâmetros e das variáveis foram omitidas por simplicidade. O sistema DAE (6.1) representa a modelagem de um processo galvanostático de um filme fino de hidróxido de níquel, onde y_1 é a concentração de NiOOH e y_2 a diferença de potencial na interface sólido-líquido.

Este sistema apresenta duas variáveis y_1 e y_2 e duas equações, desta forma ambas as variáveis são dependentes do sistema. A derivada da variável y_2 não aparece diretamente no sistema, porém esta pode ser obtida derivando-se (6.1b) com relação à variável independente. Aplicado este procedimento pode-se concluir que o sistema possui índice diferencial igual a um. Analisando o sistema dinamicamente, o mesmo apresenta quatro variáveis $[y_1, y_2, y_1', y_2']^T$ e três equações independentes: [(6.1), (6.1b)']. Assim, o número de graus de liberdade dinâmicos deste sistema é igual a um, portanto uma de suas quatro variáveis deve ser arbitrada como condição inicial e as outras são determinadas pelo sistema.

Uma condição inicial consistente para o sistema em estudo não é de trivial obtenção, para um eletrodo de níquel descarregado os valores das variáveis são:

$$\begin{aligned} y_1(0) &= 0.05 \quad (a) \\ y_2(0) &= 0.38 \quad (b) \end{aligned} \tag{6.3}$$

Entretanto, os valores apresentados em (6.3) não são consistentes com o sistema (6.1), ou seja, não são solução do sistema. Mas, estes valores podem ser utilizados como uma estimativa inicial para a obtenção de uma condição inicial consistente.

No trabalho de Wu e White (2001), o sistema (6.1) foi utilizado para a comparação de faixa de convergência da estimativa inicial de uma das variáveis quando a outra era fornecida como condição inicial. Os *solvers* utilizados na comparação baseiam-se em diferentes métodos de inicialização: DASSL utiliza a fórmula de diferenças finitas à esquerda (Brenan et al., 1989); RADAU5 utiliza o método de Runge-Kutta implícito (Hairer e Wanner, 1991); LIMEX utiliza um método de extrapolação baseado no método de Euler implícito (Deuflhard et al., 1987); e DAEIS utiliza o método de Newton (Wu and White, 2001).

Uma comparação de robustez quanto a inicialização do sistema (6.1) entre o método de inicialização proposto na Seção 5.3.2 e os pacotes numéricos acima citados foi feita. Os resultados desta comparação podem ser vistos na Tabela 6.1. Esta tabela apresenta a faixa de convergência de uma das variáveis quando a outra é fornecida, para diferentes métodos.

Como pode ser observado na Tabela 6.1 o método proposto na Seção 5.3.3, o qual utiliza o sistema de índice zero para a inicialização mostrou-se mais robusto. A faixa de convergência foi muito superior à obtida com pacotes computacionais largamente utilizados.

O arquivo de descrição deste exemplo na linguagem proposta no Capítulo 3 pode ser encontrado na Seção 3.4.1.

Tabela 6.1 Faixas de convergência de diferentes *solvers* quando da inicialização de (6.1).

Solver	**Faixa de Convergência de $y_2(0)$, dado $y_1(0)$=0.05**	**Faixa de Convergência de $y_1(0)$, dado $y_2(0)$=0.38**
DASSL (Euler explicito)*	$0.321 \le y_2(0) \le 0.370$	$0.071 \le y_1(0) \le 0.352$
LIMEX (Euler implicito)*	$0.318 \le y_2(0) \le 0.377$	$0.056 \le y_1(0) \le 0.418$
RADAU5 (RK implicito)*	$0.348 \le y_2(0) \le 0.352$	$0.143 \le y_1(0) \le 0.190$
DAEIS (tipo Newton)*	$-0.974 \le y_2(0) \le 1.663$	$0.0 \le y_1(0) \le 1.0$
Método Proposto (Índice zero)	**$-2.70 \le y_2(0) \le 2.66$**	**$-\infty \le y_1(0) \le \infty$**
Valor Consistente	$y_2(0)$=0.35024	$y_1(0)$=0.15513

* Fonte Wu e White, 2001.

O resultado de uma simulação dinâmica deste exemplo, utilizando-se da descrição do problema como apresentada na Seção 3.4.1, pode ser observado na Figura 6.9. Esta figura apresenta os perfis de y_1 e y_2 ao longo do tempo para ambos os casos: $y_1(0) = 0.05$ e $y_2(0) = 0.38$. Embora sejam apresentados apenas os valores das variáveis, os perfis de suas derivadas também são determinados na solução.

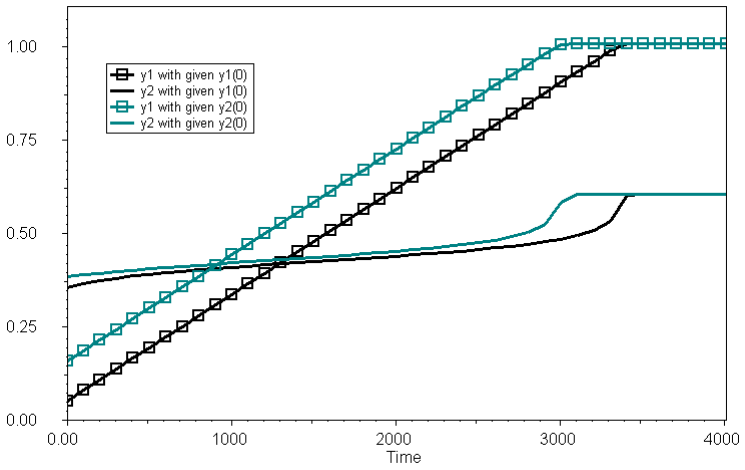


Figura 6.9 Simulação dinâmica do processo galvanostático (6.1).

6.2.2. Pêndulo Oscilante

A maioria dos códigos utilizados para a inicialização de sistemas DAE não consegue tratar sistemas com índice superior a um. Mas, mesmo para casos de sistemas com índice um podem ocorrer dificuldades ou falhas na inicialização, como discutido na Seção 6.2.1. A metodologia proposta na Seção 5.3.3, a qual aplica primeiramente uma redução de índice até zero para posterior inicialização pela solução de um sistema não-linear, independe do índice do sistema original.

No exemplo anterior foi feita uma comparação de robustez na inicialização de um sistema DAE de índice um entre diversos *solvers*. Nesta seção, um exemplo largamente citado na literatura (Pantelides, 1988, Unger *et al.*, 1994 e Mattsson *et al.*, 2000), que apresenta índice três, é utilizado como um teste para a metodologia proposta quanto a inicialização de sistemas DAE de índice elevado. O sistema estudado é o de um pêndulo oscilante, ilustrado na Figura 6.10, e modelado conforme (6.4).

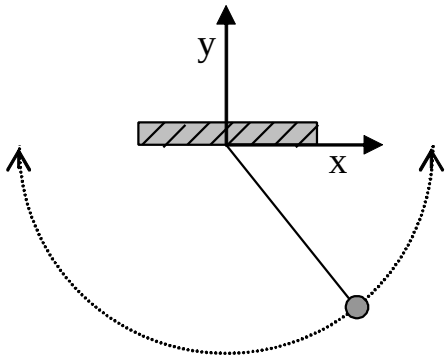


Figura 6.10 Representação do sistema do pêndulo oscilante.

$$\begin{aligned} w &= x' && (a) \\ z &= y' && (b) \\ T.x &= w' && (c) \\ T.y - g &= z' && (d) \\ x^2 + y^2 &= L^2 && (e) \end{aligned} \tag{6.4}$$

onde, x e y representam as posições horizontal e vertical do pêndulo, w e z são as correspondentes velocidades nestas direções, T é a tensão na haste, L = 1,0 m o seu comprimento e g = 9,8 m/s^2 é a aceleração da gravidade.

Para que os métodos numéricos tradicionais possam avançar na solução de um sistema DAE é necessário que este obtenha estimativas para o valor da derivada de todas as variáveis do problema, inclusive aquelas que se apresentam apenas como variáveis algébricas.

A variável *T* apresenta-se neste sistema apenas na forma algébrica, e as dificuldades na solução deste problema advém exatamente deste fato. Uma tentativa de obter-se *T'* pela diferenciação de (6.4c) ou (6.4d) gera novos problemas, pois a diferenciação destas equações incluem derivadas de mais alta ordem no sistema (*w''* e *z''*).

A aplicação da metodologia de redução de índice apresentada na Seção 5.3.2 indica que, para a obtenção de um sistema de índice zero é necessário adicionar ao sistema as equações: (6.4a'), (6.4a''), (6.4b'), (6.4b''), (6.4c'), (6.4d'), (6.4e'), (6.4e''), (6.4e'''); e variáveis: *x''*, *y''*, *x'''*, *y'''*, *w''* e *z''*. Isto significa em adicionar ao sistema 9 equações e 6 variáveis.

O sistema (6.4) apresenta 5 variáveis e 5 equações, ou seja, todas as variáveis são dependentes do sistema. O sistema de índice reduzido contém: 16 variáveis (5 variáveis originais, suas derivadas e 6 variáveis extras) e 14 equações independentes (5 equações originais e 9 extras), o que leva a dois graus de liberdade dinâmicos para o sistema original. Embora o sistema apresente, dois graus de liberdade dinâmicos, não se pode arbitrar qualquer conjunto de duas variáveis do mesmo, como por exemplo *x*(0) e *y*(0), pois estas variáveis já estão relacionadas por (6.4e).

Desta forma, fica claro que não se pode arbitrar livremente qualquer par de variáveis para a inicialização do sistema. O método para teste de consistência de condições iniciais apresentado na Seção 5.3.3 é capaz de detectar tais inconsistências. Através da aplicação deste método foram testados alguns pares de variáveis como condição inicial, resultando nos dados da Tabela 6.2, os valores em negrito foram fornecidos como condição inicial enquanto os outros foram calculados.

Tabela 6.2 Inicialização do sistema (6.4) para alguns casos de condições iniciais.

Caso	***x***	***y***	***w***	***z***	***T***
1	**0.5**	0.87	**0.0**	0.0	8.49
2	**0.5**	0.87	1.74	**-1.0**	4.49
3	0.0	**1.0**	**2.0**	0.0	5.80
4	0.87	**0.5**	0.58	**-1.0**	3.56
5*	**0.0**	**1.0**	-	**-**	-
6**	**1.2**	**-**	**0.0**	**-**	-
7**	**-**	**-**	**0.0**	**0.0**	-

*Inconsistência estrutural detectada; ** Falha na solução numérica.

Embora a Tabela 6.2 apresente apenas pares de variáveis a serem arbitradas, cada condição inicial pode ser uma expressão envolvendo qualquer das variáveis do problema, incluindo também as variáveis extras, embora na maioria dos casos não haja sentido para isto. A utilização de condições iniciais na forma de expressões não traz restrição alguma ao

algoritmo de teste de consistência de condições iniciais. Além disto, no caso de inconsistência, as estruturas geradas pelo algoritmo permitem sugerir ao usuário modificações que gerem um sistema consistente, como por exemplo envolver uma determinada variável nas condições iniciais.

É importante frisar que o método proposto detecta inconsistências estruturais, portanto falhas devido à inconsistências numéricas ainda podem ocorrer, como por exemplo se for especificado um valor para $x(0)$ ou $y(0)$ maior do que o comprimento da haste L, como observado no Caso 6 da Tabela 6.2. Outro exemplo é o Caso 7, onde a condição inicial fornecida leva a infinitas soluções, fato que gera uma falha no método tipo Newton, utilizado para a inicialização do sistema.

6.3. Eventos e Reinicialização

Na Seção 4.6 foi discutida importância e os casos em que há a necessidade de modelagem e solução de sistemas na forma híbrida (contínua-discreta).

Embora a linguagem apresentada no Capítulo 3 tenha deixado sintaxe de modelagem para sistemas híbridos como uma proposta para futuros desenvolvimentos, nesta seção é apresentada a solução de um sistema contínuo-discreto. Este sistema tem sua modelagem contínua conforme a linguagem apresentada, porém a sua execução apresenta eventos discretos. Assim sendo, esta seção e o exemplo contido na mesma têm como objetivo apenas apresentar a viabilidade da solução de problemas híbridos utilizando-se da metodologia de detecção de eventos de descontinuidade proposta na Seção 4.6.1.

O processo de um eletrodo de níquel modelado como (6.1) pode ser utilizado como um exemplo de sistema que apresenta eventos discretos de descontinuidade, quando da execução do seguinte procedimento:

- Carregar durante 500 s, com 100 s adicionais para cada ciclo, com uma corrente $i_{app} = 1\times10^{-5}$. Terminar este processo se o potencial y_2 chegar ao valor de 0,60 V;
- Abrir o circuito ($i_{app} = 0$) durante 500 s. Terminar este processo se o potencial y_2 alcançar o valor de 0,25 V;
- Descarregar o sistema com uma corrente $i_{app} = -1\times10^{-5}$ durante 1000 s. Terminar este processo se o potencial y_2 chegar ao valor de 0,25 V.

Resultados da simulação deste processo de carga/abertura/descarga com a utilização da metodologia de localização de *eventos* descrita na Seção 4.6.1 podem ser observados na Figura 6.11.

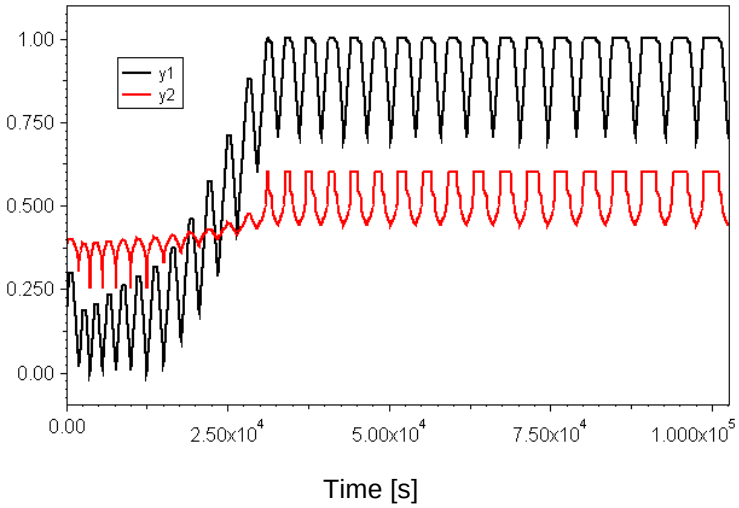


Figura 6.11 Resultados da simulação contínua-discreta do sistema (6.1) com processos de carga, abertura e descarga.

A resposta do sistema apresentada na Figura 6.11 é exatamente a encontrada no trabalho de Wu e White (2001), demonstrando que a metodologia de localização de eventos proposta na Seção 4.6.1é adequada.

6.4. Integração de sistemas DAE de Índice Elevado

A inicialização e evolução na solução de sistemas DAE de índice elevado tem sido objetivo de estudo nas duas últimas décadas. Existem alguns códigos capazes de lidar com sistemas com índice maior do que 1, como apresentado nos trabalhos de Ascher e Petzold (1992) e Sand (2002). Mas estes códigos são restritos a sistemas com índice no máximo 3 e freqüentemente são restritos a estruturas específicas como Hessenberg ou semi-explícita (Unger *et al.*, 1994). No trabalho de Costa Jr. *et al.* (2001) uma técnica de redução até índice 1 é apresentada.

