PUC

MARCELO TÍLIO MONTEIRO DE CARVALHO

UMA ESTRATÉGIA PARA DESENVOLVIMENTO DE APLICAÇÕES CONFIGURÁVEIS EM MECÂNICA COMPUTACIONAL

TESE DE DOUTORADO

Departamento de Engenharia Civil
PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Rio de Janeiro, Junho de 1995

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Marcelo Tilio Monteiro de Carvalho

Uma Estratégia para Desenvolvimento de Aplicações Configuráveis em Mecânica Computacional

Tese apresentada ao Departamento de Engenharia Civil da PUC-Rio como parte dos requisitos para a obtenção do título de Doutor em Ciências em Engenharia Civil. Ênfase: Estruturas

Orientador: Luiz Fernando Martha

Departamento de Engenharia Civil
Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 29 de junho de 1995

Ao meu avô Roberto Tilio (*in memorian*) e
a meu tio Joac Roberto (*in memorian*)

Agradecimentos

Ao professor Luiz Fernando Martha, pela orientação, apoio e confiança depositada ao longo deste trabalho. E, mais importante do que tudo, pela amizade e dedicação em todas as etapas deste trabalho, especialmente durante o período na Universidade de Cornell.

Ao professor Marcelo Gattass, responsável pela minha decisão de fazer o doutorado. Sua conduta profissional sempre foi motivo de inspiração e seu apoio constante foi fundamental para a realização deste trabalho.

Ao professor Anthony R. Ingraffea, pela amizade, apoio e pelas excelentes condições de trabalho que tive na Universidade de Cornell.

Ao Dr. Paul Wawrzynek, responsável direto por este trabalho, pela orientação e aprendizado, durante o trabalho na Universidade de Cornell, onde desenvolvi a maior parte da minha pesquisa.

Aos meus pais, pelo carinho, apoio e pelo exemplo sempre demonstrado.

Ao Dr Bruce Carter, companheiro de *office*, pelas discussões, sugestões e apoio, durante o período na Universidade de Cornell.

Aos membros do grupo CFG (Cornell Fracture Group), em especial à David Potyondy e Will Ridell.

Ao amigo Waldemar Celes Filho, pela participação em todas as etapas desta pesquisa, com valiosas discussões e sugestões.

Ao amigo Eduardo Setton da Silveira, pela implementação e testes de diversas idéias, sem a qual não teria sido possível a finalização deste trabalho.

Ao amigo Ivan Menezes, pela amizade e apoio fundamentais nos momentos mais difíceis deste período.

À equipe do TeCGraf (*Grupo de Tecnologia em Computação Gráfica*), em especial a Luiz Henrique de Figueiredo, Prof. Roberto Ierusalimschy, Luiz Cristovão Gomes Coelho e Carlos Henrique Levy, pelas constantes discussões, idéias e sugestões.

Aos funcionários do Departamento de Engenharia Civil e do TeCGraf, pela colaboração, eficiência e amizade. Em especial à Ana Roxo, Bianca, Yedda, D. Neuza, Santana e Claudinei.

Ao CNPq, CAPES e ao TeCGraf, pelo apoio financeiro.

Resumo

Muitos sistemas utilizados para simulações de mecânica computacional (análise térmica, análise de tensões, etc.) são específicos para uma aplicação em particular, sendo portanto difícil a sua utilização em novas aplicações, ou são muito grandes e genéricos, dificultando o uso e a manutenção. Por outro lado, existem vários aspectos da simulação, como modelagem geométrica, discretização e visualização, que são independentes do tipo de análise e poderiam ser utilizados em diversas aplicações.

Numa tentativa de minimizar este problema e permitir um melhor aproveitamento das tecnologias referentes aos aspectos comuns da simulação, alguns sistemas utilizam uma modelagem baseada em geometria. Nesse procedimento, os atributos (informações adicionais à descrição geométrica, necessárias para a completa definição do problema) são associados à geometria do modelo ao invés de serem associados às entidades da discretização. Apesar de ser mais adequada a problemas complexos, a forma como esta abordagem é implementada, geralmente, não é muito flexível. É difícil adicionar novas funcionalidades para atender a diferentes tipos de simulações, pois qualquer modificação requer um conhecimento detalhado do código da implementação.

Esta tese propõe um sistema que busca oferecer, com alto grau de abstração, um ambiente para se realizar simulações de mecânica computacional, que seja independente do tipo de problema e possa, ao mesmo tempo, ser facilmente estendido e configurado, para atender a requisitos específicos. A especificação dos atributos é feita de forma independente da descrição geométrica do modelo e de sua discretização. Para isto, utiliza-se uma estruturação de classes, dentro do paradigma de orientação a objetos, que são responsáveis pela interface entre os atributos e o modelador geométrico. O sistema utiliza uma linguagem de extensão para a configuração da aplicação, que permite a construção de novas classes sem a necessidade de recompilação do sistema. Uma das metas importantes do sistema proposto é a de poder ser implementado não só em novos programas, mas também em códigos já existentes. Isto permite aplicações tornarem-se

configuráveis, mesmo não tendo sido concebidas originalmente com essa finalidade.

A tese apresenta, inicialmente, uma discussão sobre os aspectos envolvidos em simulações de mecânica computacional, discutindo-se os problemas existentes. Apresenta-se, como ilustração de um sistema existente e motivação para o sistema proposto, uma descrição de um programa para simulação de propagação de trincas tri-dimensionais (FRANC3D). A seguir, é feita uma discussão, em termos gerais, de re-uso de código e configuração de aplicação. Apresenta-se, então, a arquitetura do sistema proposto, e uma breve descrição das classes existentes. Por fim, são apresentados dois ambientes em que o sistema foi implementado, e alguns exemplos para ilustrar a utilização do sistema.

Abstract

Most current computational mechanics systems (e.g., termal analysis, stress analysis) are either specific for one particular application, and thus difficult to reconfigure for new applications, or huge and general, which makes them difficult to use and maintain. On the other hand, there are many aspects of the simulation process (e.g., geometric modeling, discretization and visualization), that do not depend on the type of analysis being studied and could be used by different applications.

Attempting to minimize this problem and take better advantages of the common features of the simulation, a geometry-based modeling approach can be used. In such an approach, the attributes (additional information needed to describe the physics of the problem) are assigned to geometry rather than directly to nodes and elements. Although this approach is much better to support the increasing complexity of computational mechanics simulations, the way it is usually implemented is not very general or flexible. It is often difficult to add new features or extend software capabilities to deal with new applications. Moreover, this usually requires complete knowledge of the software implementation details.

This work proposes an extensible computational mechanics environment. The software architecture of this environment is independent of the type of mechanics problem under consideration. The application can be easily extended through the configuration of attributes, which constitute the semantics of the application. An embedded interpreted configuration language is adopted for this purpose. Attribute specification is performed independently of the geometric model description and its discretization. The system provides a high level interface to link the simulation attributes to the geometric and discretization entities. The model information and associated attributes are organized in object oriented classes. The classes are defined by way of the configuration language. New classes and methods can be created at run time, allowing the application to be reconfigured without recompilation. An important aspect of the proposed system is that

it can be implemented not only on new programs, but also on existing ones. It allows any application to become customized.

The work begins with a discussion about the main aspects of the numerical simulation on computational mechanics. An existing system called FRANC3D, used for the numerical simulation of 3-D crack propagation, is presented as an illustration and a motivation for the proposed system. A general discussion about code reusability and customization of applications is presented. Next, the architecture of the proposed system and a brief description of the classes are given. Finally, two different environments, where the proposed system has been implemented, are shown together with some application problems to illustrate the performance of the proposed system.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Em simulações numéricas de problemas complexos de mecânica são, geralmente, utilizados *métodos numéricos*, principalmente o método dos elementos finitos [Bathe 1982]. Esses métodos requerem um modelo numérico que consiste, basicamente, de atributos (informação adicional à descrição geométrica, necessária para a definição do problema em termos físicos) associados a uma discretização do domínio do objeto (descrição do objeto em termos de elementos de formas simples, como quadriláteros ou tetraedros).

A geração do modelo numérico é um dos principais passos de uma simulação numérica, e também um dos maiores, ou seja, consome muito tempo e esforço do analista. No sentido de aumentar a produtividade, deve-se procurar automatizar ao máximo essa etapa. Um sistema totalmente automático pode ser definido como sendo um sistema onde o usuário descreve o problema em um alto nível de abstração e determina um desejável nível de precisão. A partir desse instante, o sistema deve ser capaz de prosseguir com a simulação, independentemente do tipo de análise e sem outras intervenções do usuário, até convergir para uma solução. Nesse sentido, muito esforço tem sido concentrado no desenvolvimento de ferramentas apropriadas, principalmente na parte de modelagem geométrica e modelagem de sólidos (ACIS [Spatial Technology Inc 1994]). A mesma atenção não é notada, entretanto, em relação a outros aspectos da simulação, como por exemplo a especificação dos atributos [Sheppard 1988].

Até recentemente, os atributos eram especificados em termos do modelo utilizado para a análise numérica, o que não é compatível com a intenção de automação do processo, conforme discutido acima. A definição do problema (descrição geométrica e atributos)

deve ser feita independentemente da discretização do objeto. Com essa abordagem, o processo de criação do modelo numérico consiste em construir a geometria do modelo do objeto a ser analisado, associar os atributos à partes dessa geometria, discretizar o modelo e mapear os atributos das entidades geométricas para as entidades da discretização (nós e elementos).

Esta tese trata dos aspectos desse processo referentes aos atributos, ou seja, à especificação e a associação desses atributos à geometria do modelo e ao mapeamento para os nós e elementos da malha. Dentro de um ambiente configurável pelo usuário, foi desenvolvido um sistema para o gerenciamento dessas tarefas. É proposta uma arquitetura, utilizando esse sistema, que possibilita o desenvolvimento de aplicações configuráveis para mecânica computacional. Isto significa que uma aplicação em uma área de mecânica computacional pode ser configurada, por um usuário, com um alto nível de abstração, para ser usada em outras áreas, aproveitando-se todas as atividades de modelagem e visualização que são comuns a todas as áreas. A configuração dos atributos é feita de maneira simples e independente do modelador geométrico utilizado. Os aspectos referentes à definição da geometria, geração da malha e visualização dos resultados, não são tratados neste trabalho.

1.1 Motivação

Muitos sistemas existentes para simulações de mecânica computacional são específicos para uma aplicação em particular, sendo portanto difícil a sua utilização em novas aplicações, ou são muito grandes e genéricos, dificultando o uso e a manutenção.

Esses sistemas são, geralmente, bastante complexos e utilizam estruturas de dados sofisticadas, como estruturas topológicas para tratar situações *non-manifold* [Weiler 1986]. Exemplos dessas situações incluem a combinação de diferentes tipos de elementos (sólidos, vigas ou cascas) em um mesmo modelo de elementos finitos ou a representação de entidades sem volume (faces que representam a fronteira entre duas regiões com materiais diferentes). Nesses sistemas é difícil adicionar novas funcionalidades para atender diferentes tipos de simulações, pois qualquer modificação requer, normalmente, um conhecimento detalhado do código da implementação.

Por outro lado, existem vários aspectos da simulação, como modelagem geométrica, discretização e visualização, que são independentes do tipo de análise e poderiam ser utilizados em diversas aplicações.

Como exemplo, considera-se a versão atual do código do FRANC3D [Martha 1989; Potyondy-Wawrzynek-Ingraffea 1995], um programa sofisticado para simulação de propagação de trincas não-planares em sólidos, que suporta análise elástica de elementos de contorno ou análise de elementos finitos de cascas. Os tipos de atributos implementados no código são referentes a esses tipos de análise. Desse modo, para realizar uma análise térmica, por exemplo, seria necessário uma mudança no código – tarefa não trivial – para acrescentar os novos tipos de atributos necessários. Por outro lado, as capacitações requeridas por uma análise térmica referentes à geração da geometria e da discretização, por exemplo, são as mesmas utilizadas para as análises feitas atualmente pelo programa, estando portanto já implementadas.

Desse modo, pode-se dizer que utilizar um sistema para diferentes simulações consiste em definir novos tipos de atributos, específicos para cada simulação.

É desejável, portanto, que exista um mecanismo que possibilite a especificação dos atributos de maneira independente dos serviços de modelagem da aplicação, aproveitando-se, assim, os módulos responsáveis por esses serviços em vários tipos de simulação. Esse mecanismo deve ser simples e eficiente.

1.2 Uma solução

A dificuldade envolvida em desenvolver e manter códigos de programas complexos tem motivado o desenvolvimento de novos procedimentos e novas ferramentas de programação, tais como programação orientada a objetos, re-uso de *software* e aplicações configuráveis.

Aplicações complexas são melhor estruturadas se compostas de dois ou mais componentes: um componente principal, que implementa os serviços oferecidos pela aplicação, e componentes auxiliares que oferecem acesso programável aos serviços existentes [Valdés 1991]. Aplicações configuráveis implementam essa arquitetura, de modo que o usuário tem acesso a esses componentes auxiliares, isto é, aplicações configuráveis representam

a possibilidade do usuário acessar os serviços oferecidos pela aplicação, via a parte configurável.

O acesso a esses serviços pode ser feito através de linguagens específicas, que são, tipicamente, linguagens interpretadas, pequenas e simples (*little languages*) [Valdés 1991]. Essas linguagens permitem que alguns usuários, com um determinado nível de conhecimento, sejam capazes de construir seus próprios sistemas customizados. Esses usuários avançados são os responsáveis pela configuração da aplicação.

O paradigma de programação orientada a objetos permite a estruturação de classes direcionadas a um determinado domínio de aplicações. Nesse caso, as classes incorporam conceitos semânticos relacionados com o domínio de interesse e podem ser enumeradas atendendo à totalidade das aplicações específicas dentro desse domínio. O domínio de uma aplicação é definido através da especificação de seus atributos, que representam a semântica da aplicação. Os mecanismos de herança, sobrecarga de métodos e polimorfismo da programação orientada a objetos permitem que, para a extensão de uma aplicação, classes derivadas das classes existentes possam exercer as funções necessárias no novo domínio de aplicação, sem que o sistema como um todo seja alterado.

1.3 Objetivo e escopo

O objetivo desta tese é propor um ambiente para simulação numérica em mecânica computacional, no qual seja possível a configuração de atributos específicos a um determinado problema, com um alto grau de abstração, sem a necessidade de recompilação e re-ligação do programa e, principalmente, sem ser preciso conhecer detalhes sobre o código. Como os sistemas de análise são tipicamente muito grandes, a recompilação de módulos e posterior re-ligação não é uma alternativa prática. Uma das metas importantes do sistema proposto é a de poder ser implementado não só em novos programas, mas também em códigos já existentes. A arquitetura proposta envolve aspectos e conceitos de diversas áreas, como computação gráfica, linguagens de programação, interface com o usuário, mecânica computacional e análise numérica.

Esta tese particulariza o domínio das aplicações para problemas de mecânica computacional. Esse domínio, no entanto, engloba diversos subdomínios que se referem aos difer-

entes tipos de simulação, como análise linear-elástica de propagação de trincas, análise térmica, etc. Nesse contexto, uma aplicação configurável se refere a um sistema para simulação de mecânica computacional com mecanismos para a configuração dos atributos. Uma aplicação específica se refere à utilização desse sistema para um determinado tipo de análise, ou seja, a especificação dos atributos para esse caso particular.

O mecanismo para a configuração dos atributos utiliza uma única linguagem acoplada, estendida com comandos específicos para tarefas referentes à especificação de atributos e à geração do modelo numérico. Uma biblioteca de classes direcionadas para esse domínio de aplicação formam a base a partir da qual se faz a configuração da aplicação. Isto é feito através da redefinição de métodos das classes existentes ou a partir da criação de novas classes de atributos.

O sistema proposto idealiza diferentes responsáveis pelo seu desenvolvimento, com crescente nível de abstração para a realização das tarefas. Os grupos diferem entre si de acordo com seus conhecimentos de programação e tarefas que devem desempenhar dentro do sistema. Usuários e configuradores, por exemplo, não devem ser responsáveis por serviços de programação do código do sistema, já que não são especialistas em computação, mas sim nas áreas relacionadas com seu serviço.

O sistema que implementa a arquitetura proposta, chamado de ESAM (*Extensible System for Attribute Management*), consiste de um conjunto de funções responsáveis por serviços referentes à configuração da aplicação, um conjunto de classes que representam a abstração das entidades envolvidas no processo de geração dos modelos goemétrico e numérico (*core classes*), um conjunto de classes que formam a base a partir da qual classes específicas de atributos são construídas (*attribute classes*), e um conjunto de funções que fazem a interface entre as *core classes* e o modelador.

O acesso às classes existentes é feito através de uma ferramenta que permite ao usuário percorrer toda a estruturação de classes do sistema, com suas respectivas variáveis e métodos. Essa ferramenta, denominada *Browser*, permite ainda a construção de novas classes, no caso de classes de atributos, ou a redefinição de métodos, no caso de classes de geometria.

A Fig 1.1 ilustra uma aplicação que utiliza o ESAM.

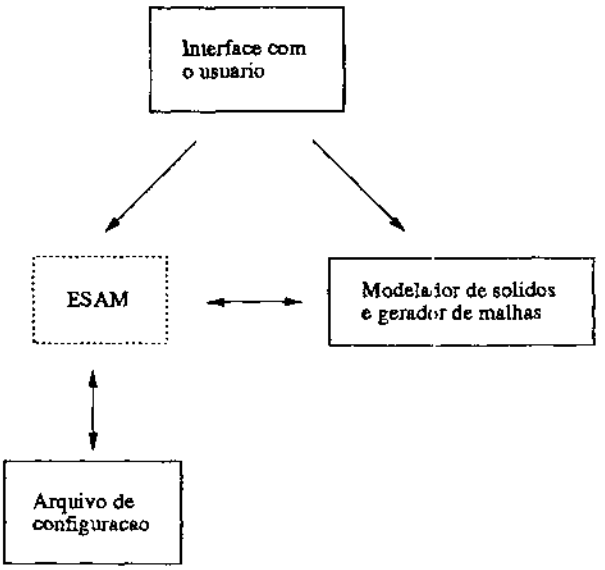

Figura 1.1: Aplicação que utiliza o ESAM.

1.4 Definição de termos

Esta seção define alguns termos utilizados ao longo desta tese. O trabalho apresentado se insere no contexto de diferentes áreas, incluindo mecânica computacional, modelagem de sólidos, análise numérica, engenharia de *software* e computação gráfica. Cada um desses campos possui um vocabulário próprio e, às vezes, palavras similares possuem significados diferentes, dependendo da área. Desse modo, os termos importantes são definidos a seguir.

Uma *simulação numérica* é uma tentativa de se reproduzir um evento físico, através do computador. Apesar de existirem diversas formas de simulação, nesta tese o interesse é em relação a simulação numérica. Algumas vezes o termo *numérico* pode ser suprimido e *simulação* vai ser usado para indicar simulação numérica.

Como parte do processo de simulação, é necessário formular um conjunto de equações que descreve o evento físico (equações de movimento no caso de mecânica dos sólidos). O termo *análise numérica* se refere somente ao processo de formulação e resolução dessas equações. No presente contexto, para a análise numérica, o domínio do modelo tem que ser discretizado em subdomínios de forma pré-definida, denominados elementos finitos ou de contorno. Durante uma simulação, várias análises podem ser necessárias.

A discretização do domínio é denominada de *malha* e consiste de uma coleção de elementos, cada um com um determinado número de nós.

O termo simulação numérica deve ser entendido no sentido amplo, pois envolve além da análise numérica, os aspectos de preparação do modelo e de visualização e interpretação dos resultados. Exemplos de simulações incluem, entre outros, distribuição de tensões em meios elásticos, difusão térmica em um domínio arbitrário e propagação de ondas sísmicas na crosta terrestre. Esses tipos de problemas são referidos ao longo desta tese como *simulações numéricas em mecânica computacional.*

Atributos de simulação representam as informações adicionais à descrição geométrica, necessárias para a definição completa de um problema.

Um *modelo geométrico* é uma representação real ou hipotética do objeto físico em uma forma que pode ser armazenada e manipulada por um programa de computador. O modelo geométrico, nesta tese, tem uma definição mais ampla e descreve todos os dados necessários para a definição completa do problema. Isso inclui informação topológica, geométrica e os atributos de simulação, tais como cargas, material, etc.

Um *modelo numérico* consiste da informação da malha e dos atributos de simulação mapeados para essa malha.

Modelagem geométrica envolve o uso de computador para ajudar na criação, manipulação, manutenção e análise de representações da forma geométrica de objetos bi e tridimensionais.

O termo *modelador* se refere a um módulo responsável pelos aspectos da simulação que não dizem respeito à especificação dos atributos, ou seja, a descrição da geometria, a geração de malha e a visualização dos r esultados. Desse modo, o termo *modelagem* se refere a essas atividades.

Linguagens de extensão são utilizadas para estender ou customizar uma aplicação para fins particulares. Em geral, são linguagens simples, pequenas e específicas para realizar uma determinada tarefa.

Linguagens acopladas são linguagens de extensão que não possuem o conceito de programa principal, ou seja, dependem de um outro programa, denominado *programa hos-*

pedeiro.

Linguagem de configuração se refere a uma linguagem acoplada, utilizada para a configuração dos atributos, estendida com comandos específicos para esse fim.

Outros termos, que possam ser duvidosos ou ambíguos, serão definidos ao longo do texto, conforme sejam introduzidos.

1.5 Organização da tese

O restante desta tese está organizada em 6 capítulos. Os capítulos estão escritos de modo a poderem ser entendidos independentemente dos demais. Desse modo, quando se faz necessário, algumas definições são repetidas.

O capítulo 2 define o domínio de interesse da tese, isto é, apresenta o Contexto da tese. São discutidos os aspectos envolvidos em uma simulação de mecânica computacional, principalmente os aspectos de preparação do modelo numérico (modelagem geométrica, especificação de atributos, geração de malha e mapeamento de atributos da geometria para a malha). Apresentam-se as principais características de uma simulação baseada em geometria, onde a topologia é utilizada como fator de organização dos dados. Procura-se enfatizar a complexidade das tecnologias envolvidas nesse tipo de problema e dos sistemas utilizados, de modo a justificar as vantagens em se ter um mecanismo independente do tipo de análise para a especificação dos atributos.

A seguir, no Capítulo 3, apresenta-se uma discussão, em termos gerais, sobre diversos tópicos referentes aos procedimentos e ferramentas que possibilitam um melhor aproveitamento dos serviços oferecidos por uma aplicação, tais como re-uso de *software*, configuração de aplicações e programação orientada a objetos. Procura-se mostrar que aplicações complexas são melhor estruturadas se decompostas em duas partes, uma referente aos serviços oferecidos pela aplicação e outra que permite o acesso programável a esses serviços. Desse modo, o desenvolvimento de sistemas extensíveis e configuráveis representa uma maneira eficiente de se permitir o acesso, por parte do usuário, à tecnologia implementada em uma aplicação. Além disso, mostra-se que uma aplicação pode ser utilizada para diferentes tipos de problemas, de acordo com o tipo de seus atributos.

Desse modo evidencia-se que a definição desses atributos deve ser feita pelo usuário, independentemente da tecnologia da aplicação.

O Capítulo 4 representa a proposta principal desta tese. Esse capítulo utiliza os conceitos apresentados nos Capítulos 2 e 3 para apresentar uma arquitetura de um ambiente extensível para simulações de mecânica computacional baseada em geometria. Essa arquitetura permite uma clara separação entre a parte configurável e a parte dos serviços de modelagem da aplicação. Essas partes, no entanto, precisam se comunicar. Um sistema foi implementado para permitir essa comunicação e, ao mesmo tempo, manter a independência das duas partes. Discute-se as características do ambiente e os módulos envolvidos no sistema proposto. Mostra-se os níveis de abstração envolvidos no desenvolvimento de uma aplicação com a arquitetura proposta.

O Capítulo 5 descreve os aspectos referentes a implementação do sistema. São apresentados os serviços oferecidos e os métodos das classes que formam a biblioteca básica do sistema. É descrito, ainda, uma ferramenta para acessar e manipular essas classes (*browser*).

O Capítulo 6 apresenta duas aplicações desenvolvidas com a arquitetura proposta. Cada aplicação foi implementada em um ambiente diferente, com versões diferentes do sistema proposto. São discutidas as principais diferenças observadas nas duas implementações. É apresentada ainda, uma nova aplicação que está sendo desenvolvida utilizando-se o sistema proposto. São mostrados dois exemplos para ilustrar pontos do sistema. Um exemplo corresponde à implementação da funcionalidade para se aplicar um atributo em uma entidade geométrica. O outro exemplo apresenta uma das aplicações que utilizam o sistema sendo configurada para dois tipos diferentes de análise.

Finalmente, no Capítulo 7 são apresentadas as conclusões e sugestões para os futuros trabalhos.

Capítulo 2

Simulação de mecânica computacional baseada em geometria

Análise estrutural pode ser definida como o exame de uma estrutura para revelar o seu comportamento físico (Fig. 2.1). Em muitos casos se deseja prever o comportamento da estrutura proposta para poder projetá-la de maneira segura e eficiente.

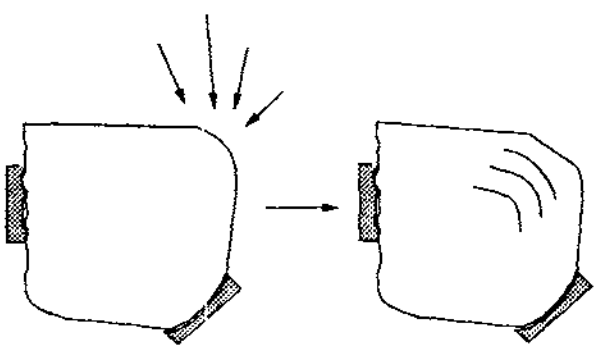

Figura 2.1: Deformação elástica de uma estrutura.

Uma simulação numérica, de acordo com a definição feita no Capítulo 1, é uma forma de se realizar uma análise estrutural com o uso de um computador. Uma simulação envolve, além da análise numérica (formulação e solução numérica das equações físicas que governam o problema), os aspectos de preparação do modelo e de visualização e interpretação dos resultados. Exemplos de análises estruturais incluem, entre outros, distribuição de tensões em meios elásticos, difusão térmica em um domínio arbitrário e propagação de ondas sísmicas na crosta terrestre.

Em simulações de problemas complexos são utilizados, usualmente, métodos numéricos, principalmente o método dos elementos finitos [Bathe 1982]. Esses métodos requisitam um modelo numérico, que consiste, basicamente, de atributos associados a uma discretização do domínio do objeto.

A geração do modelo numérico é uma das principais etapas de uma simulação e consome, normalmente, bastante tempo e um grande esforço do usuário. Com o intuito de facilitar essa tarefa e aumentar a produtividade, deve-se utilizar ferramentas que permitam a automação de aspectos desse processo. Para isso, é desejável que a definição do problema (descrição geométrica e atributos) seja feita independentemente da discretização do modelo, pois essa discretização será feita automaticamente, a partir da descrição geométrica. Com essa abordagem, o processo de geração do modelo numérico consiste em construir o modelo geométrico da estrutura a ser analisada, associar os atributos às partes deste modelo, discretizar este modelo e mapear os atributos das entidades geométricas para as entidades da discretização (nós e elementos).

Programas de simulação de mecânica computacional apresentam significativas demandas em relação ao armazenamento e manipulação de dados. Essas demandas se agravam em simulações interativas e incrementais de problemas complexos, como simulação de propagação de trincas. Um meio eficiente de organizar esses dados é através de uma estrutura de dados topológica, que consiste da representação do contorno do modelo em termos da descrição de suas entidades topológicas e das relações de adjacências entre elas. Nesse contexto, a estrutura de dados topológica serve como um fator organizador da representação do objeto. A geometria é um atributo das entidades topológicas.

Simulações complexas impõem algumas exigências em relação à estrutura de dados, que deve poder representar situações *non-manifold*, isto é, situações onde o contorno do modelo não corresponde a uma única variedade (sub-espaço de mesma dimensão) no espaço de modelagem. Essas situações surgem, por exemplo, quando se combina diferentes tipos de elementos (sólidos, vigas ou cascas) em um mesmo modelo de elementos finitos ou se deseja representar entidades sem volume, como por exemplo, faces que representam a fronteira entre duas regiões com diferentes materiais. Nesses casos, deve-se utilizar uma estrutura de dados *non-manifold*, como por exemplo, a *radial-edge*, introduzida em [Weiler 1986].

Este capítulo apresenta o contexto de modelagem para o desenvolvimento desta tese. São descritas as principais características de uma simulação de mecânica computacional que é baseada em geometria, isto é, onde a geometria e a topologia são os agentes centralizadores das informações da simulação. Procura-se salientar a complexidade da tecnologia envolvida nesses tipos de aplicações. O objetivo principal é evidenciar que várias atividades são comuns a diversos tipos de simulação nesta área, m as que a re-utilização dos recursos comuns fica comprometida pela dependência que os atributos de simulação, usualmente, têm em relação à representação dos dados da modelagem.

2.1 Atividades envolvidas em uma simulação

Tradicionalmente, as simulações de mecânica computacional envolvem três fases. Em um pré-processamento, é gerado o modelo numérico, que representa os dados necessários para a análise numérica. A análise gera resultados que são visualizados e interpretados em um pós-processamento. Esse processo envolve as atividades de modelagem, solução e interpretação [Forde 1989].

2.1.1 Modelagem

Um modelo é a representação de algumas características de uma entidade concreta ou abstrata. O objetivo do modelo é permitir a visualização e o entendimento do comportamento ou da estrutura dessa entidade.

A atividade de modelagem, em uma simulação numérica, está presente nos diversos níveis de abstração do processo (físico, analítico e numérico) (Fig 2.2). O nível físico corresponde à estrutura existente na realidade e a sua complexa descrição. Em um nível de abstração abaixo, é feita uma idealização simplificada do problema, em termos de descrições analíticas. No nível de abstração mais baixo, são utilizadas técnicas numéricas para aproximar o modelo analítico. Essa abordagem sobre modelagem é consistente com uma representação clássica de Requicha (1980), que divide a modelagem em três níveis: objetos físicos, objetos matemáticos e representações.

Partindo do nível físico, o analista extrai as propriedades espaciais pertinentes para

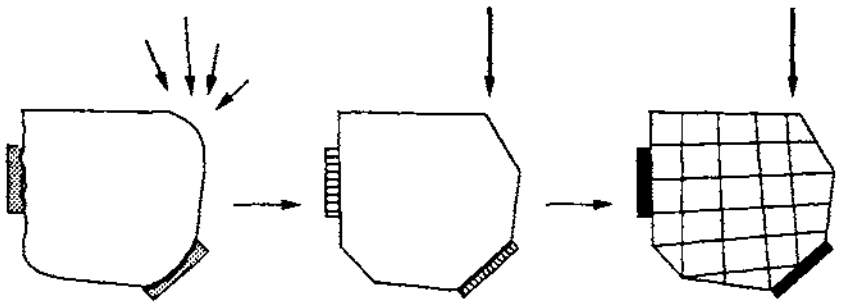

Figura 2.2: Diferentes níveis de idealização em um problema de mecânica computacional.

a geração do modelo analítico ou matemático, que consiste de uma aproximação do problema real. Atribui-se ao modelo matemático uma representação, que consiste em um conjunto de estruturas de dados e um conjunto de operadores que manipulam essas estruturas. Descrições espaciais, geralmente, incluem geometria (coordenadas de um ponto ou equações de superfícies, por exemplo) e topologia (relações de adjacências entre as entidades geométricas).

No caso de uma simulação numérica, ainda é definido o modelo numérico, que é o nível mais baixo de abstração (Fig 2.2). As informações requisitadas por programas de análise de elementos finitos são, tipicamente, as informações da discretização do modelo (malha) e um conjunto de parâmetros de controle. A malha contém as informações básicas do problema (nós, elementos, conectividade, condições de contorno, materiais, etc.) e os parâmetros de controle contêm as informações que indicam ao programa de análise como resolver o problema.

2.1.2 Solução

A atividade de solução pode ser vista como uma forma de se obter informações sobre o problema em um determinado nível de abstração. No nível físico, a natureza é responsável pela solução, utilizando seus próprios meios para revelar o comportamento físico da estrutura. No nível analítico, a solução envolve manipulações de formulações matemáticas para produzir respostas analíticas. No nível numérico, são utilizadas técnicas de aproximação para computar a resposta numérica, isto é, a resposta analítica computada através de algum procedimento numérico.

Soluções analíticas não são possíveis para problemas complicados, já que a complexi-

dade matemática cresce demasiadamente para problemas com formas geométricas complicadas e condições de contorno não uniformes. Por outro lado, soluções numéricas são facilmente automatizadas. O método dos elementos finitos é um exemplo de como simplificar um problema dividindo-o em vários pequenos problemas, que podem ser resolvidos por métodos genéricos. A tarefa do projetista de um programa de elementos finitos é desenvolver soluções eficientes e práticas para cada um desses pequenos problemas e oferecer meios para, a partir dessas soluções individuais, se obter a solução completa do problema original.

2.1.3 Interpretação

A atividade de interpretação corresponde à tradução da resposta numérica em informações significativas para o usuário. Os números fornecidos pelo programa de análise correspondem a valores discretos no nível numérico (deslocamentos, tensões, etc.). Esses números, que representam os resultados numéricos, devem ser validados em relação ao modelo analítico utilizado (resposta analítica). Deve-se ainda confirmar se a resposta analítica é consistente com o comportamento físico observado, ou desejado, da estrutura. Esses resultados são interpretados, muitas vezes, através de programas de pós-processamento, que mostram a resposta numérica em um nível alto de abstração, facilitando a tarefa de interpretação.

2.2 Geração do modelo numérico

A geração do modelo numérico é uma atividade fundamental, quando soluções numéricas são aplicadas em uma simulação. O objetivo é gerar as informações requisitadas pelos programas responsáveis por essas soluções numéricas. Geralmente, essas informações são fornecidas através de arquivos de entrada específicos dos programas de análise.