Na Seção 6.2 a inicialização de tais sistemas com o procedimento apresentado na Seção 5.3.2 foi exemplificada. Nesta seção, este mesmo método é testado, através de sua aplicação em alguns exemplos, quanto a sua capacidade na evolução da solução de sistemas de índice elevado.

6.4.1. Pêndulo Oscilante

A inicialização do problema do pêndulo oscilante (6.4) foi realizada na Seção 6.2.2. Uma vez que um sistema DAE teve seu índice reduzido até zero, sua solução pode ser obtida diretamente com a utilização de algum solver capaz de resolver problemas ODE implícitos.

A solução do problema (6.4) através da utilização de seu sistema equivalente de índice reduzido pode ser vista na Figura 6.12.

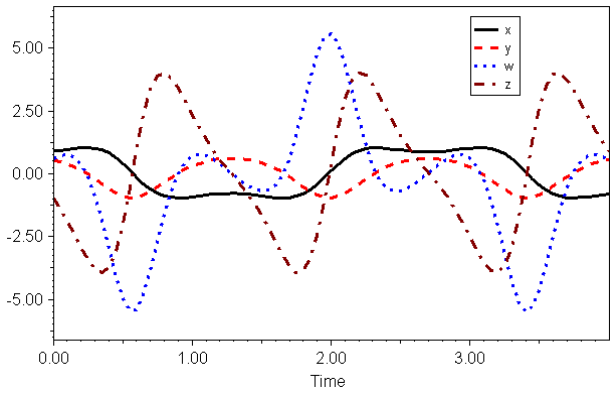


Figura 6.12 Evolução na solução do problema do pêndulo (6.4), de índice 3.

6.4.2. Coluna de Destilação Batelada com Controle de Pureza

Logsdson e Biegler (1993) propuseram um modelo simplificado para uma coluna de destilação batelada, da seguinte forma:

$$
\begin{aligned}
\frac{d}{dt}H_0 &= \frac{V}{R+1} && (a)\\
\frac{d}{dt}x_{0,j} &= \frac{V}{H_0}\left(x_{0,j} - y_{0,j} + \frac{R(x_{1,j} - x_{0,j})}{R+1}\right) && (b)\\
\frac{d}{dt}x_{i,j} &= \frac{V}{H_i}\left(y_{i-1,j} - y_{0,j} + \frac{R(x_{i+1,j} - x_{i,j})}{R+1}\right) && (c)\\
x_{np+1,j} &= \frac{V\left(y_{np+1,j} - x_{np+1,j}\right)}{H_{np+1}} && (d)\\
y_{i,j} &= K_{i,j}.x_{i,j} && (e)
\end{aligned}
\qquad (6.5)
$$

onde, V é a vazão molar nos estágios; H_i é o número de moles em cada estágio; R é a taxa de refluxo; np é o número de pratos e nc o número de componentes; $x_{i,j}$ e $y_{i,j}$ são respectivamente as frações molares no líquido e no vapor em cada estágio; K_i é a constante de equilíbrio. Na Figura 6.13 este modelo é apresentado na linguagem de modelagem proposta neste trabalho e os dados requeridos para sua solução podem ser encontrados no trabalho de Costa Jr. *et al.* (2001).

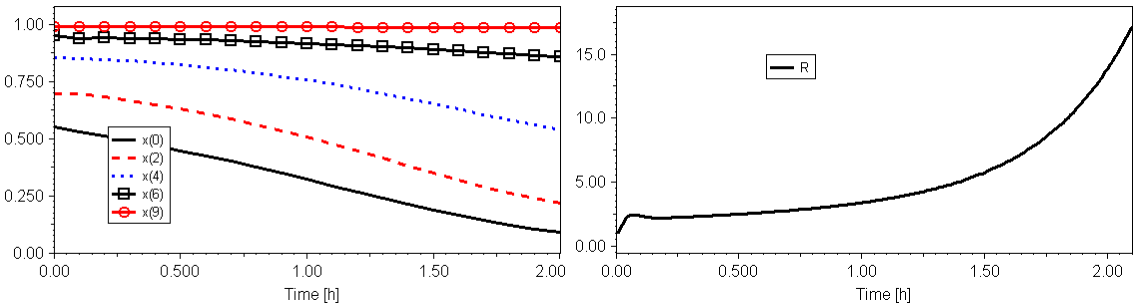
```
1   FlowSheet BatchColumn
2    PARAMETERS
3    Np  as Integer(Brief="Number of Plates");
4    Nc  as Integer(Brief="Number of Components");
5    Purity; Hi; A(Nc); B(Nc); C(Nc);
6    P  as Real(Brief=="Pressure"); V as Real;

7    VARIABLES
8    H0   as  Real(Default=1); R as Real(Default=1);
9    x1(Np+1)  as Real(Default=0.55, Lower = 0, Upper = 1);
10   x2(Np+1)  as Real(Default=0.45, Lower = 0, Upper = 1);
11   y1(Np+1)  as Real(Default=0.3, Lower = 0, Upper = 1);
12   y2(Np+1)  as Real(Default=0.7, Lower = 0, Upper = 1);
13   T(Np+1)   as Real(Default=85, Lower = 0, Upper = 400);
14
15   EQUATIONS
16   #Prato 0
17   diff(H0) = -V/(R+1);
18   diff(x1(0)) = V/H0 * (x1(0)-y1(0) + R * ((x1(1) - x1(0))/(R+1)));
19   #Outros pratos
20   diff(x1(1:Np-1)) = V/Hi * (y1(0:Np-2) - y1(1:Np-1) + (R * ( x1(2:Np) -
     x1(1:Np-1) ) / (R+1)));
21   #Último prato
22   diff(x1(Np)) = V * ( y1(Np-1) - x1(Np) )/Hi;
23
24   y1*P = x1 * exp( A(0) - (B(0)/(T + 273.15 + C(0))) );
25   y2*P = x2 * exp( A(1) - (B(1)/(T + 273.15 + C(1))) );
26   y1 = 1 - y2; x1 = 1 - x2;
27   x1(Np) = purity;
28  end
```

Figura 6.13 Modelo de uma coluna batelada com controle ótimo.

Este é um modelo simples, mas com características numéricas interessantes, principalmente se a pureza de um dos componentes é perfeitamente controlada através da especificação de um perfil de composição no condensador. Neste caso, o sistema (6.5) apresenta índice 3 e dificuldades surgem tanto na sua inicialização quanto na evolução da solução quando *solvers* projetados para sistemas DAE de baixo índice são utilizados.

A aplicação do procedimento apresentado na Seção 5.3.2 torna possível a solução do sistema (6.5), mas requer a adição de 13 variáveis *novas* para o caso de $nc = 2$ e $np = 3$. Para o caso de mais componentes ou mais pratos, este número é ainda maior. Uma discussão mais detalhada sobre a redução do índice deste sistema pode ser encontrada em Soares e Secchi (2002a, em anexo).

A solução do problema (6.5) para o caso de $nc = 2$ e $np = 12$ e termodinâmica de gás ideal pode ser vista na Figura 6.14 (a). Na Figura 6.14 (b) pode ser observado o perfil para a taxa de refluxo, que, como esperado, aumenta com o tempo devido a perda de capacidade do processo de separação.



(a) (b)

Figura 6.14 Perfis de concentração e taxa de refluxo provenientes da solução do sistema de índice elevado (6.5).

6.5. Conclusão

A avaliação das propostas apresentadas nos capítulos anteriores através de sua aplicação em um conjunto de problemas mostrou que estas são adequadas.

A *linguagem de modelagem* proposta no Capítulo 3 mostrou-se capaz de descrever problemas típicos de modelagem de sistemas dinâmicos.

A conversão da descrição textual dos modelos para sistemas de equações, como apresentado no Capítulo 4, possibilitou não só a solução de problemas típicos mas também de outros que não podem ser diretamente resolvidos pelos principais *softwares* comerciais, tais como problemas de índice elevado.

Capítulo 7 Conclusões e Perspectivas

Este trabalho apresentou um projeto que é a base para o desenvolvimento de um simulador genérico de processos dinâmicos. Este projeto envolveu a análise de ferramentas similares através de uma nova estrutura de avaliação; o desenvolvimento de uma nova *linguagem de modelagem* orientada a objetos; a proposta de uma arquitetura interna modular para a reunião de rotinas matemáticas já consagradas no meio científico; o desenvolvimento de conhecimento e algoritmos relacionados com a solução de problemas dinâmicos; testes com problemas típicos e outros que não podem ser resolvidos diretamente com *softwares* comerciais.

Cada capítulo desta dissertação apresentou suas conclusões parciais. Neste capítulo são apresentadas as conclusões do trabalho como um todo, juntamente com algumas perspectivas de desenvolvimento futuro.

7.1. Conclusões

Este trabalho teve as primeiras palavras de seu primeiro capítulo dedicadas a comentários sobre a utilidade de um simulador e durante todo o trabalho pode-se notar o quão valiosa e complexa pode ser tal ferramenta.

Ainda no Capítulo 1, um breve histórico sobre ferramentas computacionais para modelagem e simulação foi apresentado. Como pôde ser visto a história dos simuladores, ao contrário do que possa se presumir, já acumula mais de 50 anos. Este trabalho vem no sentido de acompanhar a mais nova corrente de desenvolvimento, na qual há uma migração para ferramentas orientadas a equações, poderosas e eficientes, capazes de resolver problemas com mais de 100.000 equações em intervalos de tempo razoáveis. Isto significa, por exemplo, simular 10.000 horas de uma unidade de conversão de acetileno composta por três reatores em série (cerca de 2.000 variáveis) em aproximadamente 10 segundos[◊], utilizando-se um Pentium IV 1.8 GHz.

Visando permitir o desenvolvimento do projeto de um novo simulador de processos, uma estrutura hierárquica para avaliação de pacotes computacionais para simulação de processos dinâmicos baseada em trabalhos encontrados na literatura foi proposta. Esta estrutura reúne as características mais relevantes de um simulador de processos dinâmicos e sua aplicação permitiu detectar quais são as principais deficiências de alguns dos simuladores mais populares. Além disto, a estruturação de critérios, para fins de avaliação, também serviu como um conjunto de diretrizes para o trabalho como um todo.

Uma nova linguagem orientada a objetos para a descrição de sistemas dinâmicos foi desenvolvida. Sua utilização em problemas típicos provou que seu alto nível de abstração e utilização do paradigma orientado a objetos gera uma grande eficiência na reutilização de código. Além disto, a linguagem proposta aproxima-se bastante da linguagem matemática usual e provê um bom grau de flexibilidade ao usuário.

Diagramas de classes necessários para a implementação de um simulador genérico de processos dinâmicos foram construídos. Estes envolveram: conversão da linguagem de descrição para equações executáveis e diferenciáveis (avaliação de resíduo, diferenciação automática e simbólica); simulação e otimização estacionária e dinâmica. A estrutura de classes proposta apresenta uma grande modularidade, possibilitando o compartilhamento de objetos com sistemas externos e o desenvolvimento de forma concorrente de cada um dos módulos além de propiciar uma grande facilidade de atualização independente de cada um dos subsistemas.

A estrutura interna proposta é muito semelhante ao padrão original de interfaces numéricas do projeto CAPE-OPEN e se apresenta na forma ideal para a sua implementação em alguma linguagem orientada a objetos. Espera-se que as modificações propostas por

◊ Resultado obtido com o *software* protótipo EMSO (Soares e Secchi, 2002c, em anexo).

Soares e Secchi (2002b) ao pacote numérico de interfaces do projeto CAPE-OPEN, as quais tornam o sistema muito mais eficiente em termos de tempo de execução, sejam adotadas em breve por este projeto.

Com o intuito de evitar as principais causas de falhas na solução de problemas de simulação, foram apresentados alguns métodos mais robustos que os comumente utilizados nos pacotes computacionais disponíveis no mercado. Estes métodos podem ser divididos em dois grupos: métodos para a detecção e conseqüente advertência ao usuário de problemas de descrição; e métodos numéricos para solução de problemas dinâmicos. O primeiro grupo de métodos se baseia em inconsistências de unidades de medida ou inconsistências relativas ao sistema como um todo, tais como singularidades estruturais ou numéricas. O segundo, é baseado na proposta de novos métodos para a inicialização de sistemas DAE (etapa responsável pela maior parte das falhas na solução numérica) assim como uma metodologia para evolução na solução de sistemas DAE com índice elevado.

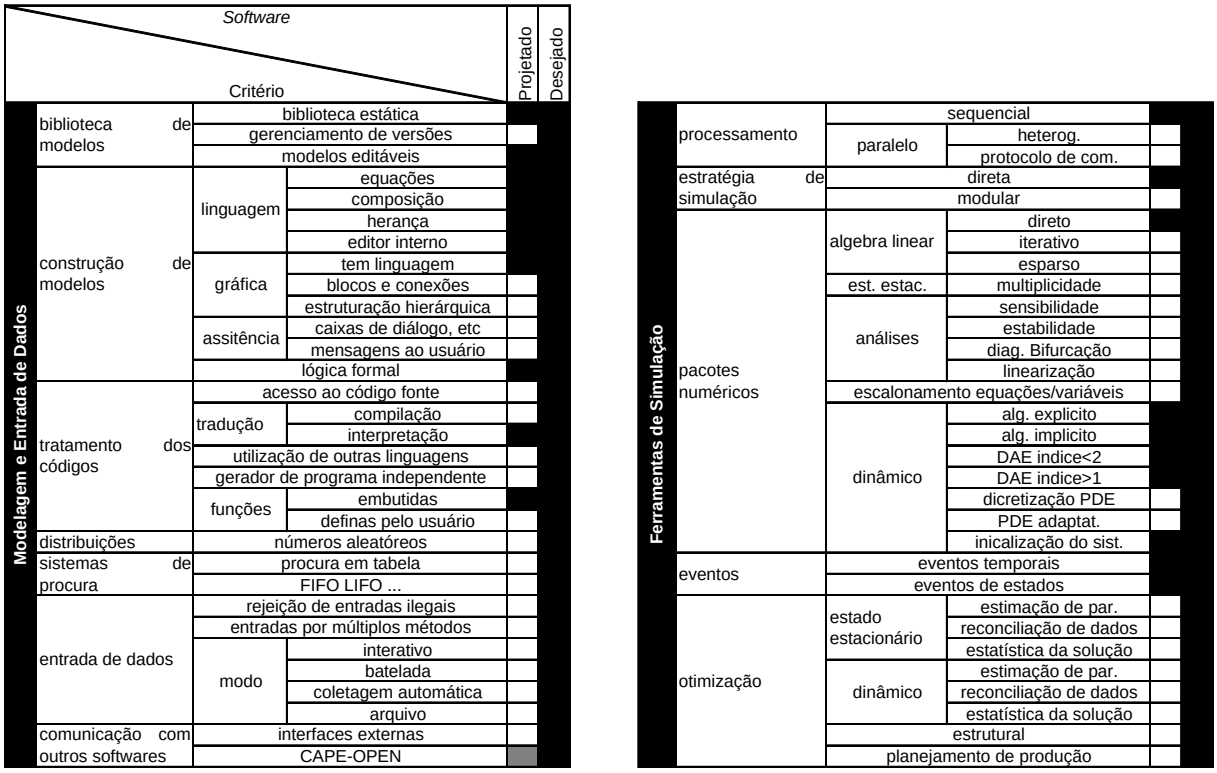
A estrutura proposta como um todo foi testada com sucesso na aplicação de problemas típicos. Além disto, as inovações apresentadas possibilitaram a solução de problemas que não podem ser resolvidos diretamente com nenhum dos pacotes para simulação disponíveis atualmente no mercado.