Os principais passos na geração de um modelo numérico consistem da geração da discretização (malha) e da associação dos atributos, referentes ao problema, aos nós e elementos da malha. A malha consiste da coleção de elementos com formas simples, cada qual com um determinado número de nós. As propriedades do modelo necessárias à análise (geometria, material, cargas, etc.) são fornecidas ao programa de análise através

de infomações sobre os nós e elementos, sobre a conectividade da malha, e sobre as funções de interpolação associadas com cada elemento.

Até recentemente, a especificação original das características da simulação (os atributos requisitados para uma determinada análise e a descrição da geometria do problema) era feita diretamente sobre o modelo utilizado pela análise (modelo numérico), ou seja, em termos das entidades da malha (nós e elementos) [Shephard-Tonias-Weidner 1982].

Com essa estratégia, no caso de uma análise por elementos finitos, a própria malha é utilizada como uma apropriada idealização para a geometria do objeto a ser analisado. Após essa etapa, as informações de cargas, material e condições de contorno, necessárias para completar a definição do modelo de análise, são definidas diretamente sobre os nós e elementos da malha. Esse procedimento não se mostra adequado para simulações complexas, tais como:

- Problemas onde se deseja incorporar conceitos físicos mais complexos e modelar processos mais complexos:
  - Propagação de trincas.
  - Conformação mecânica.
- Sistemas acoplados:
  - Diferentes métodos numéricos (elementos finitos e elementos de contorno).
  - Diferentes processos físicos (interação fluido-estrutura).
  - Diferentes idealizações (viga-casca-sólidos).
- Análise adaptativa
  - Problemas onde a discretização é automaticamente redefinida em função de uma avaliação de erro.
- Integração da simulação, como um de seus componentes, dentro de um ambiente mais amplo:
  - CAD/CAM (*Computer Aided Design and Computer Aided Manufacturing*).
  - Controle numérico.
  - Robótica.

Nesses casos, a definição do problema (descrição geométrica e atributos) deve ser feita independente da discretização utilizada na análise. Em geral, essa definição é feita através de uma idealização da geometria do objeto, chamada de modelo geométrico. Assim, quando a discretização tiver que ser mudada, pode-se aproveitar a definição feita no nível do modelo geométrico. Esta estratégia é fundamental no sentido de se conseguir a automação das ferramentas utilizadas no processo de modelagem. Este procedimento é referido como modelagem baseada em geometria.

2.3 Modelagem baseada em geometria

Neste contexto, a noção de geometria vai além da descrição matemática de curvas e superfícies que formam o objeto. Geometria pode representar as propriedades geométricas dos componentes do objeto e as características que afetam essas propriedades, a topologia dos componentes, i.e., informação sobre a conectividade entre os componentes, dados específicos de uma aplicação e outras propriedades associadas aos componentes (atributos de simulação). O processo de construção do modelo numérico consiste em:

- Construir o modelo geométrico da estrutura a ser analisada.
- Associar os atributos de simulação às partes deste modelo.
- Discretizar o modelo e mapear os atributos das entidades geométricas para as entidades da discretização (nós e elementos).

Esse procedimento oferece diversas vantagens, entre as quais um melhor suporte para a automação do processo de modelagem, incluindo geração automática de malha e análise adaptativa, e a simplificação da geração do modelo para a simulação, uma vez que a discretização herda, automaticamente, os atributos do modelo geométrico[1] [Shephard-Finnigan 1988; Finnigan et al. 1989].

A construção do modelo geométrico consiste na combinação de informação geométrica e topológica (informação de adjacência entre as entidades geométricas), no sentido de se

[1] *Um exemplo seria o caso de se ter uma distribuição de forças em uma curva do modelo geométrico. Nós de uma malha de elementos finitos (discretização) que estão sobre esta curva herdam automaticamente parcelas destas forças.*

obter uma completa definição do problema.

A topologia é o fator organizador da representação do modelo, à qual todos os outros dados são relacionados. Os atributos de simulação são associados diretamente às entidades topológicas. A própria descrição geométrica é um atributo da topologia.

A definição do modelo geométrico e a geração da malha são independentes do tipo de análise e do programa utilizado. Entretanto, a especificação dos atributos de simulação é menos geral, pois depende da classe de problemas que vão ser analisados.

2.3.1 Geometria

A geometria do modelo, no sentido mais amplo definido anteriormente, desempenha um papel fundamental em muitas aplicações computacionais. Modeladores geométricos são ferramentas que permitem criar e manipular representações dessa geometria. Um modelo geométrico é formado por componentes com geometria bem definida (por exemplo, através de suas coordenadas) e por relações de conectividade entre estes componentes (topologia). Dentre as aplicações que podem utilizar os serviços de um modelador geométrico, estão as simulações de mecânica computacional.

Diversas formas de modelagem geométrica podem ser utilizadas, dependendo do que se deseja representar, da quantidade e do tipo de informação diretamente disponível e de quais informações podem ou não ser obtidas.

Dentre os diversos tipos, existem três classes principais de modelagem geométrica que evoluíram historicamente [Requicha 1982, Hoffmann 1989]. Cada classe contém diferentes níveis de informação geométrica. Um modelo de reticulados (*wireframe*) contém informações sobre as arestas do objeto. Um modelo de superfícies contém informações sobre as arestas e superfícies que compõem o objeto. Um modelo de sólidos contém, além da descrição do contorno, informações explícitas sobre relações entre as entidades que compõem o objeto e sobre volumes internos e externos.

Modelos de reticulados (*wireframe*) foram uma das primeiras formas de modelagem geométrica e representam os objetos por arestas e pontos de sua superfície. Consistem de dados geométricos em forma de localização de pontos e curvas. Não representam

uma completa descrição do objeto, uma vez que não possuem informação do contorno. Uma malha de elementos finitos composta de elementos de barras pode ser representada diretamente por um modelo de reticulados.

Modelos de superfícies estendem a representação de reticulados incluindo a descrição matemática das formas das superfícies limitadas pelas curvas do objeto. Com essa representação, pode-se obter imagens gráficas do objeto com certo grau de realismo. Embora esse modelo contenha a descrição individual de cada superfície, também não representa completamente o objeto, pois apresenta informações incompletas para representar o seu interior. Desse modo não é possível, por exemplo, concluir se um ponto no espaço está localizado no interior ou no exterior do objeto. Oferece poucas possibilidades de verificação de integridade.

Diversos sistemas de CAD (*Computer Aided Design*) utilizam uma dessas duas primeiras formas de modelagem. Esses sistemas, no entanto, não são capazes de realizar diversas tarefas, tais como calcular automaticamente as propriedades de massa (volume, etc.) dos objetos. A principal funcionalidade desses sistemas é o desenho.

Modelagem de sólidos compreende um conjunto de teorias, técnicas e sistemas com o intuito de permitir a completa representação de um objeto tridimensional. Isto permite, pelo menos em princípio, que qualquer propriedade geométrica, de qualquer sólido representado, possa ser calculada automaticamente.

Existem três tipos principais de representação de um sólido: modelos de decomposição, modelos construtivos e modelos de fronteira [Requicha-Rossignac 1992]. Modelos de decomposição descrevem o sólido através de operações de colagem de elementos mais simples (primitivas). Modelos construtivos descrevem o sólido através de operações booleanas realizadas em sólidos de forma mais simples ou em semi-espaços. O modelo mais conhecido é o CSG (*Constructive Solid Geometry*) [Requicha 1980]. Os modelos de fronteira descrevem o sólido através da representação das superfícies que o delimitam.

Modelos de sólidos contêm informações suficientes sobre a geometria para permitir a análise de propriedades volumétricas. Podem conter, também, explicitamente, informações sobre as adjacências das entidades que compõem o objeto. Desse modo é possível determinar o interior de uma região do modelo. Sem saber se um ponto é

interior ou não, é impossível determinar, por exemplo, as propriedades de massa de um sólido. Modelos de sólidos não são ambíguos e, portanto, são capazes de suportar algoritmos automáticos para aplicações geométricas. São utilizados em diversas áreas, como computação gráfica, geração de malhas de elementos finitos, controle numérico, robótica, etc.

2.3.2 Topologia

Uma das formas de se obter um modelo de sólidos é através da representação explícita do seu contorno, chamada de representação de fronteira (*B-rep*). Nessa abordagem, o modelo é descrito por uma coleção de pontos, curvas e *patches* que delimitam sua superfície, associados às informações de adjacência entre essas entidades. As informações relacionadas com a geometria das entidades são referidas simplesmente como geometria e as informações relacionadas com a conectividade das entidades são referidas como topologia.

Geometria pode ser considerada para representar, essencialmente, toda informação necessária sobre a forma geométrica de um objeto, incluindo onde ele se localiza no espaço e o posicionamento geométrico de seus vários componentes.

Topologia é uma abstração, um subconjunto coerente de informaçiveis da geometria de um objeto. Mais formalmente, é um conjunto de propriedades invariantes em relação a um conjunto específico de transformações geométricas. Esta invariância implica, por definição, que propriedades representadas pela topologia não incluem o conjunto de informações que realmente mudam quando submetidas a essas transformaçôes [Weiler 1986].

Essencialmente, a topologia de um objeto é a informação sobre relação, conectividade, proximidade e qualquer outra característica da geometria que se enquadre na definição acima. Topologia representa informação geométrica incompleta. Desse modo, para se obter uma completa e única representação do objeto, deve-se combinar geometria e topologia.

Uma das maneiras de se implementar a representação da fronteira do objeto é através de uma estrutura de dados topológica, que armazena a descrição das entidades topológicas

que formam o objeto e as relações de adjacências entre elas [Baumgart 1972, Mäntylä 1988]. Essa informação topológica fica explicitamente disponível nesse tipo de representação.

Deste modo, um modelo de sólido pode ser visto como um conjunto de entidades topológicas, cada qual com um conjunto de atributos associados, dentre os quais a geometria da entidade. Assim, a estrutura de dados topológica serve como um fator organizador para a representação dos dados do objeto e, portanto, para os algoritmos que operam nas estruturas de dados.

Existem diversas vantagens em se utilizar topologia como um fator organizador para mecânica de sólidos computacional:

- Topologia, ao contrário de geometria, pode ser armazenada exatamente, sem ambiguidades ou aproximações. Devido à natureza aproximada das representações geométricas, é possível ocorrer problemas de precisão, como existência de pequenos "vazios" entre faces que deveriam ser adjacentes. Determinar relações de adjacências baseado somente em informações geométricas pode causar erros.

- Qualquer configuração topológica pode representar um infinito número de configurações geométricas. Muitos problemas podem ser resolvidos de forma genérica em um espaço topológico e então mapeado para uma instância geométrica específica. Uma das vantagens é que, uma vez definido o domínio que se deseja representar e a correspondente representação topológica selecionada, a implementação permanece estável. A representação da geometria pode ficar em módulos separados. Com isso, pode-se lidar com múltiplas representações geométricas ao mesmo tempo, e novas formas de representações podem ser introduzidas mais facilmente. O impacto dessas mudanças é minimizado, restrito a pequenas partes da implementação.

- No caso de simulação de propagação de trincas, por exemplo, a geometria do objeto muda a cada incremento da trinca. A topologia, no entanto, muda de forma bem menos frequente. Em geral, mudanças topológicas implicam em grandes mudanças geométricas.

- A utilização de estruturas de dados topológicas permite que consultas sobre a

conectividade das entidades sejam feitas de forma eficiente. Além disso, modificações no modelo podem ser feitas localmente, sem necessidade de reorganizar globalmente o resto dos dados.

- A estrutura de dados topológica permite manter a consistência do modelo durante todas as fases da modelagem.
- A utilização de operadores esconde a complexidade da manipulação dos dados.

As estruturas de dados variam de acordo com o domínio que se deseja representar. Existem algumas situações que não podem ser representadas pelas estruturas de dados normalmente utilizadas nos modelacores de sólidos, como por exemplo, casos em que se tem várias faces adjacentes a uma aresta. Essas situações são referidas como situações *non-manifold* e serão definidas na próxima seção.

2.3.3 Modelagem non-manifold

Mecânica de sólidos computacional 3D requer uma forma especial de modelagem de sólidos. Muitos dos aspectos da representação dos objetos são similares para modelagem de sólidos e mecânica computacional. Contudo, há pelo menos uma importante diferença. Em mecânica computacional, frequentemente, é desejável representar explicitamente e manipular componentes ou domínios sem volume. Isso não é permitido em muitos modeladores de sólidos clássicos, onde essas situações representam objetos inválidos [Requicha 1980]. Existem duas razões porque esses itens são de interesse. Uma é para representar idealizações estruturais de objetos reais (cascas e vigas por exemplo) e outra para representar características internas, como faces que representam a fronteira entre duas regiões com diferentes materiais. Estas idealizações não são, em geral, representadas por estruturas de dados *two-manifold* e requerem uma estrutura capaz de representar situações *non-manifold.*

Uma situação *non-manifold* é uma situação que não é *two-manifold.* Em uma representação *two-manifold* todo ponto em uma superfície contém uma vizinhança que é homeomorfa a um disco bidimensional, isto é, embora a superfície exista no espaço tridimensional, é topologicamente plana, em uma área suficientemente pequena, perto do ponto [Hoffmann 1990].

A modelagem *non-manifold* é uma forma de modelagem que remove as restrições associadas a uma modelagem *two-manifold* possibilitando a existência das três formas de modelagem mencionadas anteriormente em um único ambiente e estendendo o domínio de representação além das formas anteriores.

Operações booleanas sobre objetos *two-manifold*, podem apresentar como resultado situações *non-manifold*, ou seja, situações onde existam pontos em superfícies que não possuem uma vizinhança que seja um simples disco bidimensional (Fig 2.3) [Wawrzynek 1991].

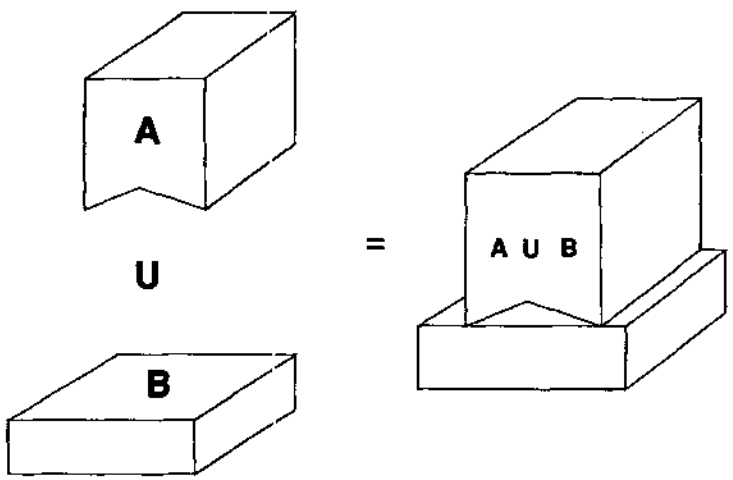

Figura 2.3: Operação booleana de sólidos *two-manifold* que resulta em uma situação *non-manifold*.

Exemplos desse tipo de situação incluem um cone tocando uma face em um simples ponto ou uma face na fronteira de duas regiões (Fig. 2.4) [Wawrzynek 1991].

Representações *non-manifold* evitam essas singularidades, representando as situações *non-manifold* diretamente, ao invés de restringir o domínio do resultado das operações. De especial interesse, representações *non-manifold* permitem um uniforme tratamento de qualquer combinação de objetos reticulados, superfícies e sólidos (Fig. 2.5).

Representações *non-manifold* permitem representar uma quantidade muito maior de objetos e, portanto, suportam uma gama maior de aplicações, mas a um custo maior do tamanho da estrutura de dados.

Uma estrutura bastante utilizada para representar situações *non-manifold* é a *radial-edge* [Weiler 1986]. Essa estrutura de dados contém a representação dos elementos topológicos

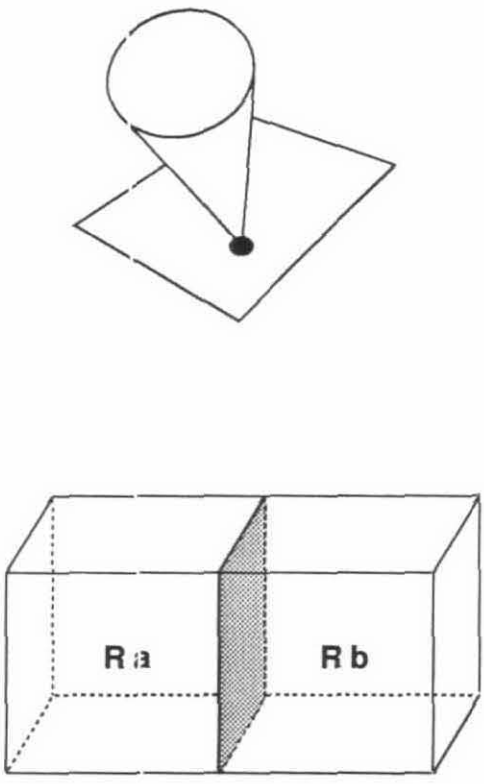

Figura 2.4: Exemplos de situações *non-manifold*.

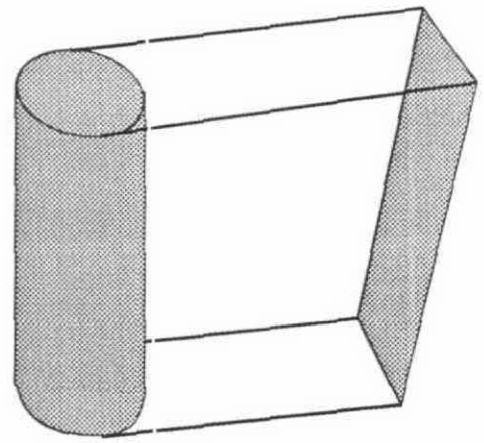

Figura 2.5: Combinação de componentes de diferentes dimensões.

comuns (vértices, arestas, faces e regiões) e ainda de elementos para a representação de cavidades, ou seja, fronteiras desconexas (ciclo e casca). O nome da estrutura se deve ao fato de ser possível a ordenação radial de faces ao redor de uma aresta (Fig. 2.6). Para armazenar as informações de adjacência do objeto, foi introduzido o conceito de uso de um elemento topológico. Uma face, por exemplo, possui dois usos, uma para cada região adjacente à face. Arestas e vértices podem ter vários usos. Através desse conceito, é bastante simplificada a travessia da estrutura de dados para obtenção das informações de adjacências entre os elementos. A estrutura *radial-edge* organiza os elementos topológicos de uma maneira hierárquica, conforme ilustrado na Fig. 2.7

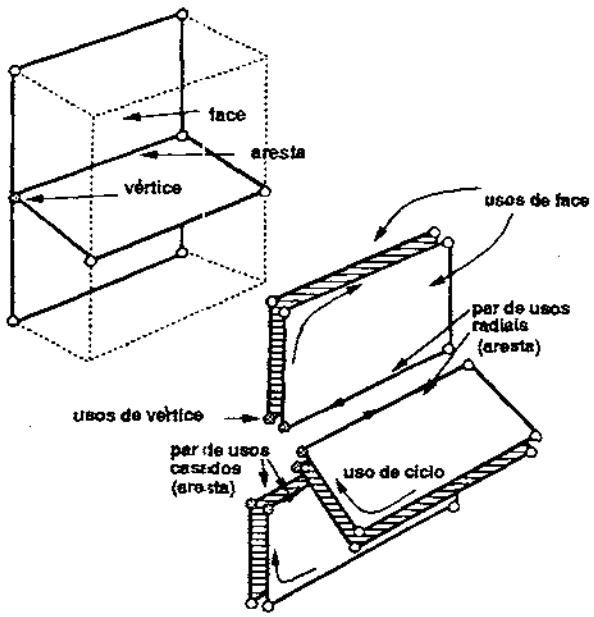

Figura 2.6: Ordenação radial das faces ao redor da aresta [Cavalcanti 1992].

2.3.4 Atributos

Topologia e geometria sozinhas não são suficientes para se realizar uma simulação. Os modelos requerem informações adicionais para descrever o problema em termos físicos e para controlar a análise numérica. Essas informações adicionais são referidas como atributos de simulação e são associadas às entidades geométricas do modelo geométrico. Atributos incluem informações geométricas e não geométricas, entre as quais pode-se ressaltar:

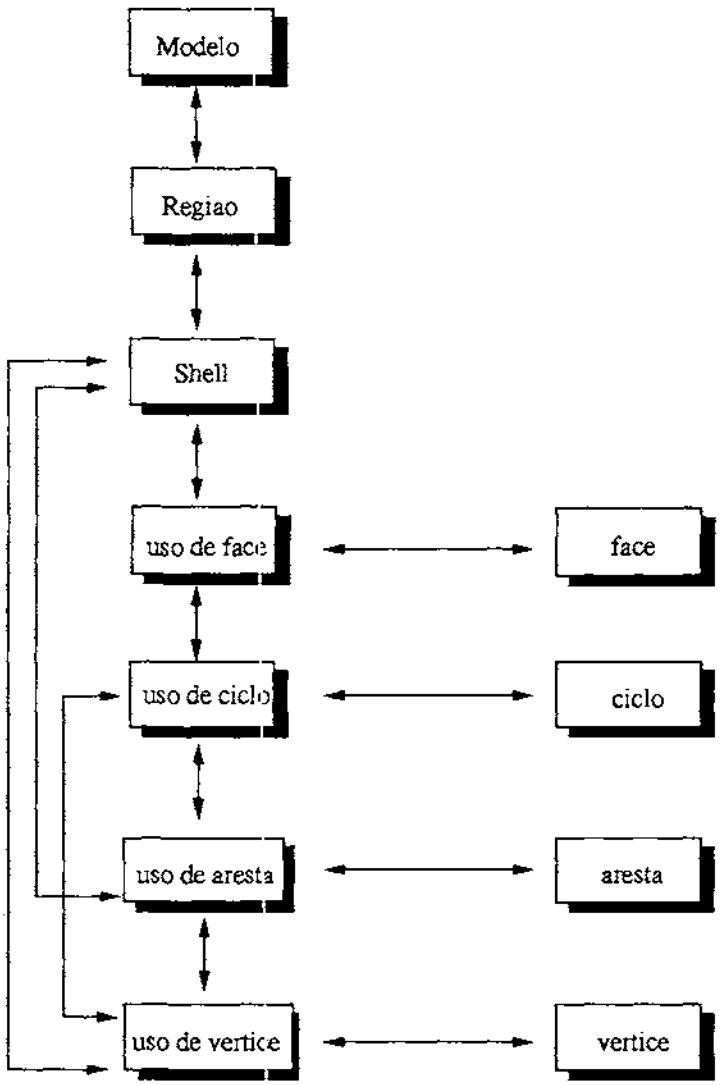

Figura 2.7: Hierarquia dos elementos na estrutura *radial-edge*.

- Atributos de análise, que consistem de informações que definem o problema em termos físicos (propriedades de material, condições de contorno, condições iniciais, etc.).

- Atributos referentes à modelagem matemática, que correspondem à diferença entre o domínio do modelo geométrico original e o modelo numérico a ser analisado.

- Atributos referentes ao controle da discretização, ou seja, parâmetros de controle para a discretização e a análise.

Atributos de análise qualificam o problema a ser analisado sobre o domínio definido pelo modelo geométrico. Em uma análise de tensões seriam as propriedades do material, os deslocamentos fixos, o carregamento aplicado em partes do objeto, etc. Esses atributos são independentes do método utilizado para a análise numérica.

Assim como os atributos de análise, os atributos referentes à modelagem matemática também são independentes do método utilizado para a análise numérica. Esses atributos são responsáveis pela especificação das diferenças entre o modelo geométrico e o modelo que vai ser discretizado ou pelas simplificações da geometria. Exemplos desses atributos incluem buracos que devam ser ignorados, detalhes geométricos que devam ser removidos ou redução da dimensão de um componente, como considerar uma parte fina do modelo pela sua superfície média ou a representação de partes dominadas por uma dimensão por linhas.

Os atributos de controle do modelo discreto são específicos do método de análise numérica que vai ser utilizado. Em modelagem de elementos finitos, um exemplo é o parâmetro de controle do grau de refinamento da malha, ou tipos de elementos que vão ser utilizados.

Conforme mencionado anteriormente, os atributos são associados às entidades topológicas, o que os tornam independentes dos detalhes do sistema de modelagem geométrica. Desse modo, é possível associar atributos com entidades geométricas, via topologia, sem a necessidade de lidar com detalhes específicos da forma da entidade.

2.4 O programa FRANC3D

Nesta seção é discutido um programa que ilustra a complexidade encontrada em simulações sofisticadas de mecânica computacional.

Simulação de propagação de trincas discretas requisitam as mesmas tarefas que uma simulação de um problema mecânica computacional comum. Contudo, a natureza incremental e interativa dessas simulações introduzem um grau adicional de complexidade. A simulação é incremental no sentido que uma simulação completa pode requerer diversas análises individuais. A geometria, a malha e possivelmente as condições de contorno podem mudar a cada análise. A simulação é interativa no sentido que é dada ao usuário a possibilidade de manipular aspectos do modelo diretamente apontando para sua imagem na tela do computador. Estritamente falando, não seria necessário uma simulação interativa. Contudo, no atual estado da arte, simulações tridimensionais não podem ser efetuadas sem uma significante participação do usuário.

Como exemplo de um sistema para realizar esse tipo de simulação, considera-se o código do FRANC3D [Martha 1989, Potyondy-Wawrzynek-Ingraffea 1995], um programa sofisticado para simulação de propagação de trincas não-planares em sólidos, que é independente do modelo de análise numérica utilizado. A versão atual suporta análise elástica de elementos de contorno ou análise de elementos finitos de cascas. O FRANC3D é escrito em linguagem C e contém mais de 200.000 linhas de código fonte.

A modelagem no FRANC3D consiste de um modelo topológico hierárquico, ao qual são associados os atributos necessários à simulação. A hierarquia representa desde a descrição do problema em um alto nível até a discretização do modelo numérico. A topologia é armazenada através da estrutura de dados *radial edge*.

A estrutura de dados *radial-edge* é uma poderosa ferramenta para a representação da topologia. No entanto, somente uma representação não é suficiente para muitas das tarefas de modelagem incremental e interativa tridimensional. Nessas situações, um forte sentido de restrição é necessário para ajudar a automação do processo de modelagem. Por exemplo, uma malha de elementos finitos deve ser restrita para se conformar com a geometria do objeto e uma trinca deve ser restrita a se manter dentro do objeto. Para forçar esses tipos de restrições, foi introduzida uma hierarquia de representações no

FRANC3D. Essa hierarquia consiste de cinco níveis de modelos topológicos, relacionados hierarquicamente (Fig. 2.8).

O nível mais alto contém a descrição topológica da representação do contorno do objeto. Esse nível, que restringe todos os demais, representa a geometria do objeto.

O próximo nível na hierarquia é a decomposição do volume. Nesse nível, regiões do modelo geométrico podem ser divididas em sub-regiões. Isso pode ser desejado por inúmeras razões, tais como definir regiões de diferentes materiais ou definir uma região de forma regular onde um algoritmo de geração de malha pode ser aplicado (*hyperpatch*).

O próximo nível é o nível de subdivisões de superfícies, onde superfícies podem ser subdivididas em sub-superfícies, que são restritas pela superfície original. Esse passo pode ser necessário por um número de razões, incluindo a especificação de condições de contorno sobre parte da superfície original ou a decomposição de faces irregulares em faces onde algoritmos de geração de malha de superfície podem ser mais facilmente aplicados.

O próximo nível da hierarquia das representações é o nível da subdivisão das arestas. Existem duas razões que explicam a existência desse nível. Primeiro, em todos os níveis superiores, a geometria das arestas é representada como curvas espaciais *splines*. No nível da subdivisão de aresta, essas curvas são aproximadas por pedaços de segmentos de retas. A segunda razão é que o número de subdivisões de arestas é usado posteriormente como parâmetro para densidade da malha de superfície ou de sólidos.

O último nível na hierarquia é o nível da malha (podendo ser malha de sólidos ou de superfície ou uma combinação das duas). Esse nível, como os demais, é restrito pelos níveis superiores, de modo que os segmentos de arestas subdivididas tornam-se lados de elementos e os pontos finais das arestas tornam-se pontos nodais da malha. Além disso, por causa da restrição dos níveis superiores, os elementos não correspondem a superfícies ou volumes diferentes da geometria.

Os atributos necessários para a simulação são associados às entidades topológicas de um particular modelo dentro dessa hierarquia. As entidades no nível da discretização podem ter atributos associados diretamente, ou herdar de entidades localizadas em níveis superiores. A organização do armazenamento dos dados é ilustrado na Fig. 2.9.

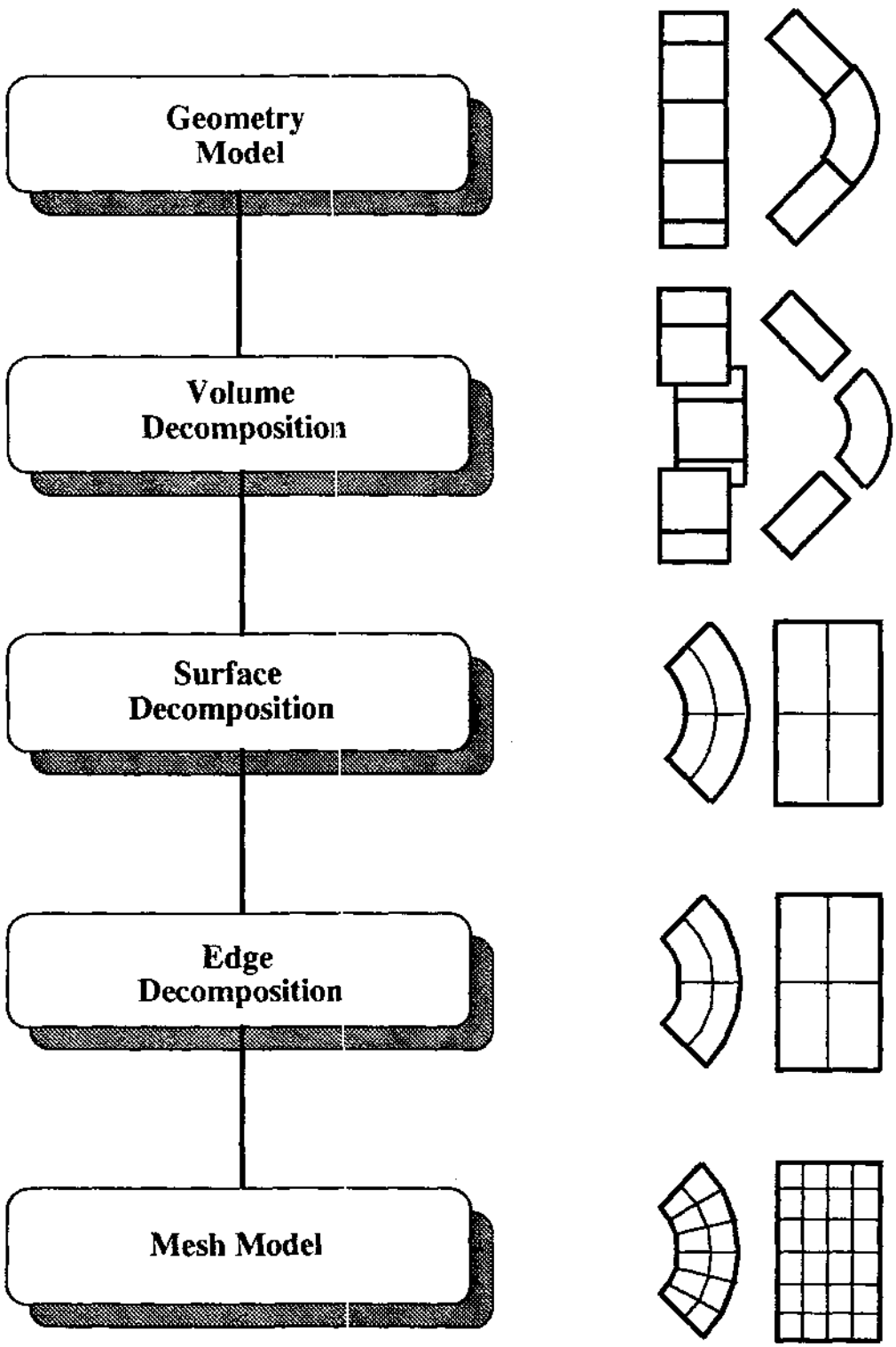

Figura 2.8: Hierarquia de representações do modelo no FRANC3D [Martha 1989].