7.2. Perspectivas

Na Figura 7.1 são apresentados os critérios de avaliação (propostos no Capítulo 2) já contemplados neste trabalho frente aos desejados. Como pode ser observado nesta figura, para que se tenha um *software*, considerado por este trabalho como ideal, ainda existe a necessidade do desenvolvimento de: uma biblioteca de modelos, uma interface gráfica para a manipulação de *FlowSheet*s e entrada de dados, sistemas de análises (sensibilidade, estabilidade, bifurcação e linearização), solução de PDEs e otimização (estimação de parâmetros e reconciliação de dados). Com a arquitetura interna modular proposta neste trabalho, cada uma destas tarefas de desenvolvimento torna-se independente, facilitando a formação de grupos de desenvolvimento. Para que isto seja possível, a formação de projetos conjuntos com entidades interessadas é crucial.

O movimento na direção da formação de um *software* de simulação baseado no projeto apresentado neste trabalho já está iniciado. Um novo simulador chamado EMSO, abreviação para **E**nvironment for **M**odeling, **S**imulation and **O**ptimization (Soares e Secchi, 2002c, em anexo), está atualmente em fase experimental e foi utilizado para a geração de todos os resultados apresentados nos capítulos anteriores.

Espera-se que em breve o EMSO tenha disponível sua primeira versão estável e que esta, ao passar do tempo, obtenha sucesso e contemple a boa parte das características desejáveis apresentadas no Capítulo 2.



■ contempla o critério totalmente; ▒ parcialmente; □ não contempla.

Figura 7.1 Critérios de avaliação contemplados pelo projeto apresentado neste trabalho.

Referências Bibliográficas

ASCHER, U. M.; PETZOLD, L. R., “Projected Collocation for High-Order Higher-Index Differential-Algebraic Equations”. Journal Computational and Applied Mathematics, 43, p. 243-259, 1992.

ASCHER, U.M.;. MATTHEIJ, R.M.M; RUSSELL, R.D, “Numerical Solution of Boundary Value Problems for Ordinary Dierential Equations”. Classics in Applied Mathematics, Vol. 13, Society for Industrial and Applied Mathematics (SIAM) Publications, Philadelphia, PA, ISBN 0-89871-354-4, 1995.

BACHMANN, R.; BRÜLL, L.; MRZIGLOD, Th.; PALLASKE, U., “On Methods for Reducing the Index of Differential Algebraic Equations”. Computers and Chemical Engineering, Vol. 14, 11, pp. 1271-1273, 1990.

BERZ, M, “The differential algebra FORTRAN precompiler DAFOR”. Tech. Report AT-3: TN-87-32, Los Alamos National Laboratory, Los Alamos, N.M., 1987.

BISCHOF, C.; CARLE, A.; CORLISS, G.; GRIEWANK, A.; HOVLAND P, “ADIFOR – Generating Derivative Codes from FORTRAN Programs”. Scientific Programming, 1(1), p. 1-29, 1992.

BISCHOF, C.; ROH, L.; MAUER, A., “ADIC -- An extensible automatic differentiation tool for ANSI-C”. Argonne Preprint ANL/MCS-P626-1196. To appear in Software: Practice and Experience, 1997.

BORUTZKY, W., “The Bond graph Methodology and Environments for Continuous Systems Modeling and Simulation, European Simulation Microconference”. York, UK, p. 15-21, 1992.

BROENINK, J. F., “Bond-Graph Modeling in Modelica”. European Simulation Simposium, Passau, Germany, p. 19-22, 1997.

BRENAN, K. E.; CAMPBELL, S. L.; PETZOLD, L. R., “Numerical Solution of Initial-Value Problem in Differential-Algebraic Equations”. Classics Appl. Math. 14, SIAM, Philadelphia, 1996.

BANKS J., “Interpreting Simulation Software Checklists”. OR/MS Today 22 (3), p. 74-78, 1996.

BIEGLER, L. T.; Grossmann, I. E.; Westerberg, A. W., “Systematic Methods of Chemical Process Design”. Prentice Hall International Series in Phisical and Chemical Engineering Sciences, 1997.

CHE-COMP, “The State of Chemical Engineering Softwares”. www.che-comp.org, 2002.

CO, “Cape-Open RoadMap”. CO-RM-II-01, www.co-lan.org, 2002.

CO, “Open Interface Specification Numerical Solvers”. CO-NUMR-EL-03, Version 1.08, www.co-lan.org, 1999.

COSTA JR., E. F.; VIEIRA, R. C.; SECCHI, A.R.; BISCAIA, E.C., “Automatic Structural Characterization of DAE Systems”. Proceedings of the 11th European Symposium on Computer Aided Process Engineering (ESCAPE 11), Kolding, Denmark, p. 123-128, 2001.

DENG, H. L.; GOUVEIA, W.; SCALES, J., “The CWP Object-Oriented Optimization Library”. Center for Wave Phenomena, http://coool.mines.edu/, 1994.

DEUFHARD, P.; HAIRER, E.; ZUGCK J., “One Step and Extrapolation Methods for Differential-Algebraic Systems”. Numerical Mathematics, 51, p. 501-516, 1987.

DONGARRA, J.; LUMSDAINE, A.; POZO, R.; RMINGTON, K. A., “IML++ v. 1.2 Iterative Methods Library Reference Guide”. http://math.nist.gov/iml++/, 1996.

ECKEL , B., “Thinking in C++”. 2nd Ed., www.bruceeckel.com, 2000.

FOGLER, H. S., “Elements of Chemical Reaction Engineering”. 2nd Ed. Prentice Hall, 1992.

FRANKE, R., “Omuses: a tool for Optimization of Multistage systems and HQP: a solver for Sparce Nonlinear Optimization”. Dept. of Automation and Systems Engineering, Technical University of Ilmenau – Germany, http://hqp.sourceforge.net, 1998.

GILL, P. E.; JAY, L. O.; LEONARD, M. W.; PETZOLDD L. R.; SHARMA, V., “An SQP method for the optimal control of large-scale dynamical systems”. Journal of Comp. and Applied Math., 120, p. 197-213, 2000.

GOGG T. J.; MOT, J. R. A., “Introduction to Simulation”. In Preceedings of the 1993 Winter Simulation Conference, San Diago, p. 9-17, 1993.

GRIEWANK, A.; JUEDES, D.; MITEV, H.; UTKE, J.; VOGEL, O.; WALTHER, A., “ADOL-C: A Package for Automatic Differentiation of Algorithms Written in C/C++”. Ver. 1.8.2, http://www.math.tu-dresden.de/wir/project/adolc/, 1999.

GRISBY, D.; LO, SAI-LAI; RIDDOCH, D., “The omniORB version 4.0 User’s Guide”. http://omniorb.sourceforge.net/, 2002.

HAVIER, E.; WANNER, G., “Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems”. New York: Springer-Verlag, 1991.

HEARN, A. C., "REDUCE – User's Manual and Contributed Packages Manual". http://www.uni-koeln.de/REDUCE/, 1999.

HLUPIC, V., "Simulation Software: User's Requirements". Comp. Ind. Engrg, 37, p. 185-188, 1999.

HLUPIC, V.; PAUL, R. J., "Methodological Approach to Manufacturing Simulation Software Selection". Comp. Integrated Manufacturing Systems, Vol. 9., No. 1., p. 49-55, 1995.

KAY, A., "The Early History of SmallTalk". Apple Computer, http://www.metaobject.com/papers/SmallHistory.pdf, 1993.

KESLER, M. G.; KESSLER, M. M., World Pet., Vol. 29, p. 60, 1958.

KÖHLER, R.; GERSTLAUER, A.; ZEITZ, M., "Symbolic Preprocessing for Simulation of PDE Models of Chemical Processes". Math. and Comp. in Simulation, 56, p. 157-170, 2001.

LEITOLD, A.; HANGOS, K. M., "Structural Solvability Analysis of Dynamic Process Models". Computers. Chem. Engng, 25, p. 1633-1646, 2001.

LEFKOPOULOUS, A.; STADTHERR, M. A., "Index Analisys of Unsteady-State Chemical Process Systesms- I. An Algorithm for Problem Formulation". Computers Chem. Engng, Vol. 17, No. 4, p. 399-413, 1993.

LOGSDON J. S.; BIEGLER, L. T., "Accurate Determination of Optimal Reflux Policies for the Maximum Distillate Problem in Batch Distillation". Ind. Eng. Chem. Res., Vol. 32 no 4, p. 692-700, 1993.

LAW, A. M.; KELTON, W. D., "Simulation Modelling and Analysis". 2nd ed., McGraw-Hill, Singapore, 1991.

LAWSON, C. L.; HANSON, R. J.; KINCAID, D.; KROGH, F. T., "Basic Linear Algebra Subprograms for FORTRAN usage". ACM Trans. Math. Soft., 5, p. 308-323, 1979.

LORENZ, F., "A Classification of Modeling Languages for Differential-Algebraic Equations". Simulation Practice and Theory, 7, p. 553-562, 1999.

LUCA, L. DE; MUSMANNO, R., "A Parallel Automatic Differentiation Algorithm for Simulation Models". Simulation Practice and Theory, 5, p. 235-252, 1997.

MALY, T.; PETZOLD, L.R., "Numerical methods and software for sensitivity analysis of dierential-algebraic systems". Appl. Numer. Math. 20, p. 57-79, 1996.

MAPLESOFT, www.maplesoft.com, 2002.

MARQUARDT, W., "Trends in Computer-Aided Process Modelling". Computers Chem Engng, Vol. 20, No. 6/7, p. 591-609, 1996.

MATHWORKS, www.mathworks.com, 2002.

MATTSSON, S. E.; OLSSON, H.; ELMQVIST, H., "Dynamic Selection of States in Dymola". Modelica Workshop 2000 Proceedings, p. 61-67, 2000.

MERRIAN WEBSTER, "Merrian Webster on-line – The Language Center". http://www.m-w.com/, 2002.

MEZA, J. C., "Opt++: An Object-Oriented Class Library for Nonlinear Optimization". Sandia National Laboratories, http://csmr.ca.sandia.gov/projects/opt/, 1994.

NETLIB, "Netlib Repository". http://www.netlib.org, 2002.

NIKOKAUMAN, J.; HLUPIC, V.; PAUL, R. J., "A Hierarchical Framework for Evaluting Simulation Software". Simulation Practice and Theory, 7, p. 219-231, 1999.

OH, M.; PANTELIDES, C. C., "A Modelling and Simulation Language for Combined Lumped and Distributed Parameter Systems". Computers Chem. Engng, Vol. 20, No. 6/7, p. 611-633, 1996.

OMG, "The Common Object Request Broker". Version 2.3.1, www.omg.org, 1999.

OMG, "Unified Modeling Language Specification". Version 1.3, www.omg.org, 2000.

OTTER, M.; ELMQVIST, H., "Modelica – Language, Libraries, Tools". Workshop and EU-Project RealSim, Simulation News Europe, p. 3-8, 2000.

PANTELIDES, C. C., "The consistent initialization of differential-algebraic systems". SIAM J. Sci. Statist. Comput., 9, p. 213-231, 1988.

PIDD, M., "An Introduction to Computer Simulation". In: Proceedings of The 1994 Winter Simulation Conference, Florida, USA, p. 7-14, 1994.

POZO, R.; REMINGTON, K. A.; LUMSDAINE, A., "SparseLib++ v. 1.5 Sparse Matrix Class Library Reference Guide". http://math.nist.gov/sparselib++/, 1996.

SAND, J., "On Implicit Euler for High-Order High-Index DAEs". Applied Numerical Mathematics, 42, p. 411-424, 2002.

SECCHI, A. R.; PEREIRA, F. A., "DASSLC: user's manual – v2.0 (Differential-Algebraic System Solver in C)". Departamento de Engenharia Química – UFRGS, www.enq.ufrgs.br/enqlib/numeric/, 1997.

SECCHI, A. R., “Apostila do curso Otimização de processos”. Departamento de Engenharia Química – CPGEQ-UFRGS, 2002.

SEILA, A. F., “Introduction to Simulation”. In Proceedings of The 1995 Winter Simulation Conference, Virginia, USA, p. 7-15, 1995.

SOARES, R. DE P., “Runits: the Measurement Units Handling Tool - Reference Manual”. Version 0.8, rafael@enq.ufrgs.br, 2002.

SOARES, R. DE P.; SECCHI, A. R., “Direct Initialization and Solution of High-Index DAE Systems with Low-Index DAE solvers”. Comp. Chem. Engrg, submitted 2002a.

SOARES, R. DE P.; SECCHI, A. R., “Efficiency of the CAPE-OPEN Numerical Open Interfaces”. Technical Reporting, UFRGS, Porto Alegre, Brasil, 2002b.

SOARES, R. DE P.; SECCHI, A. R., “EMSO: A new Environment for Modelling, Simulation and Optimisation”. ESCAPE-13, submitted 2002c.

TSUKANOV, I.; HALL, M., “Fast Forward Automatic Differentiation Library (FFADLib) – A User Manual”. Spatial Automation Laboratory - University of Wisconsin - U.S.A, http://sal-cnc.me.wisc.edu, 2002.

UNGER, J.; KRÖNER, A.; MARQUARDT, W., “Structural Analysis of Differential-Algebraic Equation Systems – Theory and Application”. Computers and Chemical Engineering, 19, No 8, p. 867-882, 1994.

WESTERBERG, A. W., “Process Engeneering: Part II – A System View, Institute for Complex Engineered Systems”. ICES, Carnegie Mellon University Pittsburgh, PA 15213, 1998.

WU, B.; WHITE, R. E., “An initialization subroutine for DAEs solvers: DAEIS”. Computers and Chemical Engineering, 25, p. 301-311, 2001.

Apêndice A
Dados dos Exemplos

No Capítulo 6 foi apresentado um conjunto de exemplos com intuito de demonstrar que as propostas apresentadas nos capítulos anteriores são adequadas. Este apêndice contém os dados omitidos no texto por simplicidade, embora necessários para a solução de tais exemplos.