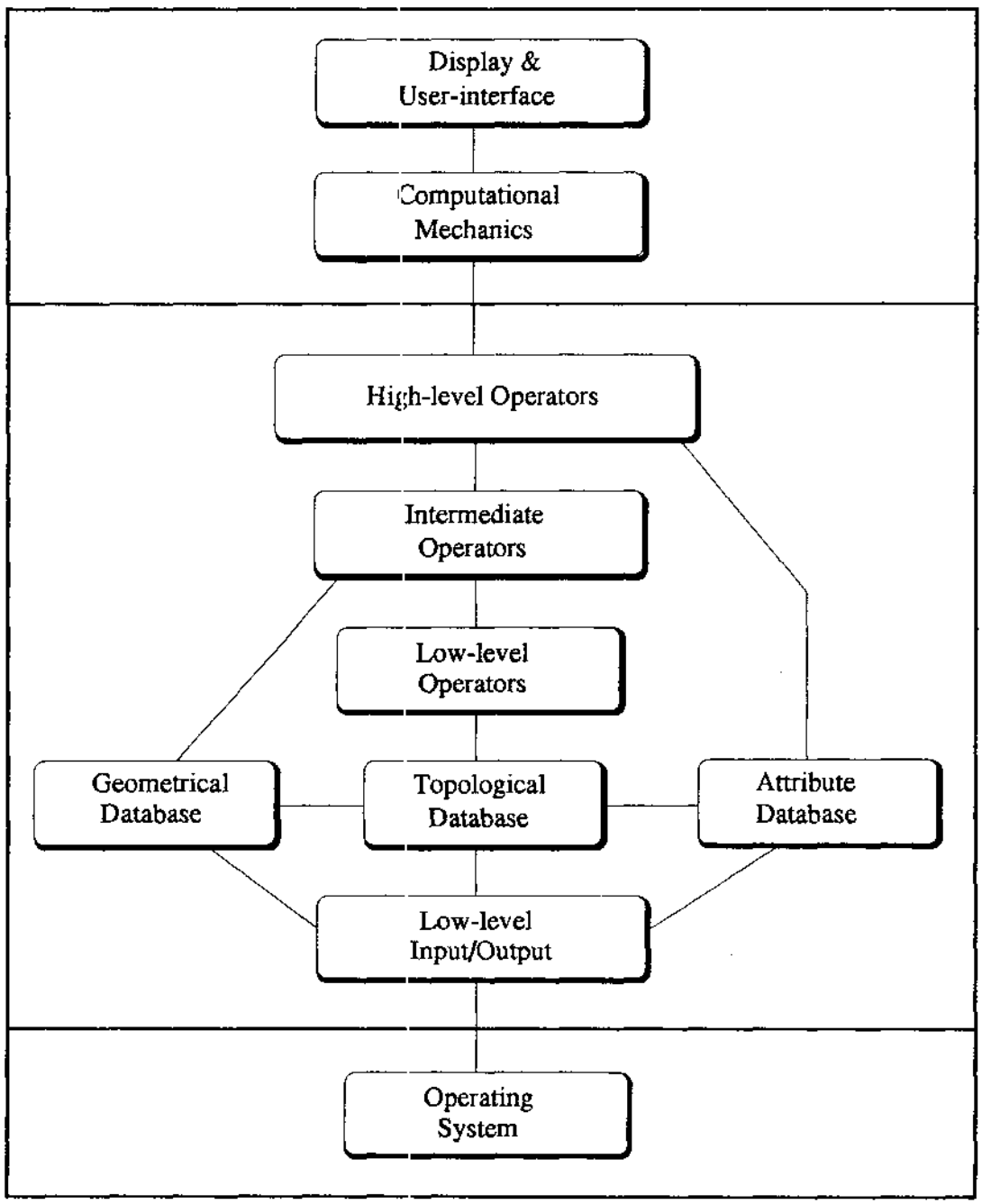

Figura 2.9: Organização do FRANC3D [Martha 1989].

A base de dados geométricos contém a descrição geométrica das faces e arestas que definem o objeto. Dois tipos de arestas são suportados: retas, definidas pelas coordenadas de seus vértices extremos, e *splines* definidas como *B-splines* cúbicas. Superfícies são de três tipos: poligonal plana, *B-spline* bi-cúbica quadrilateral e *Bezier-spline* triangular. As definições das *splines* são armazenadas na base de dados geométrica.

A base de dados topológicos contém a representação topológica do objeto na estrutura de dados *radial-edge*. Conforme mencionado acima, são mantidas as representações nos cinco níveis hierárquicos. Cada estrutura nessa base de dados contém um ponteiro para um bloco genérico de atributos. Toda informação geométrica, ou outro atributo qualquer, é acessada através desse bloco. Os blocos de atributos, no entanto, são mantidos como parte da base de dados dos atributos.

A base de dados de atributos armazena toda informação não contida nas duas bases de dados anteriores, além de informações que inter-relacionam essas bases. Essa é a parte mais dinâmica do programa, sendo constantemente modificada para atender a novas características requisitadas. É organizada como uma coleção de blocos de dados com características de objetos (no sentido de programação orientada a objetos). Esses blocos contêm dados propriamente ditos, referências para entidades geométricas, referências para entidades topológicas e referências para outros blocos de atributos.

O acesso aos dados é feito através de operadores que formam uma camada de abstração sobre esses, com o intuito de esconder detalhes de implementação e facilitar o seu uso. Operadores de baixo nível são os operadores de Euler [Mäntylä 1988] estendidos para uma modelagem *non-manifold* [Weiler 1988]. Nesse nível, operadores agem somente em uma representação topológica de cada vez. Acima desses operadores estão os operadores intermediários, que também manipulam uma representação de cada vez. Esses operadores mantêm a consistência entre a descrição geométrica e topológica do modelo. Operadores de alto nível manipulam todos os cinco níveis ao mesmo tempo e são responsáveis por manter a consistência entre eles. A correspondência entre as entidades topológicas é mantida através de blocos de atributos.

Abaixo dessa base de dados está a interface com o sistema operacional, que permite, por exemplo, modelos intermediários serem armazenados e chamados posteriormente. Isso pode ser feito em arquivos com formato binário ou texto.

A partir dessa descrição sucinta do FRANC3D, pode-se ter uma noção da complexidade que domina a tecnologia de uma aplicação sofisticada de mecânica computacional.

Esse sistema pode, potencialmente, ser utilizado em outros tipos de simulações além das que foram originalmente concebidas. Isso porque, diversos aspectos da simulação em mecânica computacional, como a modelagem geométrica, a geração da malha e a visualização dos resultados, são comuns a essas simulações.

Entretanto, a extensão do FRANC3D para outras simulações é uma tarefa não trivial. Isso porque os atributos de simulação estão intrinsecamente ligados à sua estrutura de dados. Estender a aplicação para outros tipos de simulação, implica no entendimento praticamente completo da toda a sua tecnologia. Essa é a situação em praticamente todos os sistemas de mecânica computacional que se tem conhecimento. Uma solução para esse problema constitui o principal objetivo desta tese.

Capítulo 3

Re-uso de software e configuração de aplicações

A dificuldade envolvida em desenvolver e manter códigos de programas complexos tem motivado o desenvolvimento de novos procedimentos e novas ferramentas de programação. Programação orientada a objetos, re-uso de *software* e aplicações configuráveis e extensíveis têm merecido considerável atenção ultimamente.

Devido ao alto custo de desenvolvimento e ao crescente aumento de complexidade das aplicações computacionais, é necessário o investimento em mecanismos que possibilitem o re-uso de partes de programas já desenvolvidos e testados. O uso disciplinado do paradigma de orientação a objetos facilita a tarefa de reaproveitamento de código existente.

Tem sido crescente a demanda por aplicações extensíveis e configuráveis. Estender significa adicionar módulos e funcionalidades a um programa. Idealmente, isso deve ser feito sem necessidade de recompilação e religação. Deve ser possível, por exemplo, adicionar um verificador ortográfico sem a necessidade de atualizar a versão de um editor de texto.

Uma arquitetura composta por um único módulo, formado por vários subcomponentes, não se mostra mais adequada às necessidades de diversas áreas. É comum encontrarmos programas que, apesar de implementarem tecnologias bastante sofisticadas, deixam de poder ser aplicados em áreas afins por simples inadequação de sua interface. Outros casos requerem que o reaproveitamento seja feito com acesso direto à implementação

da tecnologia da aplicação, o que, em geral, torna a tarefa bastante árdua. Aplicações complexas são melhores estruturadas se compostas de dois ou mais componentes: um componente principal, que implementa os serviços oferecidos pela aplicação, e componentes auxiliares que oferecem acesso programável aos serviços existentes [Valdés 1991]. Este acesso programável aos serviços oferecidos pode ser feito através do acoplamento de linguagens específicas. Essas linguagens são, tipicamente, linguagens interpretadas, pequenas e simples (*little languages*). Linguagens interpretadas, neste contexto, são linguagens compiladas em tempo de execução. Existem diversos exemplos dessa arquitetura em diferentes áreas, como banco de dados (Paradox), planilhas (Excel) e CAD (Autocad).

A implementação dos serviços representa a tecnologia da aplicação. Aplicações configuráveis representam a possibilidade de o usuário acessar essa tecnologia. Entretanto, obrigatoriamente, este acesso à tecnologia deve ser feito com alto grau de abstração, pois caso contrário recai-se no problema de acessar diretamente implementações complexas.

Diversas aplicações, por exemplo, trabalham com uma representação que é basicamente uma subdivisão do espaço bidimensional, sobre a qual se aplicam propriedades pertinentes [Cavalcanti et al. 1991]. Estas propriedades (atributos) caracterizam a semântica associada à aplicação, como, por exemplo, representação de mapas cartográficos, sistemas de informações geográficas, geradores de malha de elementos finitos. Desse modo, é possível reutilizar a tecnologia da subdivisão planar em diversas aplicações, sendo necessário, para isso, a definição dos atributos específicos de cada aplicação. No sentido de se obter uma grande flexibilidade, é desejável que esta definição seja feita pelo usuário, através de mecanismos de configuração.

A configuração de aplicações pode ser vista também como uma forma de re-uso, já que a implementação de uma mesma tecnologia é reaproveitada para diversos fins mais específicos. Além disso, as ferramentas utilizadas para a construção de aplicações são utilizadas em diferentes etapas e por diferentes usuários dentro do processo de desenvolvimento. Essas ferramentas também podem ser configuráveis. Assim, a estratégia de configuração de aplicações engloba, em diferentes níveis até o produto final, diversas etapas de re-uso de *software*.

Nesse contexto, surge o conceito de usuários avançados, que são mais do que simples usuários das aplicações. Esses usuários avançados, apesar de não possuírem um pro-

fundo conhecimento de programação, desejam construir, ou customizar, suas próprias aplicações. Esses usuários são os responsáveis pela configuração das aplicações.

Este capítulo resume o estado da arte no desenvolvimento de sistemas extensíveis e configuráveis através de linguagens interpretadas acopladas à linguagem que implementa a tecnologia de uma aplicação. Procura-se mostrar que este enfoque permite que a tecnologia de uma aplicação sofisticada, tal como uma simulação de mecânica computacional, possa ter seus atributos configuráveis pelo usuário em tempo de execução. É possível também o acesso programável à tecnologia da aplicação com um alto grau de abstração. Este capítulo também introduz conceitos e terminologias de programação orientada a objetos que vão ser utilizados no resto desta tese.

3.1 Uma estratégia de re-uso de software

Devido ao alto custo de desenvolvimento e ao crescente aumento de complexidade das aplicações em computador, tem sido grande a preocupação com o re-uso de programas ou de partes de programas.

Uma grande dificuldade encontrada para se re-usar código é o fato de, muitas vezes, módulos conterem informações que não são intrínsecas às suas funcionalidades, como, por exemplo, uma estrutura de dados ter informações sobre a interface da aplicação ou sobre como é feita a sua visualização, pois há diversas maneiras de se visualizar uma determinada estrutura de dados. Do mesmo modo, um módulo deve especificar seus requisitos, mas não a maneira como estes devem ser satisfeitos. As propriedades que não são fundamentais não devem estar acopladas no mesmo módulo. Para facilitar o re-uso de código, deve existir uma interface que separe os serviços oferecidos pelos diversos módulos que compõem o programa. Esta interface deve ter informações sobre os requisitos dos seus módulos, mas os módulos não devem entender sobre sua interface. Um aspecto bastante vantajoso desta abordagem é que uma interface bem especificada entre módulos define uma clara separação de tarefas [Krasner-Pope 1988; Cowan et al. 1993].

Em abrangente discussão a respeito de re-uso de programas, Krueger (1991) ressalta que a tarefa de re-usar código está intimamente ligada ao conceito de abstração. A eficácia na

reutilização de técnicas pode ser medida pela distância cognitiva para reutilizar, parcial ou integralmente, um trabalho anteriormente desenvolvido. O autor conceitua distância cognitiva, informalmente, como sendo o esforço intelectual que deve ser gasto para realizar tal tarefa. Krueger ressalta também o fato de que, tipicamente, abstração em *software* se apresenta em dois níveis: um nível de especificação, que descreve o que a abstração faz, e um nível de realização, que descreve como essa abstração é implementada.

3.2 Conceitos de programação orientada a objetos

Tarefas como inserir novos procedimentos ou mesmo novos componentes requer a modificação do *software* existente, o que é uma tarefa difícil de ser realizada com a programação convencional. Nesse sentido, é desejável se ter disponíveis códigos genéricos, onde os programas não são tão importantes pelo que fazem, mas sim pela forma como permitem os usuários realizarem as tarefas.

Neste contexto, programas podem ser vistos como um conjunto de tipos abstratos de dados e uma estratégia de controle que manipula esses dados para produzir o resultado esperado. Cada tipo abstrato de dado tem a sua própria representação interna e um apropriado conjunto de rotinas que operam sobre os seus dados. Este processo enfatiza as propriedades do sistema e esconde os detalhes de implementação, ou seja, existe uma clara separação entre o conceito e a implementação desse conceito. Objetos são um mecanismo para implementar tipos de dados abstratos [Cox-Novobilsky 1991].

A implementação dos conceitos fundamentais de programação orientada a objetos varia de linguagem para linguagem. O modelo descrito a seguir ilustra a implementação utilizada nesta tese.

De acordo com o discutido acima, *objetos* podem ser definidos como componentes compostos de dados privados e procedimentos. Objetos podem ser criados através da combinação de outros objetos. Operações são realizadas nos objetos através de *mensagens* emitidas para eles. Quando um objeto recebe uma mensagem, executa um de seus procedimentos, que são chamados de *métodos*. Esses métodos operam em dados privados do objeto, de modo que os detalhes de implementação do objeto são escondidos do resto do programa que usa esse objeto. Um conjunto de objetos que têm os mesmos dados e

métodos formam uma *classe*. Classes podem ser vistas como modelos (*templates*) usados para definir a estrutura de um objeto, enquanto um objeto é uma instância de uma classe. Esse *template* contém uma lista de variáveis que pertencem à objetos desta classe e uma lista de mensagens que são reconhecidas por estes objetos. Uma característica adicional é geralmente oferecida e consiste na possibilidade de se redefinir um método. Uma nova classe pode ser derivada de uma classe já existente, chamada nesse caso de *superclasse*. Quando uma mensagem não é encontrada em uma classe, é procurada na sua superclasse. Subclasses herdam dados e métodos de suas superclasses. Esse mecanismo de herança permite a existência de classes genéricas, de onde classes específicas são geradas.

3.3 Aplicações extensíveis

Estender significa adicionar módulos e funcionalidades a um programa existente. Os sistemas que utilizam orientação a objetos oferecem essa funcionalidade através dos mecanismos de encapsulação, polimorfismo e *late binding* [Pountain 1994].

Um objeto encapsulado se comporta como uma "caixa preta", cujo estado interno somente pode ser modificado através de uma interface bem definida. Uma mudança na sua implementação interna não deve afetar, caso a interface se mantenha inalterada, nenhuma aplicação que utilize este objeto.

Polimorfismo é a possibilidade de se ter uma operação aplicada a diferentes tipos e classes de objetos. Com o mecanismo de *late binding* não há a necessidade de se saber o tipo de um objeto até a execução da operação. Isto permite código antigo chamar código novo. Combinando polimorfismo com *late binding* é possível escrever códigos para classes e objetos que nem existem ainda e que irão funcionar apropriadamente. Isto é essencial para se obter sistemas extensíveis. Outros programadores têm que poder estender seu trabalho de maneira que não pode ser antecipado pelo programador original.

O paradigma de programação orientada a objetos oferece, ainda, aos programadores a oportunidade de re-usar significantes partes de seu código, através do mecanismo de herança. Isso acontece quando classes derivadas de outras classes incorporam o comportamento da classe base. Além disso, as modificações nas novas classes são feitas

de forma localizada: novas implementações ficam restritas às classes derivadas, já que externamente, devido ao uso de tipos abstratos de dados, essas modificações são transparentes.

Desse modo, objetos existentes podem ser armazenados em uma biblioteca para formar uma base para o desenvolvimento de programas. Essa é a idéia de aplicações extensíveis. A partir dessa base, projetistas podem desenvolver novas classes que possuem métodos específicos de uma determinada aplicação e adicionar essas classes à biblioteca original (Fig 3.1) [Forde 1989].