Separação em um Estágio de Equilíbrio - Seção 6.1.1

- Problema de separação em um estágio de equilíbrio de uma mistura de n-hexano, n-heptano e n-octano;
- Correlação para cálculo do calor específico do vapor: c_p = A+B.T+C.T^2+D.T^3+E.T^4, onde c_p está em kJ/(kmol*K), T em K e constantes obtidas em http://www.cheric.org/kdb;
- A entalpia na fase líquida foi considerada igual à do vapor menos o calor de vaporização, considerado constante com a temperatura (obtido em http://www.cheric.org/kdb);
- Correlação para cálculo das pressões de vapor: ln(P_{vap}) = A.ln(T)+B/T+C+D.T^2, onde P_{vap} está em kPa, T em K e constantes obtidas em http://www.cheric.org/kdb;
- Alimentação a uma temperatura de 300 K, pressão de 2 atm, vazão de 100 kmol/h e composição de [0.5, 0.3, 0.2];
- Temperatura do vaso especificada em 400 K;
- Condição inicial de uma massa interna de 60 kmol e composição igual a da alimentação.

CSTR não-isotérmico - Seção 6.1.2

- Vazão de entrada e de saída iguais a 0.126 m^3/h;
- Alimentação do componente A puro de 9.36 gmol/m^3;
- Sistema isolado (U = 0);
- Calor especifico da mistura de 375 cal/ºC e massa específica de 0.9 gmol/m^3;
- Fator pré-exponencial de 0.8/h e energia de ativação de 28960 cal/mol;
- Condição inicial de um volume total dentro do reator de 5 m^3, concentrações de A e B de 6 gmol/m^3 a uma temperatura de 140 ºC.

Apêndice B

Solução de Sistemas DAE de Índice Elevado

Este apêndice contém uma reprodução do artigo de título *Direct Initialization and Solution of High-Index DAE Systems with Low-Index DAE solvers* enviado em dezembro de 2002 à revista Computers and Chemical Engineering.

DIRECT INITIALIZATION AND SOLUTION OF HIGH-INDEX DAE SYSTEMS WITH LOW-INDEX DAE SOLVERS

R. de P. Soares, A. R. Secchi

Departamento de Engenharia Química - Universidade Federal do Rio Grande do Sul
Rua Sarmento Leite 288/24 - CEP: 90050-170 - Porto Alegre, RS - Brasil
{rafael, arge}@enq.ufrgs.br

Abstract

A method for direct initialization and solution of general differential-algebraic equations (DAE) systems is presented. It consists in a prior index reduction, by adding new variables and equations without manipulation, before the numerical solution of general DAE systems. This approach has shown enhanced robustness to initialize index-one system by reducing it to zero when compared with typical techniques. Furthermore, this method allows the initialization and solution of general high-index DAE systems with solvers designed for low-index problems.

Keywords: High-index DAE systems; DAE initialization; index reduction.

1. Introduction

Differential-algebraic equations (DAE) systems arise naturally from modeling many dynamic systems. This kind of system of equations is more difficult to handle than ordinary differential equations (ODE) systems due to the existence of algebraic constraints. General implicit DAE system can be represented by:

$$F(t, y, y') = 0 \tag{1}$$

where t is the independent variable (normally the time, which is used to refer the variable t hereafter), $F \in \Re^n$, y and $y' \in \Re^n$ are the dependent variables and their time derivatives, respectively. In this notation, there is no distinction for the variables that appear only in the algebraic form.

It is well known that difficulty arises when DAE systems are solved with inconsistent initial values of y and y' and may cause solution failures of many popular DAE solvers (Wu and White, 2001).

The solution of general index-1 DAE systems is, in principle, no much more difficult than the solution of ODE systems, but the initialization can pose problems (Pantelides, 1988).

Actually, the most often failures in solving a DAE system occurs in its initialization for both low- and high-index systems. Furthermore, the solution of high-index problems requires specially designed integration methods, which are usually restricted to a specific problem structure.

For this reason, in this work a method for initializing and solving general DAE systems is presented. This method enhances the robustness for the case of index-one systems and allows the solution of general high-index systems with solvers designed for low-index problems.

2. Index Reduction Method

It is a common sense that the property known as the *index* plays a key role in the characterization of DAE systems. This property can be defined as:

Definition. The minimum number of times that all or part of (1) must be differentiated with respect to t to determine y' as a continuous function of y and t is the *differential index* of the DAE (1).

Others definitions of index can be found in the literature (see Unger *et al.*, 1994). Each definition has its own solvability criterion and there is no guarantee about to produce the same value for the index with different definitions. For the method studied in this work the following definition is preferred:

Definition. The minimum number of times that all or part of (1) must be differentiated with respect to t to get $\partial F / \partial y'$ not structurally singular is the *structural singular index* of the DAE (1).

The literature presents some works regarding the index reduction of DAE systems. Pantelides (1988), Bachmann *et al.* (1990), and Unger *et al.* (1994) have proposed structural algorithms that reduces the index of general DAE systems to 1.

But, as mentioned before, the initialization of index-one problems still can pose problems. These problems are concerned with the initial condition consistency and with the approximate methods required to initialize index-one systems. An alternative is to reduce the index to zero (total index reduction) and solving the non-linear algebraic (NLA) system formed by the reduced system and the initial conditions without using any approximate method, making the stiffness of the problem a unimportant matter.

The total index reduction can be achieved by applying the following steps to a general DAE system:

Step 1. Create a list containing all the time derivative variables of the DAE system;

Step 2. If each variable of the list can be associated with a unique equation of the system, the current system has index zero, and the maximum number of differentiations over all equations is the structural singular index of the original system, finishing the procedure;

Step 3. Try to obtain the missing variables of the list by adding equations to the system (derivatives of the current equations of the system). If the differentiation of the equations generates high-order time derivatives these *new* variables must be added to the list. Return to Step 2.

The application of this method determines the structural singular index of general DAE systems and generates an index-zero equivalent system. The application of this method requires no actual differentiation at all, once structural algebra can be used to determine the relationship equation-variable. Although the method does not require any differentiation or manipulation, when one is seeking for the numerical solution of (1) the equations must be differentiated.

According to the presented method of total index reduction, reduced systems may have *new* variables. Once it is not known whether, in the reduced system, these *new* variables will be considered as new dependent variables or time derivative of existing algebraic variables, a special treatment for such variables is needed in order to obtain the numerical solution.

The following procedure was applied to all *new* variables after performing the total index reduction method.

- If a *new* variable was added to the list and its time derivative was not, then this new variable actually is a *new time derivative variable* and must be added to *y'* while its time integral must be added to *y*, as *new variable*, and the equation $\frac{d}{dt}(\text{original var}) = (\text{new var})$ must be added to the system of equations;
- If a *new* variable and its time derivative were added to the list, then this *new* variable is added to *y* and its time derivative to *y'*.

In the case of not applying this procedure, by considering all high-order time derivatives as *new* variables, the resulting reduced system may not have index zero.

Comparisons between the presented method and typical ones for initialization and solution of some well-established problems are carried out in the next section.

3. Applications

In this section the presented total index reduction method is tested to initialize and solve both low- and high-index DAE problems. The partial derivatives, required for the numerical solutions, were obtained by a built-in automatic differentiation algorithm. Moreover, to get the *new* equations required in the index reduction procedure, a built-in symbolic differentiation algorithm was also used.

For solving the NLA and DAE problems a Newton-like method and DASSLC (Secchi, 1997) were used respectively, the relative tolerance was fixed at 1×10^{-5} and the absolute tolerance at 1×10^{-7} for both solvers. A PENTIUM IV 1.7 GHz with 128 MB of memory was used in the experiments.

3.1. Index-One DAE Initialization

The robustness of the proposed initialization method was tested in initializing the model of a galvanostatic process of a thin film nickel hydroxide electrode (Wu and White, 2001). The system model is given below:

$$\begin{aligned} \frac{\rho V}{W} y_1' &= \frac{j_1}{F} \qquad (a) \\ j_1 + j_2 - i_{app} &= 0 \quad (b) \end{aligned} \tag{2}$$

where

$$\begin{aligned} j_1 &= i_{01}\left[2(1-y_1)\exp\left(\frac{0.5F}{RT}\left(y_2 - \phi_{eq,1}\right)\right) - 2y_1 \exp\left(-\frac{0.5F}{RT}\left(y_2 - \phi_{eq,1}\right)\right)\right] \quad (a) \\ j_2 &= i_{02}\left[2(1-y_1)\exp\left(\frac{F}{RT}\left(y_2 - \phi_{eq,2}\right)\right) - \exp\left(-\frac{F}{RT}\left(y_2 - \phi_{eq,2}\right)\right)\right] \qquad (b) \end{aligned} \tag{3}$$

The dependent variables y_1 and y_2 are the mole fraction of NiOOH and the potential difference at the solid-liquid interface, respectively. The constant values can be found at Wu and White (2001).

For this problem, the list of variables (Step 1) is $[y_1', y_2']^T$. The first variable can be associated with (2a) while the last does not appear in the original system (2). By differentiating (2b) it is obtained an equation, (2b') not shown, containing y_2'. As (2b') does not generate *new* variables (high-order time derivatives) the procedure is finished and the system [(2a), (2b')] is an index-zero DAE system. But not only this two equations must be satisfied, (2b) too.

Therefore, to initialize this system as a NLA problem the variables are $[y_1, y_2, y_1', y_2']^T$ while the equations are $[(2a), (2b), (2b')]^T$. This NLA has one degree of freedom, hence the original DAE system (2) has one *dynamic freedom degree*.

During the integration, to satisfy (2b') means to satisfy (2b) with respect to the integrator tolerances. Actually either equations could be used together with (2a) to integrate the system. By using (2b) an index-one system is formed while using (2b') an index-zero system is got.

Once (2) has one dynamic freedom degree, only one of the two dependent variables can be specified as initial condition. But consistent initial values for the variables are not easy to choose. For instance, a discharged electrode has $y_1 = 0.05$ and $y_2 = 0.38$ but these values do not satisfy the system model.

The Table 1 shows a comparison of the initial guess convergence range for each variable (when the other is given as initial condition) among the presented method and the initialization methods of some popular DAE solvers.

Table 1 - Convergence range in initializing (2).

Solver	$y_2(0)$ convergence range, given $y_1(0)=0.05$	$y_1(0)$ convergence range, given $y_2(0)=0.38$
DASSL*	0.321 – 0.370	0.071 – 0.352
LIMEX*	0.318 – 0.377	0.056 – 0.418
RADAU5*	0.348 – 0.352	0.143 – 0.190
DAEIS*	-0.974 – 1.663	0.0 – 1.0
Proposed Method	**-2.70 – 2.66**	**-∞ – ∞**
Consistent value	0.35024	0.15513

*Data from Wu and White (2001).

As can be seen in the Table 1, the initialization with the index-zero system leads to a method with enhanced robustness. Analogue results are expected for any index-one DAE system.

3.2. High-index DAE initialization

The presented initialization method can also be applied to high-index DAE systems. Aiming to show this capability, a widely used index-three example problem is considered, the pendulum model in Cartesian coordinates:

$$
\begin{aligned}
x' &= w \qquad (a)\\
y' &= z \qquad (b)\\
z' &= T.x \qquad (c)\\
w' &= T.y - g \qquad (d)\\
x^2 + y^2 &= L^2 \qquad (e)
\end{aligned}
\tag{4}
$$

where x and y are the position coordinates, w and z are the velocities in the x and y directions, respectively, T is the cable tension, L is the cable length and g is gravity constant.

The pendulum system modeled as (4) has index three. Applying the presented method, 9 equations (4a', 4a'', 4b', 4b'', 4c', 4d', 4e', 4e'', 4e''') and 6 variables (x'', x''', y'', y''', w'', z'') are created to get an index-zero system.

After the index reduction, the high-index DAE systems (4) can be initialized as usual, by solving a NLA problem. For this case the variables are [x, y, w, z, T, x', y', w', z', T', w'', z'', x'', x''', y'', y''']^T and the equations [(4), 4a', 4a'', 4b', 4b'', 4c', 4d', 4e', 4e'', 4e''']^T. This means 16 variables and 14 equations, hence (4) has two dynamic freedom degrees which must be specified in the initial condition.

The system (4) was successfully initialized for several sets of consistent initial conditions. Results of initializations for some of them can be seen in the Table 2. The values in bold were supplied as initial condition while the others were calculated.

Table 2 - Consistent initial conditions for the pendulum system.

Case	x	y	w	z	T
1	**0.5**	0.87	**0.0**	0.0	8.49
2	**0.5**	0.87	1.74	**-1.0**	4.49
3	**0.0**	1.0	**2.0**	0.0	5.80
4	0.87	**0.5**	0.58	**-1.0**	3.56

Although in Table 2 only the values of the original variable are shown, the presented method also determines the values of the time derivative variables and the *new* variables (high-order derivatives).

3.3. High-Index DAE systems integration

The initialization and time evolution in the solution of high-index DAE systems has been a matter of research in the last two decades. There is some integration codes capable to handle DAE systems of index greater than one, as presented in the works of Ascher and Petzold (1992) and Sand (2002). But these codes are restricted to at most index three and specific problem structure as Hessenberg form or semi-explicit (Unger *et al.*, 1994). In Costa Jr. *et al*. (2001) an automatic index-one reduction technique was presented to initialize and solve general high-index DAE systems. The authors also use a Newton-like method to initialize the index-one extended system.

The index reduction approach presented in the present work turns, *a priori*, any non-singular high-index DAE problem to be solvable with any code designed to implicit ordinary differential equations systems, because the DAE systems are actually converted to implicit ODE systems.

For the solution of the index-three problem given by Eq. (4) as an index-zero equivalent problem, as described previously, 4 *new* variables are required, leading to:

$$y = [x, y, w, z, T, x'', y'', w', z']^T$$

$$y' = [x', y', w', z', T', x''', y''', w'', z'']^T$$

The solution of (4) with its index reduced to zero and using the case 4 (Table 2) as initial condition, can be seen at Fig. 1. This solution was obtained using DASSLC, which is designed for systems with index one at most.

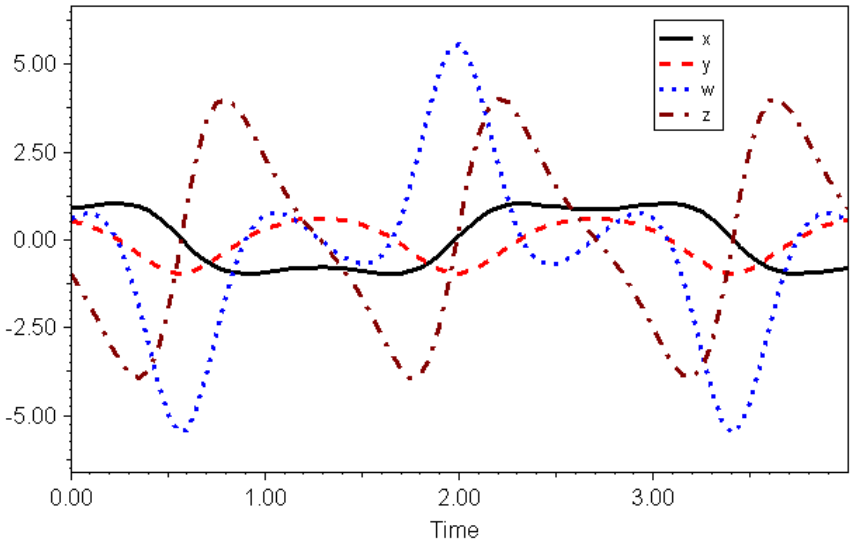


Fig. 1. Solution of the index-three pendulum problem, Eq. (4).

Although Fig. 1 shows only some of the variables, the solution by using the presented method determines the time profiles for all variables *y* and their time derivatives *y'* (including the *new* variables).

4. Index-One or Zero?

The total index reduction of the system (4) leads to an implicit ODE system with 9 variables, while the reduction to index 1 as described by Pantelides (1988) generates a DAE system with only 7 variables. Both of these systems can be used to get the solution of the problem with an index-one DAE solver. The increased size of the index-zero equivalent system is not particular to this example and has been observed for most high-index DAE systems.

The first example of the last section showed that the usage of an index-zero system instead of an index-one tends to be more robust in initializing DAE systems. But for this particular problem, the integration using the index-zero equivalent system was not advantageous, as can be seen at Table 3.

For the pendulum problem (4) different results were obtained, the index-0 formulation has showed best results. Besides, the time evolution in the solution of (4) with some initial conditions (case 2 and 4, Table 2)

using the index-1 equivalent system have failed with non-convergence due to repeated error test failures. For all cases, the usage of the index-0 system in the integration showed several advantages, as can be seen at Table 3.

The comparison index-0/index-1 for the high-index batch distillation column with purity control proposed by Logsdon and Biegler (1993) is also presented in Table 3. Two situations were studied: column with 3 and 12 trays.

For the case of 3 trays, the index-0 formulation requires less residuals and jacobians evaluations, but more CPU time was required in order to obtain the solution. This is expected once the index-0 formulation has 13 *new* variables while the index-1 has only one, and dense linear-algebra was used.

For the case of 12 trays, the usage of the index-1 requires less evaluations and much less CPU time. The increased time difference comes from the same reason that in the last case.

It was observed that the total index reduction can generate ODE systems with much more variables than the index-1 reduced system (e.g., batch distillation column) and the stiffness of the index-0 can be higher in some cases. On the other hand, the solution of some problems is possible only with the index-0 formulation (e.g., some cases of the pendulum problem). Therefore, the choice of the best formulation for the integration of general DAE systems is problem dependent.

It is known that the index is a measure of how *difficult* it is to obtain the solution of a given DAE system. But whether the *easiness* in solving an index-zero instead of an index-one DAE system is compensated or not by the increased size of the index-zero system appears to be a function of both the particular problem and the solver being used. It is likely advantageous to use the index-zero formulation when several re-initializations are necessary during the integration or when problems in using the index-1 formulation are experienced (as in the pendulum problem).

It should be stressed that there are multiple solutions for both the presented method of index reduction and the Pantelides' algorithm. More work is needed in order to elucidate the pending aspects and to develop a method of index reduction which gives the best system of equations (lowest size and stiffness).

Table 3 – Comparisons among the integration of (2), (4) and the index-3 batch distillation column problems using the index-1 and index-0 equivalent systems and DASSLC.

Solution stat. / Problem	Time-points		Residuals evaluation		Jacobian evaluation		Newton-Raphson iterations		Error test failures		Convergence test failures		Rejected time-points		Rejected Newton-Raphson		CPU time (s)	
	index-0	index-1	index-0	index-1	index-0	index-1	index-0	index-1	index-0	index-1	index-0	index-1	index-0	index-1	index-0	index-1	index-0	index-1
(2), t = [0-4000]	118	117	214	230	34	32	214	230	13	6	0	0	13	6	17	6	0.03	0.01
(4), case 1, t = [0-2]	246	482	377	818	28	64	377	818	4	23	0	0	4	23	29	41	0.03	0.03
(4), case 2, t = [0-2]	247	324*	324*	516*	40	41*	390	516*	15	9*	0	10*	15	19*	42	30*	0.05	-
(4), case 3, t = [0-2]	249	507	429	899	46	74	429	899	11	21	0	0	11	21	41	43	0.04	0.04
(4), case 4, t = [0-2]	226	95**	373	190**	36	30**	373	190**	17	9**	0	10**	17	19**	46	20**	0.04	-
column, 3 trays, t = [0-2.1]	97	107	152	212	17	31	152	212	4	11	0	0	4	11	14	33	0.49	0.1
column, 12 trays, t = [0-2.1]	257	149	361	231	27	22	261	231	7	8	0	0	7	8	10	18	13.85	0.651

* Solution failed at time=1.28. ** Solution failed at time=0.347.

5. Conclusions

The major failure cause in solving a DAE system occurs in its initialization (Wu and White, 2001). This fact justifies an index reduction to zero for initialization, once the initialization of index-zero systems has proved enhanced robustness.

The main advantages related to the presented method in initializing DAE systems are: the accuracy does not depend on the step size (exact solution with respect to the tolerances of the used NLA solver is obtained); the system can depends non-linearly on the variable derivatives; the method can be directly applied to systems of any index; not only the values of the variable and their time derivatives appearing in the original system are determined but all variables and their time derivatives; the stiffness of the problem does not matters.

On the other hand, the total index reduction tends to generate extended systems with more variables than the index-one equivalent system. Therefore, the usage of the index-one system in the integration stage appears to be advantageous when no re-initialization is needed, but there is no guarantee. A switch strategy between the index-zero and index-one formulation seems to be a good alternative. That is, the index-zero system is used in the initialization stage, and the index-one is used in the integration stage.

The presented method can turn any solver designed to solve ODE systems or index-one DAE systems capable to solve high-index DAE systems, and its automation requires a symbolic differentiation system but no symbolic manipulation is needed. Furthermore, a solver which makes use of the particular structure of the

index-reduced system (high-order time derivatives are present for some variables) could be designed and will be a matter of future work.

References

U. M. Ascher and L. R. Petzold (1992), *Projected Collocation for High-Order Higher-Index Differential-Algebraic Equations*, Journal Computational and Applied Mathematics, 43, pp 243-259.

R. Bachmann, L Brüll, Th. Mrziglod and U. Pallaske (1990), *On Methods for Reducing the Index of Differential Algebraic Equations*, Computers and Chemical Engineering, Vol. 14, 11, pp 1271-1273.

K. E. Brenan, S. L. Campbell and L. R. Petzold (1996), *Numerical Solution of Initial-Value Problem in Differential-Algebraic Equations,* Classics Appl. Math. 14, SIAM, Philadelphia.

E.F. Costa Jr., R.C. Vieira, A.R. Secchi, E.C. Biscaia (2001), *Automatic Structural Characterization of DAE Systems*, Proceedings of the 11th European Symposium on Computer Aided Process Engineering (ESCAPE 11), Kolding, Denmark, pp. 123-128.

J. S. Logsdon; L. T. Biegler (1993), *Accurate Determination of Optimal Reflux Policies for the Maximum Distillate Problem in Batch Distillation*, Ind. Eng. Chem. Res., Vol. 32 no 4, 692-700.

C. C. Pantelides (1988), *The consistent initialization of differential-algebraic systems*, SIAM J. Sci. Statist. Comput., 9, pp. 213-231.

J. Sand (2002), *On Implicit Euler for High-Order High-Index DAEs*, Applied Numerical Mathematics, 42, pp. 411-424.

A. R. Secchi, F. A. Pereira (1997), *DASSLC: user's manual – v2.0 (Differential-Algebraic System Solver in C),* Departamento de Engenharia Química – UFRGS.

J. Unger, A. Kröner and W. Marquardt (1994), *Structural Analysis of Differential-Algebraic Equation Systems – Theory and Application*, Computers and Chemical Engineering, 19, No 8, pp 867-882.

B. Wu, R.E. White (2001), *An initialization subroutine for DAEs solvers: DAEIS*, Computers and Chemical Engineering, 25, pp 301-311.

Apêndice C

Sistema de Interfaces CAPE-OPEN

Este apêndice contém uma reprodução do artigo de título *Efficiency of the CAPE-OPEN Numerical Open Interfaces* que será enviado à revista Computers and Chemical Engineering.

Efficiency of the CAPE-OPEN Numerical Open Interfaces

R.P. Soares and A.R. Secchi*

Departamento de Engenharia Química - Universidade Federal do Rio Grande do Sul
Rua Sarmento Leite 288/24 - CEP: 90050-170 - Porto Alegre, RS - Brasil
*Author to whom correspondence should be addressed. {rafael, arge}@enq.ufrgs.br

Abstract

The CAPE-OPEN (CO) project have published CORBA and COM open interfaces in order to enable native components of a process simulator to be replaced or plugged from those of independent sources. Until now the major effort of this project is regarded with sharing thermodynamic and unit operation packages. For this reason, the set of interfaces for numerical solvers is not mature yet. In this work the set of CORBA interfaces for sharing numerical solvers is treated. Some inconsistencies found at the original CO set of interfaces were corrected and simplifications were proposed. The performance impact in using the interfaces was studied and the usability of the simplified set of interfaces was tested.

Keywords: CAPE-OPEN; CORBA; continuous-discrete systems.

1. Introduction

Different simulators have different strengths, so that to obtain the best results for a particular problem, access to more than one vendor simulator and in-house software containing specific methods or data is usually required. For this reason, the European Commission under the Industrial and Materials Technologies Program (BRITE-EuRam III, Project BE 3512) sponsored the CAPE-OPEN (CO) project (1997-1999), which aimed to develop, test, describe, and publish agreed standards for interfaces of software components of process simulator. The main objective was to enable native components of a simulator to be replaced by those from another independent source, or those be part of another simulator, with minimal effort in a seamless manner as possible (CO-LAN, 2002). This objective is achieved through the usage of the CORBA and COM/ActiveX middlewares.

The CO project has delivered a set of documents produced by a consortium gathering highly competitive software suppliers, some of their major customers, and group of world-famous research

laboratories. These documents are made available as PDF files on the CAPE-OPEN LAN web site www.co-lan.com. Some of them are:

- *Thermodynamic and Physical Properties Open Interface Specifications;*
- *Unit Operations Open Interface Specifications;*
- ***Numerical Solvers Open Interface Specifications;***
- *Sequential Modular Specific Tools Open Interface Specifications;*

The above documents gives an analysis in terms of UML models (OMG, 2000) before giving the interface descriptions. At the end of the documents are found the CORBA IDL and COM IDL interfaces specifications.

The document *Numerical Solvers Open Interface Specification*, which was produced by the CO Numerical Work package (NUMR), aims to define CO standard interfaces making possible to share numerical solvers.

The structure proposed by NUMR group of the CO project appears to be only conceptual and not too much tested, because the major effort at the market is concerned with *Unit Operation* and *Thermodynamic and Physical Properties* interfaces. Currently there is no vendor that have implemented the numerical set of interfaces as published by the CO project.

In the trial of implementing the CO numerical interfaces (in this work only the CORBA interfaces are treated) some structural problems were encountered. This work aims to propose modifications to the CO pattern in order to solve these problems and to make more simple the usage of the numerical package. Further usability and efficiency analysis of the modified set of interfaces are presented.

2. CO Numerical Interfaces

Flowsheet simulators are designed to calculate the behavior of processes. They take a description of the flowsheet topology and process requirements and assemble a model of the flowsheet from a library of unit operations contained in the simulator.

In the simulation of the model coming from the flowsheet description a simulator is concerned with the solution of three different mathematical problems:

- Systems of linear algebraic equations (LA);
- Systems of nonlinear algebraic equations (NLA);

- Systems of differential-algebraic equations (DAE).

All these problems are relevant to both Sequential/Simultaneous Modular-based and Equation-based packages. The solution of systems involving partial differential and/or integral equations, and of optimization problems is considered to be outside the scope of this work.

2.1. Equation Systems

In order to represent the above listed mathematical problems, a set of interfaces was proposed by the CO project. All these interfaces are based in the abstract concept of *Equation Set Object* (ESO) which is an abstraction of a set of equations.

Clearly, the operations associated with an ESO depends on the type of system equations being represented (LA, NLA or DAE). So, a hierarchical structure of interfaces is proposed by the CO project as shown in Figure 1.

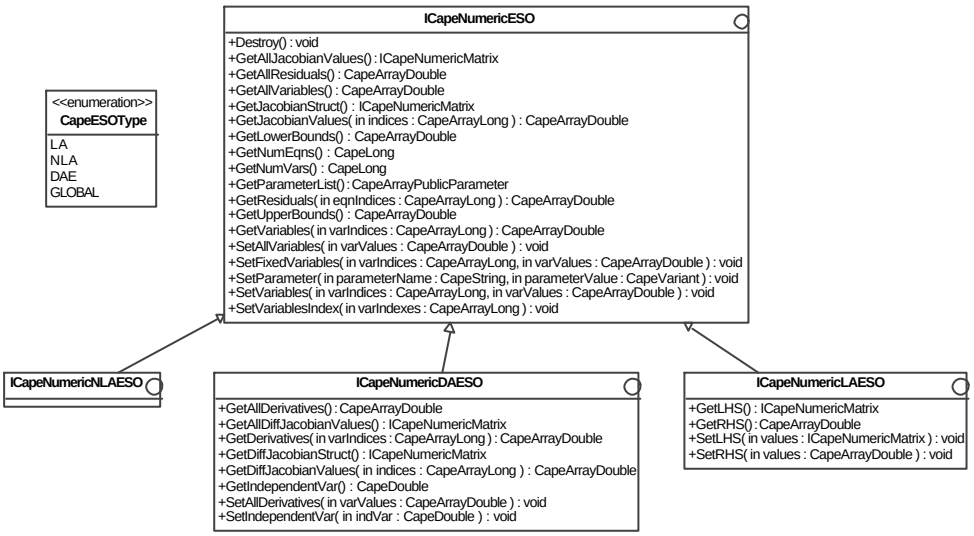


Figure 1. *Original CO interfaces to represent systems of equations.*

An detailed analysis of the interfaces found in Figure *1* reveals some faults, to correct it the following modifications are proposed:

To the enumerator *CapeESOType*:

1. Remove the item *GLOBAL*, once an ESO must be one of *LA*, *NLA*, *DAE* no matter if it is global or not.

To the *ICapeNumericESO* interface:

1. Remove the functions related with the calculation of residuals and jacobians, because these functions make no sense for the derived interface *ICapeNumericLAESO*;
2. Add the missing function *QueryType()* to the interface *ICapeNumericESO*, otherwise it is not possible to known the actual type of an abstract ESO.

To the *ICapeNumericNLAESO* interface:

1. Add the functions for calculate the residuals and jacobian (removed from the base interface *ICapeNumericESO*).

To the *ICapeNumericDAESO* interface:

1. Add the functions for calculate the residuals and jacobian (removed from the base interface *ICapeNumericESO*);
2. Add a convenient function for return the value of the *weighted* jacobian that is the addition of the jacobian with respect to the variables and the jacobian with respect to the variable derivatives times a given weight;

The set of interfaces with the proposed modifications can be seen in Figure *2*.