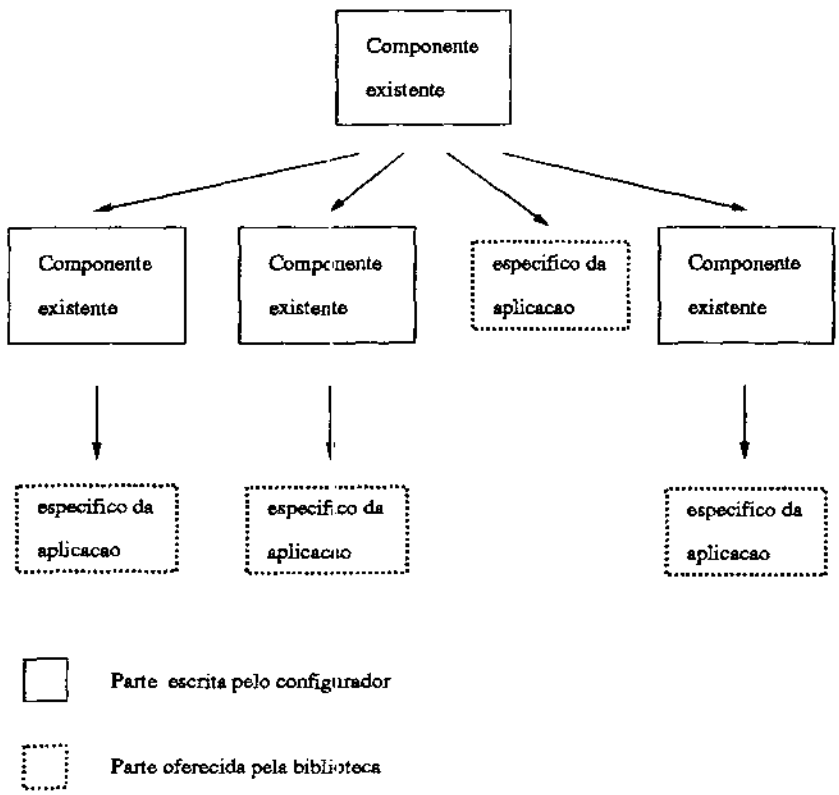

Figura 3.1: Arquitetura de uma aplicação extensível.

No entanto, Udell (1994) enfoca alguns problemas desse paradigma no aspecto referente a re-uso de códigos e apresenta como solução a comunicação entre processos. Um dos principais problemas é a dificuldade em se distribuir objetos, em forma binária, entre diferentes plataformas. Outro aspecto, consiste no fato de que estender pressupõe que módulos compilados separadamente sejam combinados, mas nem todas as linguagens orientadas a objetos possuem essa capacidade.

3.4 Ambiente para configuração de aplicações

Cada vez mais aumenta o número de pessoas envolvidas com o desenvolviemnto de programas. Em geral, esses novos programadores são profissionais qualificados nas áreas específicas em que atuam, mas possuem pouco conhecimento de computação. Para possibilitar o trabalho desses programadores ocasionais, são necessários mecanismos ou ferramentas, com interfaces adequadas, para acessar tecnologias já implementadas. Algumas características são necessárias observar para prover esses mecanismos.

3.4.1 Usuários avançados

Linguagens de programação e componentes de *software* estão atingindo um nível no qual usuários finais, com um determinado grau de conhecimento, podem ser capazes de construir seus próprios sistemas customizados. Nesse contexto, o desenvolvimento do produto final envolve diversos grupos de profissionais, com diferentes níveis de abstração para a realização das tarefas específicas de cada grupo (Fig. 3.2).

A tendência de se ter usuários finais como programadores evidencia algumas características de programação conforme discutido em [Cowan et al. 1992]. A característica principal é que, ao contrário da programação convencional, o programador e o usuário de uma parte do programa são a mesma pessoa. Desse modo, o código a ser escrito não possui o mesmo formalismo de um código convencional. O usuário final não vai construir um programa com a intenção de distribuir para uma vasta clientela. Esses usuários não criam, em geral, programas muito grandes. O objetivo é tornar mais produtivo o ambiente em que são realizadas as tarefas. O código é criado, muitas vezes, através de protótipos e não deve ter impacto em outros usuários. O que se deseja é automatizar o acesso às ferramentas computacionais utilizadas. Esses usuários são chamados de usuários avançados ou usuários configuradores.

Em geral, esses usuários configuradores têm pouco conhecimento formal de princípios de programação. A própria tarefa de aprender uma linguagem de programação é árdua e pouco animadora. O objetivo não é se tornar um profissional em programação, mas ser capaz de acessar tecnologias para solucionar problemas específicos.

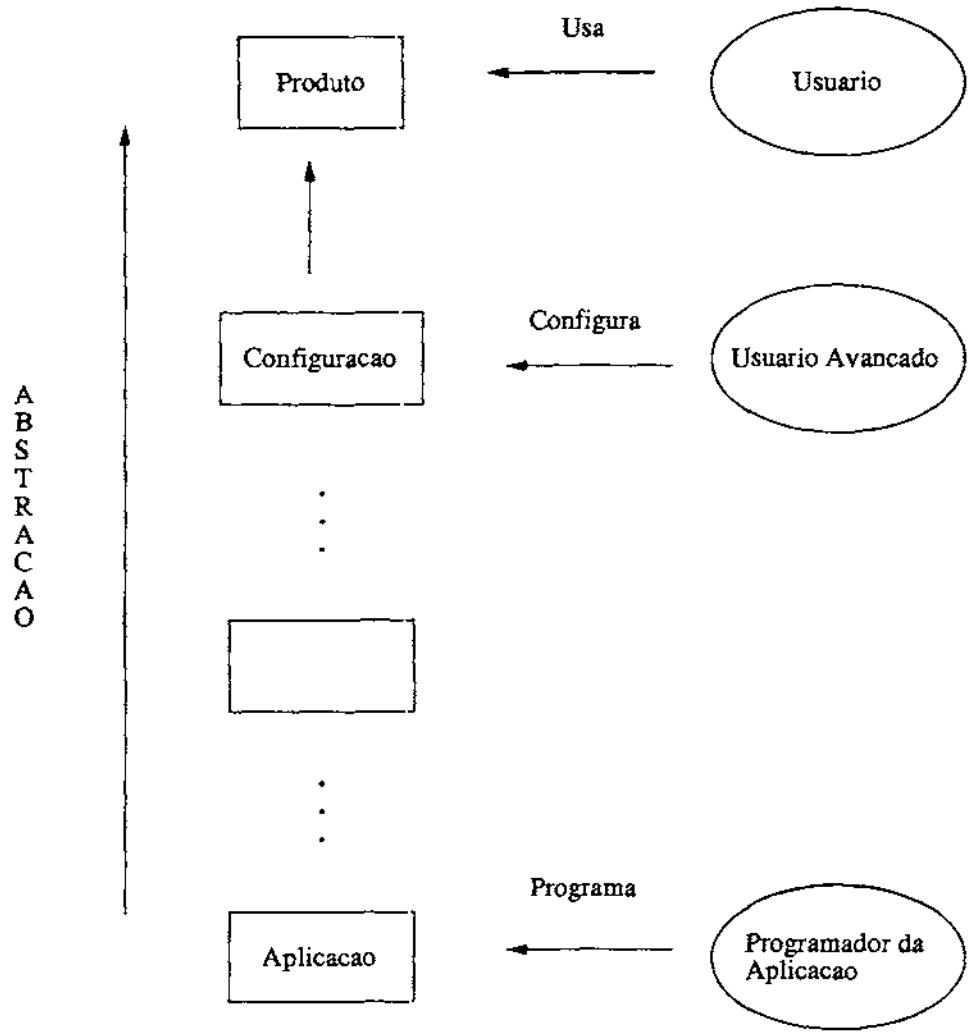

Figura 3.2: Níveis de abstração no processo de desenvolvimento de um produto [Celes 1995].

Para isso, é preciso que se tenha acesso a uma biblioteca de componentes de *software* re-usáveis que possam ser combinados em novas configurações. Esses componentes não vão ser criados pelos usuários, pois, para isso, é requisitado conhecimento especializado sobre a estrutura interna dos componentes e suas API (*Application Program Interface*). O que é necessário, portanto, é o acesso a tecnologias que permitam ao usuário interagir facilmente com esses componentes.

Essas tecnologias são, geralmente, baseadas em linguagens de programação, que variam das linguagens *shell* existentes nos sistemas Unix até sistemas baseados em objetos, como Hypercard ou Visual Basic.

A linguagem oferecida aos usuários configuradores deve ser simples e fácil de aprender. A linguagem também deve protegê-los de possíveis e frequentes erros de programação. Não é possível exigir desses usuários conhecimentos, por exemplo, para controlar ponteiros e gerenciamento de memória. Deve-se trabalhar com linguagens que oferecem os recursos de estruturação necessários, mas com elevado grau de abstração.

Para ser efetivamente útil, no entanto, as tecnologias deveriam oferecer métodos fáceis e intuitivos para acessar e combinar os componentes. Os serviços de configuração de aplicações deveriam dispensar as linguagens de programação tradicionais e permitir a programação via recursos mais abstratos. Nesse sentido, muita pesquisa tem sido feita em programação visual [Chang 1990]. No entanto, apesar dos progressos, as linguagens de programação visuais ainda não são suficientes para atender a todas as tarefas de programação.

Uma estratégia intermediária, e que tem alcançado bastante sucesso, é utilizada na construção de interfaces convencionais. Essas interfaces são compostas de objetos, como botões, listas e diálogos, e mecanismos de seleção que permitem ao usuário interagir com outros programas. Apesar de complexa, a tarefa de construir esse tipo de interface pode ser bastante simplificada através uma camada de abstração criada por recursos visuais. Um programa "hello world", em Visual Basic [Microsoft Corporation 1993], pode ser escrito em 1 linha de código, enquanto o mesmo programa, utilizando diretamente as funções do Windows, requer 100 linhas [Petzold 1993]. O êxito da programação visual nessas tarefas se deve principalmente a dois aspectos: primeiro, interfaces são compostas por um número pequeno de objetos (menus, botões, listas, etc.) e, segundo, interfaces

são naturalmente visuais.

Para implementação de procedimentos mais gerais, no entanto, ainda é preciso uma linguagem de programação convencional [Cowan et al. 1992]. A tarefa de configurar uma aplicação, por exemplo, ainda requer algum conhecimento de programação procedimental, mas é desejável que o uso dessa programação seja minimizado à medida que as tarefas dos usuários configuradores sejam automatizadas.

Um outro problema reside no fato de que cada aplicação configurável, geralmente, possui a sua própria linguagem de configuração. Isso dificulta ainda mais a tarefa do usuário-configurador, que deve aprender diferentes sintaxes. Uma linguagem padrão ameniza esses problemas, conforme será discutido adiante.

3.4.2 Linguagens de configuração

Aplicações complexas são melhores estruturadas se compostas de dois ou mais componentes: um componente principal, que implementa os serviços oferecidos pela aplicação, e componentes auxiliares que oferecem acesso programável aos serviços existentes.

Os módulos auxiliares são, tipicamente, linguagens interpretadas, pequenas e simples (*little languages*), em contraste com as "grandes" linguagens na qual as aplicações são normalmente escritas. Atualmente, no entanto, diversas aplicações são escritas com as próprias linguagens de configuração. Algumas dessas linguagens são projetadas de uma maneira *ad hoc* e não tem relação com nenhuma outra linguagem. Outras linguagens são baseadas em linguagens existentes (C ou LISP) [Valdés 1991]. Estas linguagens são, geralmente, específicas para uma determinada tarefa.

Há bastante tempo o desenvolvimento de pequenas linguagens associadas às aplicações é visto como uma solução adequada para promover o acesso programável aos serviços implementados [Bentley 1986]. Hoje, é crescente o número de novas pequenas linguagens devido, principalmente, à existência de ferramentas utilitárias para construção de compiladores [Valdés 1991].

Essas linguagens permitem um ambiente de programação no qual o usuário pode customizar e estender a aplicação para fins específicos. Um uso comum dessas linguagens

pode ser notado na interface com sistemas operacionais, como por exemplo, o acesso a funcionalidades do DOS, através de arquivos *batch*, ou arquivos *script* em sistemas Unix, ou na interface com ambientes gráficos como Microsoft Windows.

Simples linguagens de extensão permitem, no entanto, somente a definição de parâmetros e sequências de ações, o que pode não ser suficiente. Muitas vezes, é necessário uma comunicação, entre a aplicação e a linguagem de extensão, nos dois sentidos. Essas linguagens são referidas, nesta tese, como *linguagens acopladas*. Exemplos dessas linguagens são Lua [Ierusalimschy et al. 1994, Figueiredo et al 1994] e Tcl [Osterhout 1994].

Linguagens acopladas são linguagens que necessitam de um programa hospedeiro. Não existe o conceito de programa principal: a linguagem depende de um programa hospedeiro, que pode chamar funções da linguagem para serem executadas e ainda registrar funções escritas na linguagem nativa do programa hospedeiro para serem chamadas pela linguagem acoplada. Essas linguagens possuem além da sua própria sintaxe, uma API (*Application Program Interface*) para permitir essa interface com o programa hospedeiro.

As linguagens acopladas podem ser estendidas pelo programa hospedeiro, através de comandos específicos a um determinado domínio. Deste modo a linguagem de extensão é customizada para fins específicos ao programa hospedeiro. Desse modo, é possível criar diversas linguagens de programação customizadas, com a mesma sintaxe [Beckman 1991]. O modo como isto é feito varia de acordo com a linguagem.

Estas linguagens aumentam o poder das ferramentas, permitindo aos configuradores da aplicação escrever programas nessas linguagens que estendem os comandos próprios da ferramenta. Para os usuários finais, os comandos específicos aparecem como se fossem comandos originais.

3.5 Uma arquitetura para configuração de aplicações

O acesso programável aos serviços oferecidos por uma aplicação configurável pode ser feito através do acoplamento de linguagens específicas. Neste caso, para cada aplicação, implementa-se uma linguagem para dar suporte às possibilidades de configuração da aplicação. A Fig. 3.3 ilustra a arquitetura de aplicações desenvolvidas com esta estratégia.

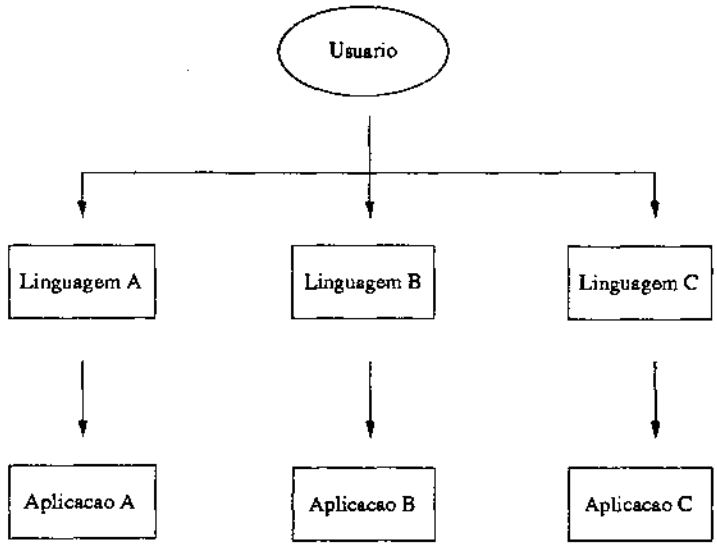

Figura 3.3: Arquitetura utilizando-se diversas linguagens de configuração [Celes 1995].

A desvantagem dessa estratégia está na necessidade de se desenvolver uma linguagem específica para cada aplicação. Com isto, exige-se a implementação de compiladores específicos que, mesmo com ferramentas apropriadas do tipo *lex e yacc* [Levine-Brown 1992], são programações não triviais. Além disso, com essa estratégia, o modo de configuração de cada aplicação apresenta uma sintaxe específica, o que dificulta o aprendizado por parte dos usuários.

A fim de contornar esses problemas, deve-se utilizar uma estratégia com uma única linguagem para dar suporte às diversas aplicações que necessitam oferecer serviços de configuração. Nesse caso, a própria linguagem é configurável e pode ser estendida para atender finalidades mais específicas. Isto evita a criação de vários compiladores e propicia a obtenção de uma sintaxe uniforme para as diversas aplicações. A Fig. 3.4 ilustra a implementação desta estratégia.

Nesse contexto, o programador de aplicação configura a linguagem para atender as tarefas específicas de seu sistema, trabalhando em um nível de abstração bastante superior ao desenvolvimento de compiladores. Por outro lado, um usuário familiarizado com uma aplicação desenvolvida com essa base, não encontra dificuldades em aprender os serviços de configuração de uma outra aplicação, já que ambas apresentarão uma sintaxe uniforme. Isto justifica o investimento inicial do usuário em aprender uma nova linguagem [Celes 1995].