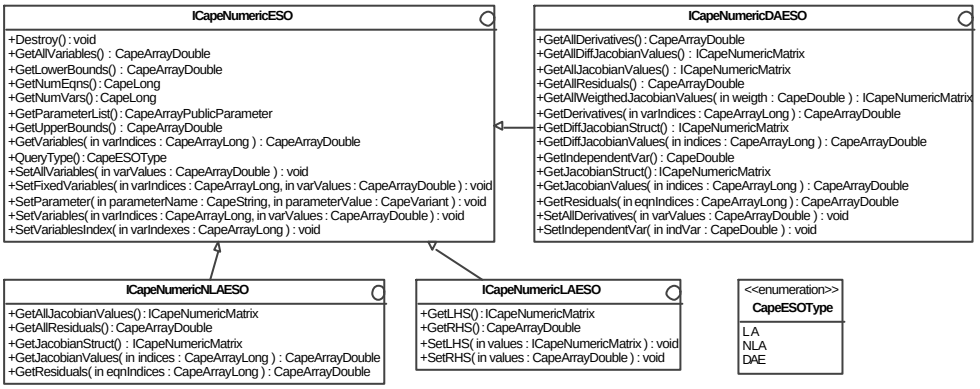


Figure 2. *Interfaces to represent systems of equations with the proposed modifications.*

2.2. Solvers

In an analogous way to the problems, three different types of solvers are needed by a process simulator:

- Linear algebraic (LA) solver;
- Nonlinear algebraic (NLA) solver;

- Differential-algebraic equations (DAE) solver.

Theses types of solvers were modeled as a hierarchical structure of interfaces too, as shown in Figure *3*.

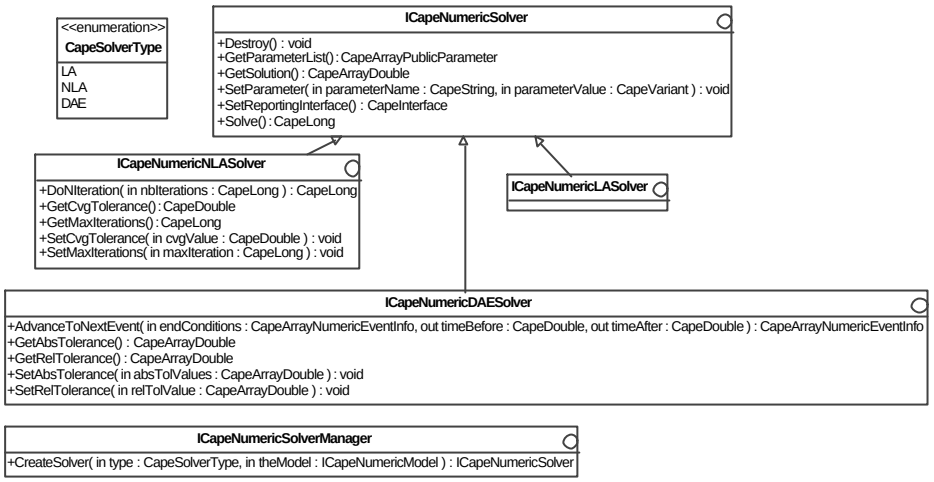


Figure 3. Original CO interfaces to represent solvers.

An implementation of each of these interfaces should contains both the data that characterize the corresponding mathematical problem *and* the numerical algorithms that solve the problem.

In addition to the solvers, the CO project introduces a "manager" interface (bottom of Figure 3). This interface is used to create solvers using information that defines the mathematical problem to be solved by each such instance.

In order to eliminate some inconsistencies found in the CO interface of Figure 3, the following modifications are proposed to the CO solvers structure of interfaces:

To the *ICapeNumericSolver* interface:

1. Remove the function *Solve()*, once it does not make sense for the derived interface *ICapeNumericDAESO*;
2. Add the function *GetParameter()*, allowing to get the parameter for a given name;
3. Correct the function *SetReportingInterface()* which should receive a *CapeInterface* and not return one;
4. Add the function *QueryType()*, otherwise it is not possible to identify the type of a *ICapeNumericSolver* instance.

To the *ICapeNumericLASolver* interface:

1. Add the function *Solve()*, removed from the base interface *ICapeNumericSolver*.

To the *ICapeNumericNLASolver* interface:

1. Remove the functions regarding to get and set solver parameters like *GetMaxIterations()*, that could be handled by the functions *GetParameterList()*, *GetParameter()* and *SetParameter()* of its base interface *ICapeNumericSolver*;
2. Add the function *Solve()*, removed from the base interface *ICapeNumericSolver*;
3. Add the function *SetLASolverFactory()* to supply a different solver manager to create solvers to the LA problems which arise during the solution of a NLA.

To the *ICapeNumericDAESolver* interface:

1. Remove the functions regarding to get and set parameters like *SetAbsTolerance()*, that could be handled by the functions *GetParameterList()*, *GetParameter()* and *SetParameter()* of its base interface *ICapeNumericSolver*;
2. Change the end conditions passed to the function *AdvanceToNextEvent()* to the type *CapeArrayNumericEvent* that simplifies the handling of events. And change the returning data of this function to a *CapeArrayLong* which is a list of the active end conditions;
3. Add functions to supply different solver managers to create solvers to the LA and NLA problems which arise during the solution of a DAE;
4. Add the function *Restart()* to communicate to the solver that the system was reinitialized and the proper actions should be made (e.g. reduce the order for the case of a multi-step DAE solver).

To the *ICapeNumericSolverManager* interface:

1. Change the arguments of the function *CreateSolver()* to the unique information required to create a solver, the problem to be solved.

The set of interfaces for sharing numerical solver with the proposed modifications is found in Figure *4*.

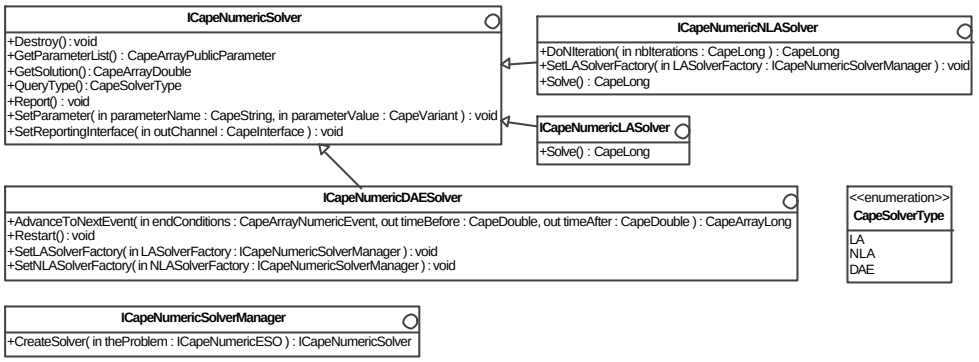


Figure 4. *Modified set of interfaces to represent solvers.*

The item 1 of the modifications proposed for both *ICapeNumericNLASolver* and *ICapeNumericDAESolver* requires a standardization of the names of the most often used parameters. So that these parameters should guarantee to exist on the solver, avoiding the need of querying their existence before setting. Table *1* shows a list of names that can be used as standard names for the parameters of the solvers.

Table 1. *Proposed standard names for parameters of solvers.*

Parameter Name	Parameter Type	Solver Type	Extra Explanation
"AbsoluteTolerance"	*CapeDouble*	*NLA* and *DAE*	Absolute tolerance.
"RelativeTolerance"	*CapeDouble*	*NLA* and *DAE*	Relative tolerance.
"MaxIterations"	*CapeDouble*	*NLA*	Maximum number of iterations.
"MaxIndVarPoints"	*CapeLong*	*DAE*	Maximum number of steps in the independent variable.
"MaxOrder"	*CapeLong*	*DAE*	Maximum order of the integrator.

2.3. Events

Discontinuities are common to occur in dynamic process but the ESO abstraction is a purely continuous mathematical description.

For modeling such cases, a structure where a model is actually a set of ESOs and events is proposed by the CO project. In this approach, the ESOs of a model are switched when a discontinuity occurs, turning active the new system of equations and set of events when a event takes place.

The exact point where a event takes place is determined by the end conditions given to the function *AdvanceToNextEvent()* of the DAE solver interface. In the CO project these end conditions are represented by a hierarchical system of interfaces as shown in Figure 5.

This work proposes a simplified structure for representing continuous-discrete systems. In such structure, neither the *DAESolver* nor the *DAESO* knows anything about what kind of event is about to

happens. The *DAESolver* only have to stop when one or more of the given end conditions (events) became active.

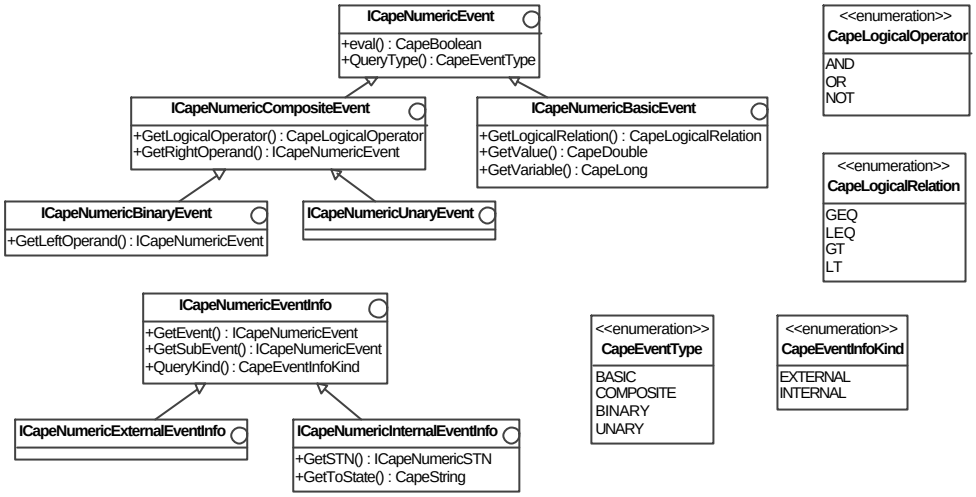


Figure 5. *CO original interfaces for event handling.*

The structure proposed in Figure *5* introduces a set of interfaces and enumerators in the attempt to supply information that is internal to the events. In this work a simplified structure for event detection is proposed, on which there are only two interfaces, as can be seen in Figure *6*. This simplification comes from the fact that what kind of event has occurred is responsibility only of the proprietary implementation and does not need to be public available.

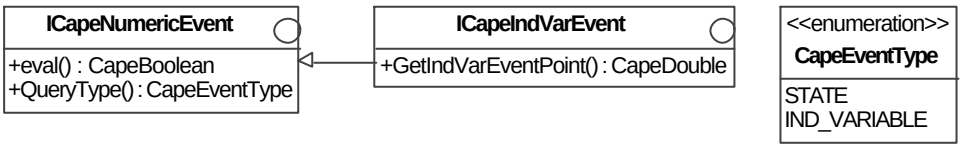


Figure 6. *Modified set of interfaces for event handling.*

The *ICapeNumericEvent* interface found in Figure *6* represents a general event and provides the function *eval()* to check if it has became active or not, returning true if the event has occurred and false otherwise. For representing events that depends only on the independent variable, the derived interface *ICapeIndVarEvent* whit the extra function *GetIndVarEventPoint()* is proposed. This function returns the exact point where the event takes place, avoiding the need of detect its occurrence.

Therefore, to advance the solution of a DAE one has just to call the function *AdvanceToNextEvent()* of the solver with a list of events (*CapeArrayNumericEvent*), then a list of the indexes of the active events is returned by the solver.

Based on this simplified structure for event handling, a simplified set of interfaces for merging continuous-discrete models is proposed in the next section.

2.4. Merging of Models

The main purpose of the CO project is to enable native components of a simulator to be directly replaced or plugged by those from another independent source. This comprises the merging of models from different sources to build a global model.

As described before, an ESO is a purely continuous abstraction, but a model for process engineering is frequently a continuous-discrete system. To represent this kind of process, a structure on which a model is a set of ESO systems and events is proposed by the CO project, as can be seen in Figure *7*.

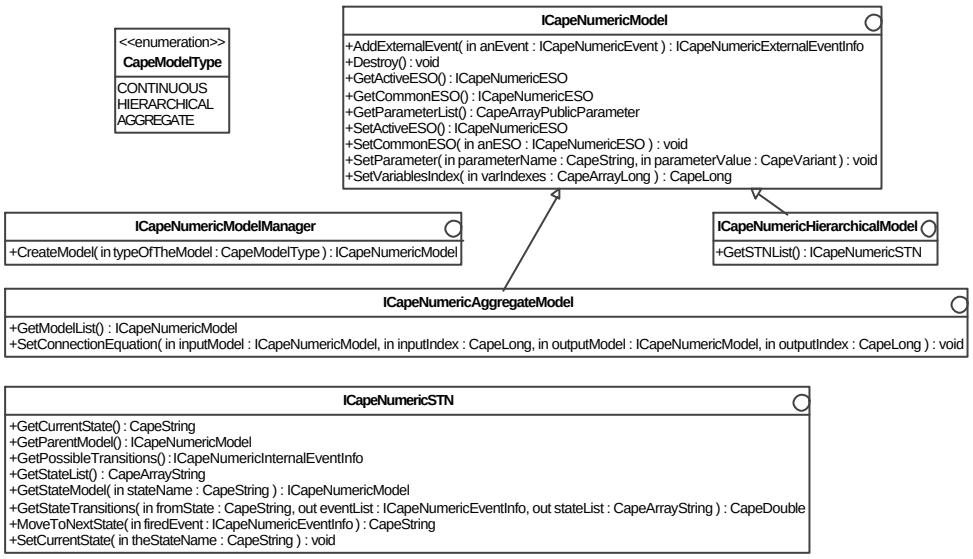


Figure 7. Original CO interfaces for representing and merging continuous-discrete models.

Once more the CO project proposes a structure where information considered by this work as required only for internal implementation is supplied. So, a simplified but not less general structure is proposed. This structure is shown in Figure *8* and allows the description and merging of continuous-discrete models with only two simple interfaces.

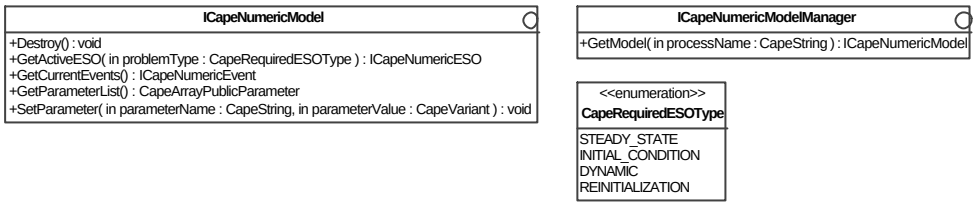


Figure 8. Set of interfaces proposed for representing and merging continuous-discrete models.

With the structure proposed in Figure *8* a CO compliant process simulator has to provide a *ICapeNumericModelManager* interface. With a reference for a instance of such interface a client can ask for a model with a given name using the member function *GetModel()*.