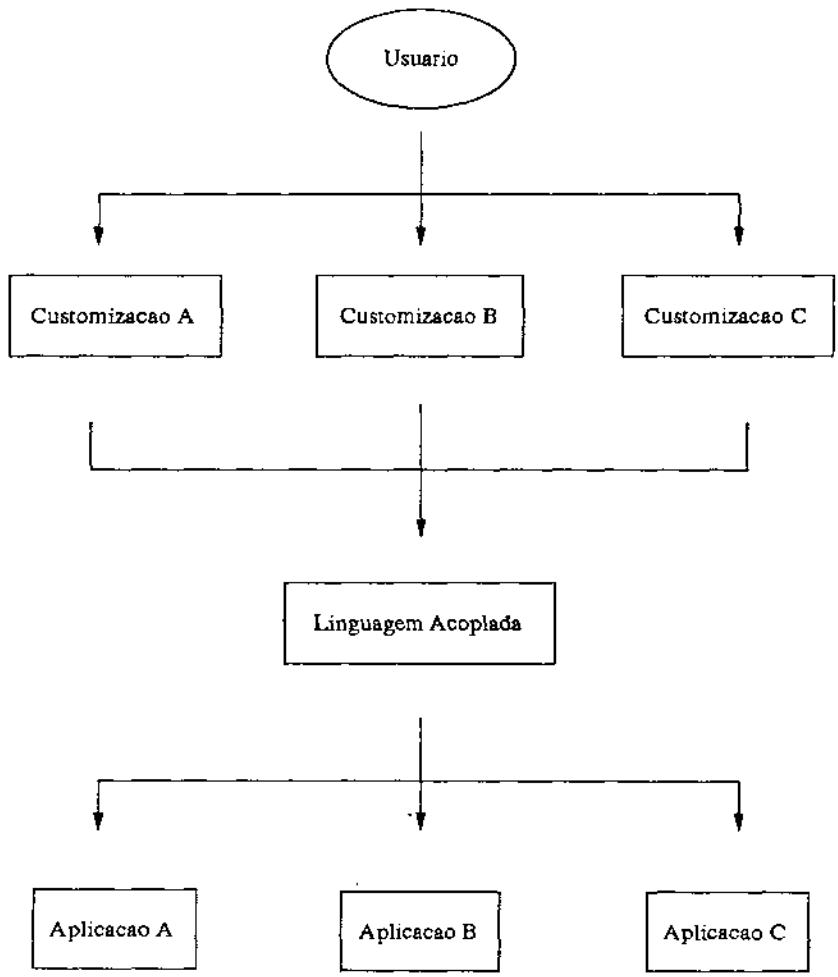

Figura 3.4: Arquitetura utilizando-se somente uma linguagem de configuração [Celes 1995].

Conforme mencionado anteriormente, programadores de aplicações podem estender a linguagem, construindo serviços com alto grau de abstração orientados para um determinado domínio de interesse. Isso é feito através do cadastramento de novas funções. Dessa forma, a configuração pode acessar a tecnologia da aplicação através de serviços específicos, além de poder contar com os recursos de programação inerentes à linguagem de configuração.

A Fig. 3.5 ilustra a distribuição dos módulos numa aplicação típica que utiliza essa estratégia. O configurador tem acesso tanto aos recursos convencionais da linguagem de configuração (operadores, controladores de fluxo, etc.) quanto aos recursos oriundos da extensão (novas funções específicas).

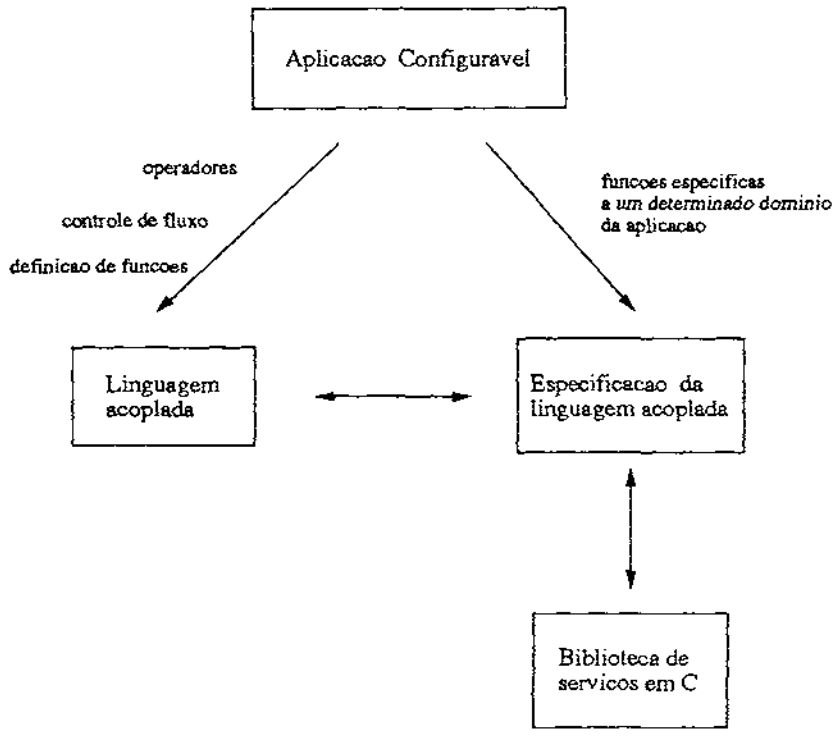

Figura 3.5: Arquitetura de aplicações que utilizam linguagens acopladas [Celes 1995].

3.6 Caracterização do domínio da aplicação

O domínio das aplicações é caracterizado pelo conjunto de propriedades que podem estar associadas ao modelo (objeto). Essas propriedades, referidas como atributos, representam a semântica das aplicações e, portanto, caracterizam as funcionalidades oferecidas

ao usuário final. Existem aplicações que utilizam a mesma tecnologia, mas devido à semântica são aplicações completamente diferentes. Com o intuito de se obter uma maior flexibilidade e aproveitar de forma mais eficiente essa tecnologia, é desejável que a especificação dos atributos seja feita de forma independente da tecnologia da aplicação e que se tenha mecanismos para a configuração desses atributos. Isto permite uma mesma aplicação ser utilizada em diversos domínios específicos.

O paradigma de programação orientada a objetos permite a estruturação de classes direcionadas a um determinado domínio de aplicações. Nesse caso, as classes incorporam conceitos semânticos relacionados com esse domínio de interesse e, portanto, podem ser enumeradas atendendo à totalidade das aplicações específicas dentro desse domínio. Extensões da aplicabilidade a novos domínios podem ser alcançadas através dos mecanismos de herança e de redefinição de métodos.

A generalização dessa estratégia em domínios mais amplos pressupõe a eliminação de qualquer semântica na definição das classes: a semântica associada a um gerador de malhas de elementos finitos não tem qualquer relação com a semântica de representação de mapas cartográficos, embora as duas aplicações possam ser desenvolvidas sobre um mesmo gerenciador de subdivisões planares. Um sistema utilizando essa estratégia é apresentado por Celes Filho (1995).

Nesta tese os conceitos de configuração de atributos, que caracterizam uma determinada aplicação, e de programação orientada a objetos são combinados para definir e criar um ambiente extensível para simulações de mecânica computacional baseadas em geometria, tal como discutido no capítulo anterior. O próximo capítulo descreve a arquitetura de tal ambiente.

Capítulo 4

Um ambiente extensível para simulações de mecânica computacional

Este capítulo apresenta a arquitetura proposta para um ambiente de configuração de atributos para simulações de mecânica computacional. Essa arquitetura permite a definição de atributos específicos a um determinado tipo de simulação e a associação desses atributos às entidades de um modelo geométrico (fornecido por algum modelador), sem necessidade de recompilação e re-ligação da aplicação. Como os sistemas de análise são tipicamente muito grandes, a recompilação de módulos e posterior re-ligação não é uma alternativa prática. A configuração dos atributos é feita de maneira simples e independente do modelador geométrico utilizado.

A estratégia adotada combina conceitos discutidos nos dois capítulos anteriores. É utilizada a abordagem de modelagem baseada em geometria, na qual os atributos são associados à descrição geométrica do objeto e o modelo numérico (para análise por elementos finitos, por exemplo) é obtido a partir do modelo geométrico, ou seja, através da discretização da descrição geométria e do mapeamento dos atributos das entidades geométricas para as entidades da malha (nó e elemento).

O mecanismo para a configuração dos atributos utiliza uma única linguagem acoplada, estendida com comandos específicos para tarefas referentes à especificação de atributos. Uma biblioteca de classes, dentro do paradigma de programação orientada a objetos, direcionadas para esse domínio de aplicação forma a base a partir da qual se faz a

configuração da aplicação, através da redefinição de métodos das classes existentes, ou a partir da criação de novas classes de atributos.

4.1 Aspectos de um ambiente para simulação de mecânica computacional

Conforme discutido no Capítulo 2, um ambiente para simulação de mecânica computacional deve apresentar as seguintes funcionalidades:

- Mecanismos para gerar o modelo geométrico.
- Mecanismos para especificar atributos e para associar esses atributos às entidades do modelo geométrico.
- Mecanismos para gerar a discretização do objeto e para mapear os atributos das entidades geométricas para as entidades da discretização (o modelo numérico consiste dos atributos associados à discretização).
- Módulos para a análise numérica e para a visualização dos resultados fornecidos pela análise.

Alguns aspectos desse processo são independentes do tipo de análise numérica a ser efetuada, tais como a modelagem geométrica e a discretização do objeto. Uma determinada discretização, por exemplo, pode ser utilizada em diferentes análises. O que especifica o tipo de análise são os atributos de simulação existentes.

Geralmente, os sistemas utilizados para simulações de mecânica computacional são muito grandes e genéricos, dificultando o uso e a manutenção, ou são específicos para uma aplicação em particular. Nesse segundo caso, é difícil a sua utilização em novas aplicações, pois adicionar novas funcionalidades para atender a diferentes tipos de simulações requer, normalmente, um conhecimento detalhado do código da implementação.

Utilizar um sistema para diferentes simulações consiste em especificar novos tipos de atributos, já que os outros aspectos da simulação, como a modelagem geométrica, a geração da malha e a visualização dos resultados são independentes do tipo de análise.

4.2 Configuração de aplicações direcionadas para um domínio específico: mecânica computacional

O domínio de uma aplicação é definido através da especificação de seus atributos, que representam a semântica da aplicação. Conforme discutido no capítulo anterior, é desejável que esses atributos sejam especificados independentemente da tecnologia da aplicação e que esses atributos possam ser configurados para diferentes domínios específicos. Desse modo, uma aplicação configurável consiste de uma parte referente à tecnologia da aplicação e de mecanismos para configuração dos atributos específicos da aplicação.

Esta tese particulariza o domínio das aplicações para problemas de mecânica computacional. Esse domínio, no entanto, engloba diversos subdomínios que se referem aos diferentes tipos de simulação, como análise linear-elástica de propagação de trincas, análise térmica, etc. Nesse contexto, uma aplicação configurável se refere a um sistema configurável para simulação de mecânica computacional e uma aplicação específica se refere à utilização desse sistema para um determinado tipo de análise, ou seja, a especificação dos atributos para esse caso particular. Os atributos se referem às informações, adicionais à descrição da geometria, necessárias para a definição do problema. A Fig. 4.1 particulariza a arquitetura mostrada na Fig. 3.5 para o domínio específico de simulações de mecânica computacional baseada em geometria. A linguagem acoplada utilizada para a configuração dos atributos é estendida com comandos relacionados à realização dessa tarefa.

No caso da configuração dos atributos ser feita de forma independente, o modelador geométrico não tem nenhum conhecimento da semântica incorporada pela configuração. Sendo assim, a estratégia de configuração deve oferecer mecanismos para que as edições sobre as entidades do modelo geométrico sejam validadas pelo ambiente de configuração. Desse modo, a estratégia de configuração de atributos envolve a tarefa de especificação dos atributos e a tarefa de validação das ações sobre as entidades do modelo geométrico.

Considerando-se os aspectos discutidos, é proposta uma arquitetura para o desenvolvimento de aplicações de mecânica computacional, que possam ser facilmente estendidas e configuradas para fins específicos. Para isso, é utilizada a estratégia mencionada no

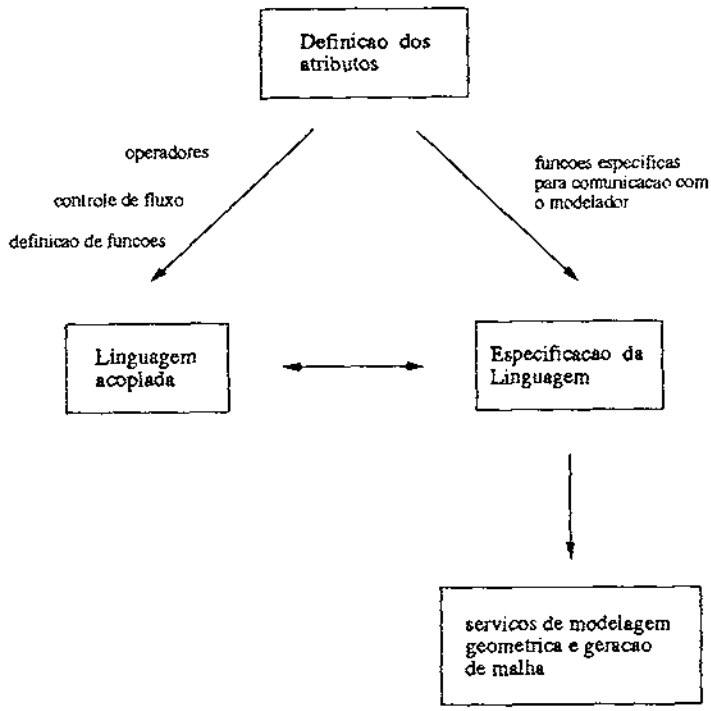

Figura 4.1: Arquitetura de um sistema configurável para simulação de mecânica computacional.

capítulo anterior de uma estruturação de classes, dentro do paradigma de programação orientada a objetos, que incorporam conceitos semânticos relacionados com um determinado domínio de interesse. Desse modo, é idealizada uma biblioteca básica de classes relacionadas com o processo de geração de modelos numéricos para uma simulação de mecânica computacional (geometria, nós, elementos, atributos, etc.). Nesse ambiente, é possível modificar ou criar novos tipos de atributos de acordo com o tipo de simulação. A especificação dos atributos é feita independentemente da descrição geométrica e da discretização do modelo. Desse modo, é possível utilizar os módulos responsáveis por esses serviços para diferentes tipos de análise, o que representa uma grande vantagem, considerando-se a complexidade desses serviços. O esforço para atualizar ou mesmo adicionar novas funcionalidades ao sistema deve ser pequeno.

4.3 Arquitetura proposta

O ambiente proposto não se destina a tratar de detalhes referentes aos serviços de modelagem, tal como detalhes da geometria do modelo ou da geração da malha, mas sim dos

aspectos referentes à especificação dos atributos de simulação: Assume-se que os serviços de modelagem são fornecidos por algum modelador.

Os atributos, no entanto, devem ser associados às entidades geométricas. Além disso, a especificação de um atributo pode requisitar informações referentes ao modelador como, por exemplo, o valor da normal de uma superfície em um determinado ponto. É necessário, portanto, uma comunicação entre os atributos e os serviços de modelagem. De modo a permitir que a especificação dos atributos seja feita de forma independente do modelador geométrico, essa comunicação deve ser feita através de um mecanismo que mantenha a independência dos módulos envolvidos na tarefa, ou seja, o módulo responsável pela definição dos atributos não deve ter conhecimento sobre as características específicas do modelador geométrico, assim como o modelador deve ser independente da especificação dos atributos.

Nesse sentido, a arquitetura proposta divide a aplicação em três partes. Uma parte representa os serviços ou tecnologia da aplicação, composta pelos aspectos que são independentes do tipo da análise, como a modelagem geométrica, a geração da malha e a visualização dos resultados. Uma outra parte refere-se aos aspectos específicos a uma aplicação particular, ou seja, a definição dos atributos. A terceira parte, refere-se a um sistema responsável pela comunicação entre as duas partes anteriores. Esse sistema permite o acesso programável à tecnologia da aplicação.

As tarefas referentes à configuração dos atributos são feitas através de uma linguagem acoplada, estendida com comandos específicos para esse fim. Essa linguagem será referida ao longo desta tese como linguagem de configuração. A linguagem utilizada para implementação dos serviços da aplicação será referida como linguagem de programação. Uma aplicação construída com a arquitetura proposta consiste de (Fig. 4.2):

- Um módulo para modelagem de sólidos com suporte para mecânica computacional. Esse módulo representa os serviços da aplicação e é reponsável pela geração do modelo geométrico, pela discretização e pela visualização dos resultados. Esse módulo será referido a partir deste momento, de um modo simplificado, como modelador. Deve-se entender por modelador, nesse contexto, a capacidade de descrição da geometria, geração de malha e visualização dos resultados.

- Uma linguagem para configuração, como Tcl [Ousterhout 1994] ou Lua [Ierusalimschy *et al.* 1994, Figueiredo *et al.* 1994], estendidas com mecanismos para suportar a definição de classes, no contexto de programação orientada a objetos (Lua já possui esse mecanismo). A configuração da aplicação é feita através da redefinição dos métodos das classes existentes e da definição e implementação de novas classes, com seus respectivos métodos. Esse módulo é referido como a parte configurável da aplicação.

- Um *toolkit*, como Tk [Ousterhout 1994] ou EDG [Celes 1994], que oferece uma interface gráfica para acessar e manipular as classes definidas.

- Uma coleção de classes básicas, que definem, com alto grau de abstração, as entidades normalmente envolvidas no processo de geração dos modelos geométrico e numérico e que sejam comuns a um grande domínio de simulações (*Material, Boundary Conditions, Edge, Vertex, Element, Node,* etc.). Essas classes são referidas ao longo desta tese como *Core Classes*.

- Um conjunto de funções que são responsáveis pelo acesso à base de dados do modelador. Qualquer consulta a essa base de dados é feita através dessas funções. Desse modo é criada uma camada de abstração sobre a base de dados, que esconde os detalhes de como as informações requisitadas são obtidas.

- Um conjunto de funções que são responsáveis pela ligação (*binding*) entre as *core classes* e as funções mencionadas acima.