Figure *9* shows a C++ piece of code that demonstrates the usage of the proposed architecture in simulating an external model with external solvers.

```
// Getting the external model for a given name
ICapeNumericModel_var external_model =
  external_model_manager->GetModel("CSTR_Reactor");

// Temporary CORBA object to narrowing interfaces
CORBA::Object_var temp;

// Getting the ESO for calculate the initial condition
temp = model->GetActiveESO(INITIAL_CONDITION);
ICapeNumericNLAESO_var initial = ICapeNumericNLAESO::_narrow(temp);

// Creating a NLA solver and solving the system
temp = external_solver_manager->CreateSolver(initial);
ICapeNumericNLAESOSolver_var initalSolver = ICapeNumericNLAESOSolver::_narrow(temp);
initalSolver->Solve();initalSolver->Destroy();

// Getting the ESO to advance in the solution
temp = model->GetActiveESO(DYNAMIC);
ICapeNumericNLAESO_var dae = ICapeNumericDAESO::_narrow(temp);

// Creating a DAE solver and advancing in the solution
temp = external_solver_manager->CreateSolver(dae);
ICapeNumericDAESOSolver_var daeSolver = ICapeNumericDAESOSolver::_narrow(temp);
daeSolver->AdvanceToNextEvent(endConditions,tafter,tbefore);
daeSolver->Destroy();
```

Figure 9. C++ piece of code demonstrating the sharing of models, solvers and ESOs.

The piece of code of Figure *9* is for demonstrating purposes only and will not directly compile, a complete version can be obtained by contacting the authors.

A continuous-discrete model is represented by the interface *ICapeNumericModel*. Upon this interface a client can get the current continuous-system of equations through the function *GetActiveESO()*. This function can return three different ESOs (depending on its input argument): a NLA to initialize the model, a DAE to advance in the independent variable or a NLA to reinitialize the model in the case of an event detection.

The function *GetCurrentEvents()* returns a list of events that determines when the current active ESOs has been changed. Therefore, for use or merge external models one can ask to one or more instances of the interface *ICapeNumericModelManager* for the required models. Once all the desired models are available, their current ESO systems and events can be concatenated (together with new equations that are the connections between the models and specifications of some of the variables) and encapsulated in a new *IcapeNumericModel*, as exemplified in the next section.

3. Tests of Efficiency and Usability

In the prior section, several modifications for the CO numerical open interfaces were proposed. Tests of usability of the modified set of interfaces and some efficiency tests follows.

For implementing the interfaces, the omniORB IDL compiler (Grisby, 2000) was used to create stubs in C++ from the CORBA IDL interfaces. Upon the generated skeleton code a C++ code was written for implementing the interfaces. The omniORB (Lo *et al*., 2002) was used as object request broker. For test the implementations a PENTIUM 4, 1.8 GHz with 128 MB of memory running a win32 platform was used.

3.1. Efficiency Comparison

According to OMG (1999b), when variable length data (the case of a *CAPEArrayDouble*) is returned by a member function of a interface, new memory is allocated for it and the caller is responsible to deallocate it, otherwise memory leak occurs.

Therefore, for the interfaces showed in Figure *1* and Figure *2*, each time that a solver requires the evaluation of the residuals of an ESO, by using the functions *GetAllResiduals()* or *GetResiduals()*, memory for a new *CAPEArrayDouble* is allocated, and then it must be released after use. Analog behavior is observed for the jacobian evaluation represented by the functions *GetAllJacobianValues()* or *GetJacobianValues()*. These allocation-deallocation procedure required in each function call generates a great impact on the efficiency when compared with systems where direct memory access is granted.

In order to test the efficiency of the interfaces the following alternative function for calculating the residuals of a NLA problem is proposed:

```
PutAllResiduals(CapeDouble *res, CapeDouble *varValues)
```

By calling this function the residuals are stored on the given memory address `res` which are calculated based on the given variable values memory address `varValues` (clearly this function can be used only on direct memory access implementations). Table *2* shows the C++ pieces of different codes used to compare the CPU time in calculating the residuals of a NLA system.

Table 2. C++ pieces of codes for efficiency test in calculating the residuals of a NLA system.

Code 1

```
//getting the interface of a local nla
nla_iface = nla_local->_this();

//evaluating the residuals 100 times
for(i=0;i<100;i++){
  nla_iface->SetAllVariables(*y);
  out = nla_iface->GetAllResiduals();
  *res = *out;
  delete out;
}
```

Code 2

```
//getting the interface of a external nla
nla_iface =
   ext_model->getActiveESO(STEADY_STATE);

//evaluating the residuals 100 times
for(i=0;i<100;i++){
  nla_iface->SetAllVariables(*y);
  out = nla_iface->GetAllResiduals();
  *res = *out;
  delete out;
}
```

Code 3

```
//evaluating the residuals 100 times
for(i=0;i<100;i++){
  nla_local->PutAllResiduals(res,y);
}
```

The codes 1 and 2 of Table *2* uses the CO interfaces while the Code 3 is a usual implementation with direct memory access. Should be stressed that, in the case of using interfaces, to correct emulate a NLA solver, the variable values must be updated with the function *SetAllVariables()* before calling the function *GetAllResiduals()* and after this call the calculated residuals must be copied and deleted.

The NLA problems considered in the above codes has 100.000 variables. The three codes were tested: Code 1 (uses the interfaces but in the same address space), Code 2 (different address spaces but same machine) and Code 3. The results are shown in Table *3*.

Table 3. CPU time for calculating the residuals of a NLA system, using the codes of the Table 2.

Code 1 (same address)	Code 2 (different addresses)	Code 3
1.42 sec.	5.43 sec.	0.44 sec.

The efficiency loss between codes 1 and 3 of Table *3* is mainly related with the extra memory allocation and copy required by the usage of the interfaces. In the Code 2 an additional delay is experienced because of the data transferring between the two address spaces.

In practical implementations the Code 2 is used, what means a considerable efficiency loss of more than 10 times when compared with a implementation with direct memory access. Naturally, this

implementation is advantageous when components of different process simulators have to be used to solve the problem or when the problem is sufficiently parallelizable.

3.2. Continuous-Discrete Simulation

For testing the capability of the set of interfaces introduced by Figure *6* consider the continuous-discrete process proposed by Wu and White (2001). This process consists in the model of a galvanostatic charge/open-circuit/discharge circuit (details can be found at the original work). A continuous-discrete process takes place when the following procedure (thirty cycles) is applied to the continuous model of the system:

1. Charge with an applied current of 1×10^{-5} for 500 s with 100 s more for each cycle. Terminate the process if the potential reaches 0.6 V;
2. Open-circuit for 500 s. Terminate the process if the potential reaches 0.25;
3. Discharge with an applied current of -1×10^{-5} for 1000 s. Terminate the process if the potential reaches 0.25.

The above process evolves both kinds of events (independent and dependent variables). Results of the simulation of this continuous-discrete system using the proposed set of interfaces and the EMSO simulation environment (Soares and Secchi, 2002) can be seen in Figure *10*.

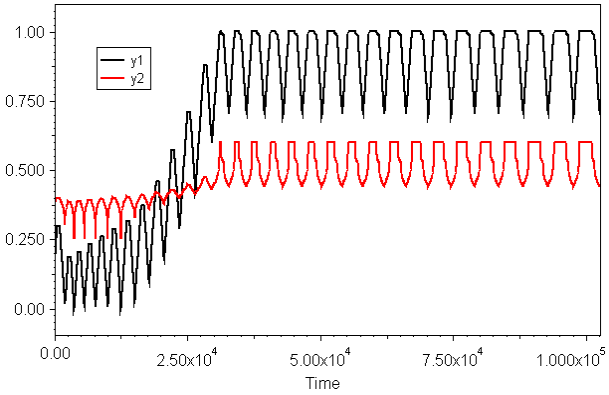


Figure 10. *Simulation of a continuous-discrete system using the proposed set of interfaces for event detection.*

3.3. Merging of Models

In order to test the capability of the proposed structure in Figure *8* for merging models from different sources, consider the chemical process outlined in Figure *11*. This process evolves several devices connected each other.

For demonstrating purposes, the flowsheet of Figure *11* was broken into two pieces (A and B). A CO compliant software should be able to get each one of these pieces from external and heterogeneous sources and simulate the entire flowsheet.

The process showed in Figure *11* was successfully solved using the EMSO simulator, which implements the set of interfaces proposed in Figure *8*, in the following way:

1. The process pieces A and B were loaded in two different EMSO instances, leading to one rectangular systems of equations in each instance;
2. These systems were obtained through the interfaces by a third EMSO instance and concatenated (wrapped in a new system of equations) leading to a system of equations still rectangular;
3. The missing connections (3 and 7, of Figure *11*) were included to the concatenated system by the addition of equality equations.
4. With the resultant square system of equations the simulations were carried out as usual.

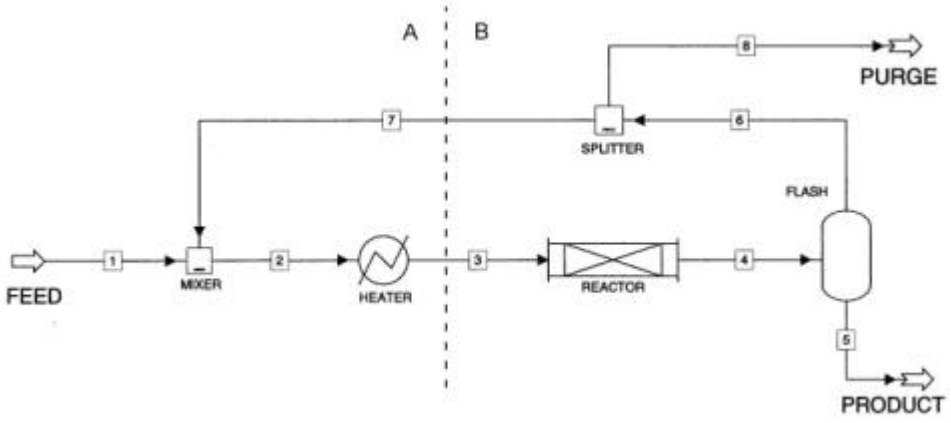


Figure 11. *Flowsheet of a methanol plant.*

Should be stressed that, in the EMSO environment, the *local* connections between devices (connections 1, 2, 4, 5, 6 and 8 of Figure *11*) of a flowsheet do not generate any extra equality equation as done with the non-local connections. In such cases the input variables are only references to the output ones, resulting in reduced-size systems. But in the case of connections between heterogeneous systems

(CO interfaces), the addition of equations is mandatory, once only values are allowed to pass through the interfaces and not memory addresses.

4. Conclusions

Several modifications to the set of CO numerical open interfaces were proposed. These modifications correct some inconsistencies and make the interfaces for sharing systems of equations and equation solvers more concise and reliable. Besides a simplified set of interfaces is proposed for merging continuous-discrete systems between heterogeneous systems. These modifications leads to a system much simpler but not less general.

Usability tests with the modified set of interfaces has proved that they are suitable for sharing the numerical problems arising in process simulators.

A comparison when using the CO interfaces and when direct memory access is granted was made. These tests reveal a high efficiency loss in using such interfaces. Therefore, the indiscriminate usage of the CO numerical interfaces is prohibitive for large systems.

References

OMG, *The Common Object Request Broker: Architecture and Specification*, version 2.3.1, www.omg.org (1999).

OMG, *OMG Unified Modeling Language Specification*, version 1.3, www.omg.org (2000).

OMG, *C++ Language Mapping*, version 2.3, www.omg.org (1999b).

CO-LAN, *Conceptual Design Document for CAPE-Open Project*, www.co-lan.org (2002).

CO-LAN, *Open Interface Specification - Numerical Solvers*, www.co-lan.org (2002b).

D. Grisby, *omniIDL – The omniORB IDL Compiler*, AT&T Laboratories, Cambridge, http://omniorb.sourceforge.net (2000).

S. Lo, D. Riddoch and D. Grisby, *The omniORB version 3 Users Manual*, AT&T Laboratories, Cambridge, http://omniorb.sourceforge.net (2000).

B. Wu, R.E. White (2001), *An initialization subroutine for DAEs solvers: DAEIS*, Computers and Chemical Engineering, 25, pp 301-311.

R. de P. Soares, A.R. Secchi, *EMSO: A new Environment for Modelling, Simulation and Optimisation*, ESCAPE-13, (2003).

Apêndice D

EMSO

Este apêndice contém uma reprodução do artigo de título *EMSO: A new Environment for Modelling, Simulation and Optimisation* enviado em 2002 e aceito para apresentação no 13º ESCAPE.

EMSO: A New Environment for Modelling, Simulation and Optimisation

R. de P. Soares and A.R. Secchi*
Departamento de Engenharia Química - Universidade Federal do Rio Grande do Sul
Rua Sarmento Leite 288/24 - CEP: 90050-170 - Porto Alegre, RS - Brasil
*Author to whom correspondence should be addressed. {rafael, arge}@enq.ufrgs.br

Abstract

A new tool, named EMSO (**E**nvironment for **M**odelling, **S**imulation and **O**ptimisation), for modelling, simulation and optimisation of general process dynamic systems is presented. In this tool the consistency of measurement units, system solvability and initial conditions consistency are automatically checked. The solvability test is carried out by an index reduction method which reduces the index of the resulting system of differential-algebraic equations (DAE) to zero by adding new variables and equations when necessary. The index reduction requires time derivatives of the original equations that are provided by a built-in symbolic differentiation system. The partial derivatives required during the initialisation and integration are generated by a built-in automatic differentiation system. For the description of processes a new object-oriented modelling language was developed. The extensive usage of the object-oriented paradigm in the proposed tool leads to a system naturally CAPE-OPEN which combined with the automatic and symbolic differentiation and index reduction forms a software with several enhancements, when compared with the popular ones.

1. Introduction

Simulator is a valuable tool for applications ranging from project validation, plant control and operability to production increasing and costs reduction. This facts among others has made the industrial interest in softwares tools for modelling, simulation and optimisation to grow up, but this tools are still considered inadequate by the users (Che-Comp, 2002). The user dissatisfaction is mainly related with limited software flexibility, difficulty to use/learn and costly. Besides the lack of software compatibility and the slowness of inclusion of new methods and algorithms. Furthermore, the users have been pointed out some desired features for further development, like extensive standard features, *intelligent* interfaces among others (Hlupic, 1999).

In this work a new tool for modelling, simulation and optimisation of general dynamic systems, named EMSO (**E**nvironment for **M**odelling, **S**imulation and **O**ptimisation) is presented. This tool aims to give the users more flexibility to use their available resources. The successful features found in the most used tools were gathered and some new methods were developed to supply missing features. In addition, some well established approaches from other areas were used, like the object-oriented paradigm. The big picture of the EMSO structure is shown at Figure 1 which demonstrates the modular architecture of the software.

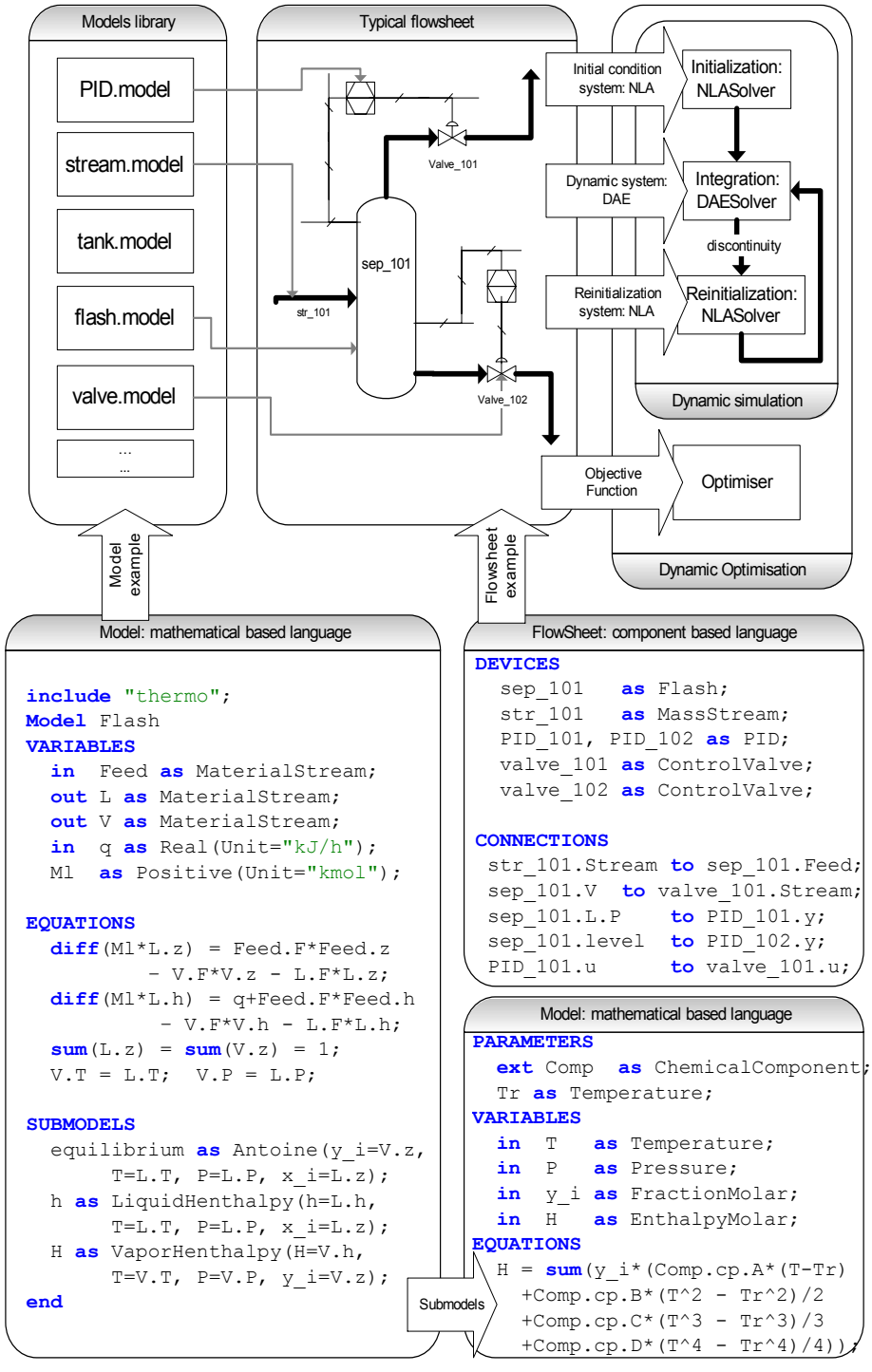


Figure 1. General vision of the EMSO structure and its components.

2. Process Model Description

In the proposed modelling language there are three major entities: *models*, *devices*, and *flowsheets*. *Models* are the mathematical description of some *device*; a *device* is an instance of a *model*; and a *flowsheet* represents the process to be analysed which is composed by a set of *devices*. At bottom of Figure 1 are given some pieces of code which exemplifies the usage of the language.
EMSO makes intensive use of automatic code generators and the object-oriented paradigm whenever is possible, aiming to enhance analyst and productivity.

2.1 Model

In the EMSO language, one *model* consists in the mathematical abstraction of some real equipment, process piece or even software. Examples of *models* are the mathematical description of a tank, pipe or a PID controller.
Each *model* can have parameters, variables, equations, initial conditions, boundary conditions and submodels that can have submodels themselves. *Models* can be based in pre-existing ones, and extra-functionality (new parameters, variables, equations, etc.) can be added. So, *composition* (*hierarchical modelling*) and *inheritance* are supported.
Every parameter and variable in a *model* is based in a predefined *type* and have a set of properties like a *brief* description, *lower* and *upper* bounds, *unit* of measurement among others. As *models, types* can have subtypes and the object-oriented paradigm is implemented. Some examples of *types* declarations can be seen in Figure 2.

```
Fraction as Real(Lower=0, Upper=1);
Positive as Real(Lower=0, Upper=inf);
EnergyHoldup as Positive(Unit="J");
ChemicalComponent as structure (
  Mw as Real(Unit="g/mol");
  Tc as Temperature(Brief="Critical Temperature");
  Pc as Pressure;
);
```

Figure 2. Examples of type declarations.

2.2 The Flowsheet and its Devices

In the proposed language a *device* is an instance of a *model* and represents some *real* device of the process in analysis. So, a unique *model* can be used to represent several different *devices* which have the same structure but may have different conditions (different parameters values and specifications). *Devices* can be connected each other to form a *flowsheet* (see Figure 1) which is an abstraction for the real process in analysis.
Although the language for description of *flowsheets* is textual (bottom right in Figure 1), it is simple enough to be entirely manipulated by a graphical interface. In which *flowsheets* could be easily built by dragging *model* objects into it to create new *devices* that could be connected to other *devices* with the aid of some pointing unit (mouse).

3. Consistency Analysis

In solving the resulting system of differential-algebraic equations (DAE) of a *flowsheet*, prior analysis can reveal the major failure causes.
There are several kinds of consistency analysis which can be applied in the DAE system coming from the mathematical description of a dynamic process. Some of them are: measurement units, structural solvability and initial conditions consistency.

3.1 Measurement Units Consistency

In modelling physical processes the conversion of measurement units of parameters is a tiresome task and prone to error. Moreover, a ill-composed equation usually leads to a measurement unit inconsistency. For this reasons, in EMSO the measurement units consistency and units conversions are automatically made for all equations, parameter setting and connections between *devices*.
Once all expressions are internally stored in a symbolical fashion and all variables and parameters holds its measurement units, the units consistency can be easily tested with the aid of the units measurement handling package RUnits (Soares, 2002).

3.2 DAE Solvability

Soares and Secchi (2002) have proposed a structural method for index reduction and solvability test of DAE systems. With this method, structural singularity can be tested and the structural differential index can be reduced to zero by adding new variables and equations. Such variables and equations are the derivatives of the original ones with respect to the independent variable.
EMSO makes use of this method, allowing the solution of high-index DAE problems without user interaction. The required derivatives of the variables and equations are provided by a built-in symbolic differentiating system.

3.3 Initial Conditions Consistency

Once a DAE system is reduced to index zero the dynamic freedom degree is determined. So, the initial condition consistency can be easily tested by an association problem as described by Soares and Secchi (2002). This approach is more robust when compared with the index-one reduction technique presented by Costa et al. (2001).

4. External Interfaces

Usually each simulation software vendor has its proprietary interfacing system, this leads to heterogeneous systems. Recently, the CAPE-OPEN project (CO-LAN, 2002) has published open standards interfaces for computer-aided process engineering (CAPE) aiming to solve this problem. EMSO complies with this open pattern. The interfaces are implemented natively rather than *wrapping* some other proprietary interface mechanism, and CORBA (OMG, 1999) was used as the middleware.
The extensive usage of the interfaces turns its efficiency a priority. For this reason some modifications for the numerical CAPE-OPEN package (CO-LAN, 2002) were proposed. This modifications consists in changing some function calling conventions, more details can be seen in Soares and Secchi (2002b).

5. Graphical User Interface

The graphical user interface (GUI) of EMSO combines *model* development, *flowsheet* building, process simulation and results visualising and handling all in one. EMSO is entirely written in C++ and is designed to be very modular and portable. In running tasks there are no prior generation of intermediary files or compilation step, everything is made out at the memory. The software is multithread, allowing real-time simulations and even to run more than one *flowsheet* concurrently without blocking the GUI. Furthermore, calculations can be paused or stopped at any time. The Figure 3 shows the EMSO GUI, it implements a Multiple Document Interface (MDI).

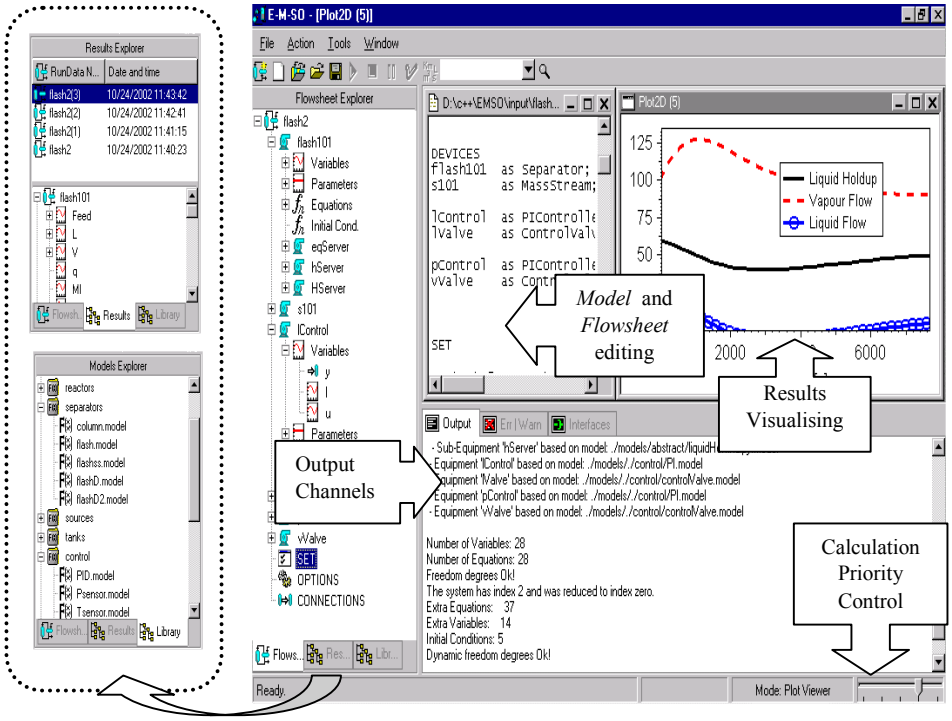


Figure 3. EMSO graphical user interface.

6. Applications

Consider the separation process outlined at *Figure 1*. It consists in a flash vessel with level and pressure control and was modelled as:

$$\frac{d}{dt}(Ml.x_i) = F.z_i - L.x_i - V.y_i \qquad \frac{d}{dt}(Ml.h) = q + F.h_F - L.h - V.H \tag{1a}$$

$$\sum_i x_i = \sum_i y_i = 1 \qquad y_i = K_i . x_i \tag{1b}$$

$$K_i = f_1(T, P, x_i, y_i) \tag{2}$$

$$h_F = f_2(T_F, P_F, z_i), \quad h = f_3(T, P, x_i), \quad H = f_4(T, P, y_i) \tag{3}$$

$$V = f_5(P), \quad L = f_6(Ml) \tag{4}$$

where F, V and L are the feed, vapour and liquid molar flow rates, h_F, H and h are the respective molar enthalpies, T is the temperature and P pressure, z_i, x_i and y_i are the feed, liquid and vapour molar fractions and q is the heat duty.

In EMSO this process can be modelled in a modular fashion: Eq.(1) as *flash device*; Eq.(2) as *thermodynamic equilibrium device*; Eq.(3) as *enthalpy devices*; Eq.(4) as *control devices*.

The dynamic simulation of this system is trivial if q and the feed conditions are known along the time and T, Ml and x_i are given as initial conditions. But if a temperature profile is specified instead of q a high-index system take place and cannot be directly solved by the popular simulation tools.

In solving this problem EMSO reports a index two system and automatically reduces the index to zero. In the Figure 4 the composition profiles of the outlet streams are showed for a start-up condition of this index two system.

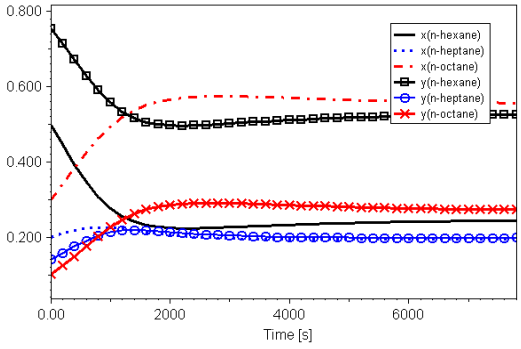


Figure 4. Solution of the index two separation process.

The index-three batch distillation column proposed by Logsdson and Biegler (1993) was also successfuly solve by EMSO, but there was no room to show the results.

7. Conclusions

An object-oriented language for modelling general dynamic process was successfully developed and its usage has proved efficiency in code reusability. The development of model libraries of models for thermodynamics, process engineering and other application areas is one of the future tasks. The DAE index reduction method allows

EMSO to directly solve high-index DAE systems without user interaction. This fact combined with the symbolic and automatic differentiation systems and the CAPE-OPEN interfaces leads to a software with several enhancements.
Dynamic optimisation is at design stage, but the modular internal architecture of EMSO allows it to be added further without re-coding the other modules.

8. References

Che-Comp, 2002, The State of Chemical Engineering Softwares, www.che-comp.org .

CO-LAN, 2002, Conceptual Design Document for CAPE-Open Project, www.co-lan.org .

Costa Jr., E.F., R.C. Vieira, A.R. Secchi and E.C. Biscaia, 2001, Automatic Structural Characterization of DAE Systems, ESCAPE 11, Kolding, Denmark, 123-128.

Hlupic, V., 1999, Simulation Software: User's Requirements, Comp. Ind. Engrg, 37, 185-188.

Logsdon, J. S. and Biegler, L. T., 1993, Ind. Eng. Chem. Res., v.32, n.4, 692-700.

Luca, L. De and Musmanno, R., 1997, *A* Parallel Automatic Differentiation Algorithm for Simulation Models, Simulation Practice and Theory, 5, 235-252.

OMG, 1999, The Common Object Request Broker, version 2.3.1, www.omg.org.

Soares, R. de P. and Secchi, A. R., 2002, Direct Initialization and Solution of High-Index DAE Systems with Low-Index DAE solvers, Comp. Chem. Engrg (submitted 2002).

Soares, R. de P. and Secchi, A. R., Efficiency of the CAPE-OPEN Numerical Open Interfaces, Technical Reporting, UFRGS, Porto Alegre, Brasil (2002b).

Soares, R. de P, Runits: the Measurement Units Handling Tool - Reference Manual, rafael@enq.ufrgs.br (2002).