Na Fig. 4.2, os retângulos com linha tracejada correspondem aos módulos que não fazem parte do sistema proposto, ou seja, o módulo referente aos serviços da aplicação (modelador) e o arquivo de configuração. Nesse arquivo, é feita a configuração dos atributos (especificação de novos tipos de atributos para uma determinada simulação), onde podem ser redefinidos os métodos das classes existentes e criadas novas classes de atributos.

Os retângulos com linhas cheias representam o sistema proposto, composto dos módulos que permitem o acesso programável ao modelador, feito em diversos níveis (desde o usuário final até o programador da aplicação). Nesse sentido, são idealizados diferentes grupos de responsáveis pelo desenvolvimento da aplicação, com crescente nível de abstração para a realização das tarefas. Os grupos diferem entre si de acordo com seus

conhecimentos de programação e com as tarefas que devem desempenhar, conforme será discutido posteriormente. Os três módulos serão descritos a seguir.

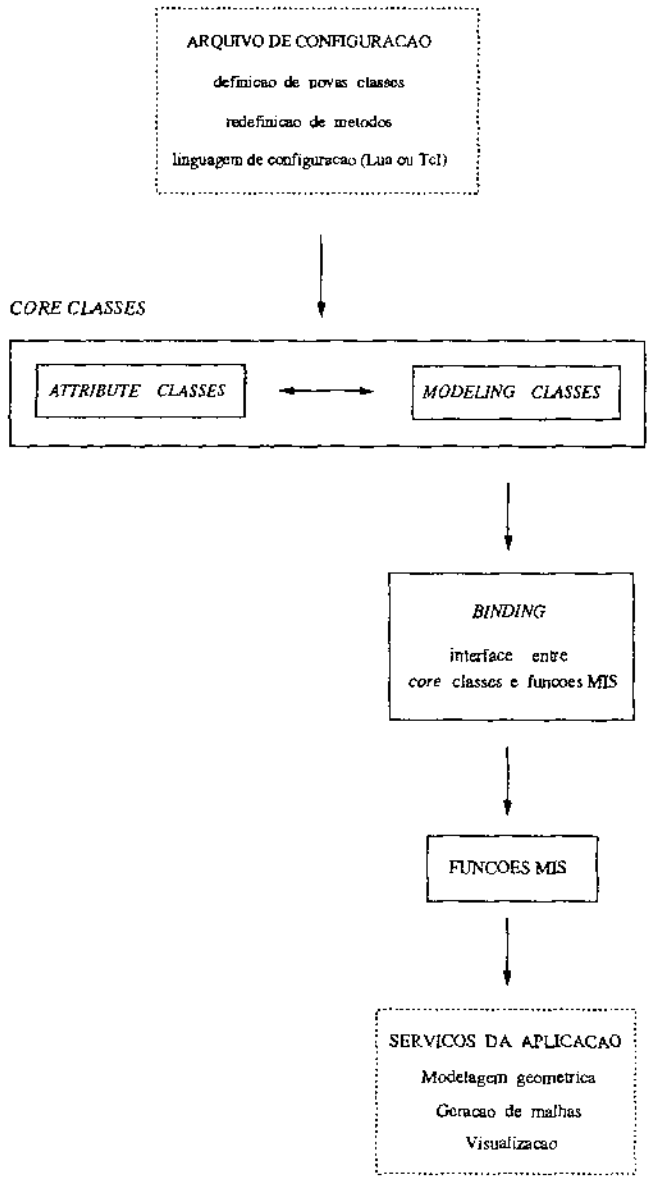

Figura 4.2: Arquitetura do sistema.

4.3.1 Core classes

Esta seção descreve a funcionalidade das *core classes*, ou seja, o papel dessas classes na arquitetura proposta. A descrição dos métodos de cada classe será feita no próximo capítulo.

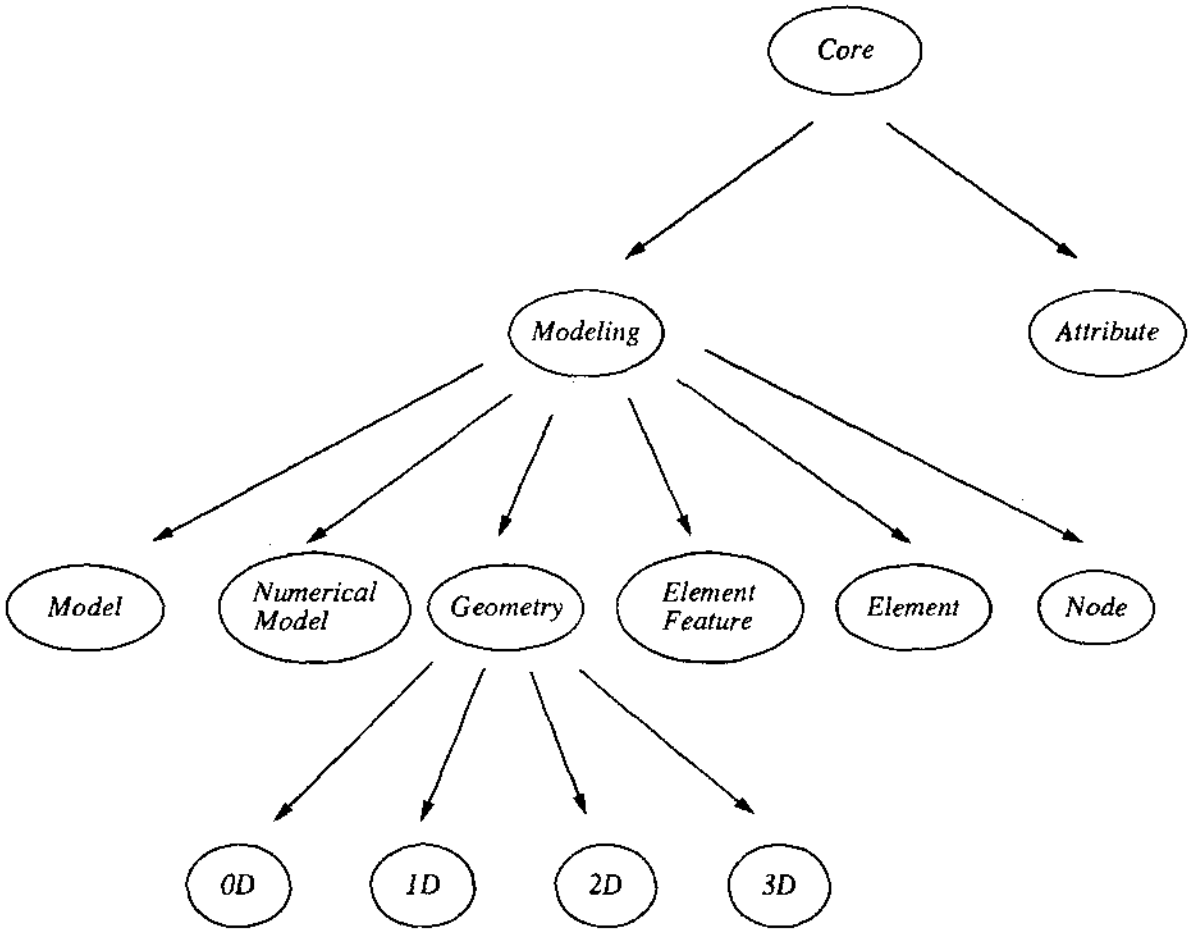

Figura 4.3: Organização das *Core Classes*.

As *core classes* formam uma biblioteca de classes que representam as grandezas comuns envolvidas em uma simulação de mecânica computacional baseada em geometria. Essas classes representam a abstração dos modelos geométrico e numérico, das entidades referentes a esses modelos e dos atributos. A hierarquia dessas classes é organizada de modo consistente com a arquitetura proposta, ou seja, as classes referentes aos atributos (*attribute classes*) são definidas independentemente das classes referentes aos modelos geométrico e numérico (*modeling classes*) (Fig. 4.3).

A associação entre os atributos e as entidades geométricas é armazenada pelo serviço de configuração, independentemente do modelador. O modelador não armazena e nem tem acesso aos objetos referentes aos atributos.

Classe Attribute

A classe *Attribute* representa a superclasse dos atributos necessários à geração do modelo geométrico. Subclasses derivadas dessa classe devem ser criadas para atender aos requisitos específicos de cada simulação, como por exemplo, condições de contorno e materiais. As instâncias dessas classes contêm as propriedades que definem os tipos de atributos, assim como referências para os objetos que representam as entidades geométricas às quais os atributos estão associados (objetos das subclasses da classe *Geometry*). Esses objetos são referenciados como objetos *atributo*.

Os objetos atributo são referenciados por um identificador. Esse identificador representa um rótulo (*label*), implementado através de uma cadeia de caracteres. O *label* de cada objeto pode ser definido explicitamente pelo usuário ou pode ser implicitamente construído pelo sistema. As novas classes são criadas em um arquivo de configuração, que é lido pela aplicação e que pode, devido ao fato da linguagem de configuração ser interpretada, ser modificado sem necessidade de recompilação do sistema.

Modeling classes

A classe *Modeling* é a superclasse das classes que representam as entidades abstratas envolvidas no processo de geração dos modelos geométrico e numérico de uma simulação. Desse modo, a hierarquia dessa classe inclui as subclasses: *Model, Numerical Model, Geometry, Element, Node, Element Feature.*

Conforme mencionado anteriormente, no processo de geração do modelo geométrico, os objetos atributo devem ser associados às entidades do modelador e, além disso, a especificação dos atributos pode requisitar informações do modelador. As *modeling classes* são responsáveis pela interface dessa comunicação entre o ambiente de configuração e o modelador. Toda ação referente à configuração dos atributos que requisitar informações do modelador deve ser feita através dessas classes, que permitem, desse modo, o acesso programável às entidades existentes no modelador.

A classe *Model* e *Numerical Model* implementam a abstração dos próprios modelos geométrico e numérico, e contêm as informações que devem existir nesses modelos. A

classe *Model* contém uma coleção dos objetos que representam as entidades geométricas do modelo, juntamente com uma coleção dos objetos que representam os atributos especificados (objetos das subclasses de *Geometry* e *Attribute*). A classe *Numerical Model* contém uma coleção de objetos que representam as entidades da discretização do modelo (objetos das subclasses de *Node* e *Element*).

Classe Geometry

Esta classe é a superclasse das classes que implementam a abstração das entidades geométricas existentes no modelador. As subclasses dessa classe referem-se à dimensão possível de uma entidade e consistem das classes: *0D, 1D, 2D, 3D.*

Os objetos dessas subclasses, referidos como objetos *geométricos*, contêm referências para as entidades geométricas do modelador e referências para os objetos atributo associados a essas entidades.

Um objeto geométrico é criado somente quando se aplica um atributo a uma entidade geométrica. Desse modo, só existem objetos geométricos referentes às entidades que possuem atributos associados. A referência à entidade geométrica, armazenada pelo objeto geométrico, é feita através de um identificador *id*, que pode ser qualquer valor que identifica a entidade no contexto do modelador. Esse *id* deve ser imutável, para evitar que, no caso do modelador alterar o seu valor, se atualize o objeto geométrico no ambiente de configuração, o que não é desejável. Desse modo, o *id* não deve ser, por exemplo, um endereço de memória.

Classe Node

Esta classe representa a abstração dos nós da discretização do modelo. É criado um objeto para cada nó da malha. Um objeto dessa classe referencia o objeto geométrico correspondente à entidade geométrica à qual o nó representado está associado. Por exemplo, se um nó da malha no modelo geométrico está sobre uma aresta, o objeto correspondente a esse nó referencia o objeto geométrico correspondente à aresta.

De acordo com a estratégia de modelagem baseada em geometria, os atributos são associ-

ados somente aos objetos geométricos. Desse modo, para a geração do modelo numérico para a análise, é necessário mapear os atributos dos objetos geométricos para os objetos da classe *Node*. Esse mapeamento, no entanto, é feito automaticamente pelo sistema, através da referência ao objeto geométrico.

Classe Element

Esta classe representa a abstração dos elementos da discretização do modelo. É criado um objeto para cada elemento da malha.

Do mesmo modo como para os objetos da classe *Node*, é necessário se fazer o mapeamento dos atributos dos objetos geométricos para os objetos da classe *Element*. O mecanismo para isso, no entanto, difere do utilizado pelos objetos da classe *Node*. Devido ao fato de em métodos numéricos, como o método dos elementos finitos, ser possível a associação de atributos a partes de um elemento, como um lado de um elemento plano ou uma face de um elemento sólido, não existe uma correspondência direta entre a entidade topológica que contém o atributo e o elemento finito.

Para se referenciar atributos associados a partes de elementos, foi criada uma classe (subclasse de *Modeling*) denominada *Element Feature*. Desse modo, cada objeto da classe *Element* contém uma lista de objetos da classe *Element Feature*.

Um objeto da classe *Element Feature*, assim como na classe *Node*, referencia o objeto geométrico correspondente à entidade geométrica, à qual a parte do elemento representado está associado.

A Fig. 4.4 ilustra a relação dos objetos das classes *Node* e *Element* com os objetos atributos e geométricos, que permite o mapeamento dos atributos das entidades geométricas para as entidades da malha.

4.3.2 Funções para acessar à tecnologia da aplicação

Conforme mencionado anteriormente, a especificação de um atributo pode requisitar informações referentes ao modelador. Essas informações são armazenadas na base de

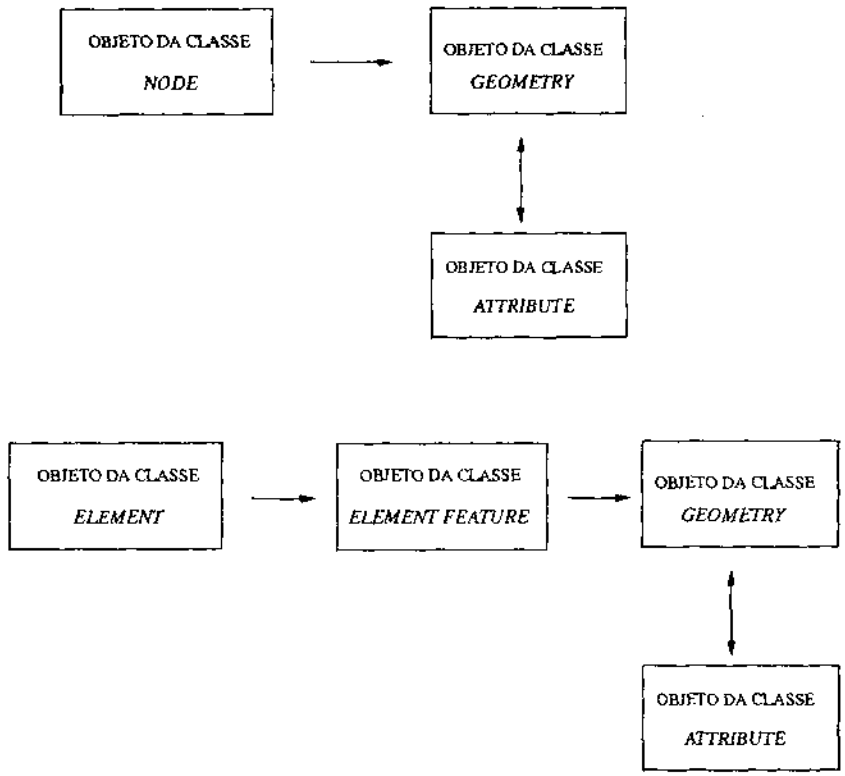

Figura 4.4: Relação entre objetos das classes *Node e Element, Geometry e Attribute.*

dados do modelador, embora possam não estar disponíveis diretamente (no exemplo citado anteriormente, o valor da normal pode não estar armazenado explicitamente, sendo necessário o seu cálculo através de outras informações). O meio como as informações requisitadas são obtidas é, portanto, dependente da implementação da base de dados. Nesse sentido, é desejável que essas informações sejam obtidas através de chamadas procedurais ao contrário de se acessar explicitamente a estrutura de dados do modelador, ou seja, deve existir uma interface entre a base de dados do modelador e o restante da aplicação. Essa interface é feita através de um conjunto de funções que são responsáveis por extrair as informações requisitadas e que representam uma camada de abstração sobre a base de dados do modelador.

Essas funções são implementadas através de operadores, que são procedimentos para desempenhar uma tarefa específica. O cliente é protegido de procedimentos internos e da estrutura de dados do programa. A comunicação é feita somente através da API *(Application Program Interface)*, ou seja, das variáveis dos argumentos desses operadores, que não contêm nenhum tipo de estrutura de dados. Essa filosofia de "caixa preta" retira do cliente a responsabilidade em saber detalhes sobre o código do modelador e por outro

lado oferece flexibilidade para se usar e acessar os dados. Um operador efetivo deve ser genérico o suficiente para ser utilizado por uma grande audiência, ainda que deva ser específico o bastante para realizar a tarefa determinada. No caso de modeladores geométricos, esses operadores retornam o resultado de algum tipo de operação realizada sobre a base de dados desse modelador.

Nesse contexto, o sistema proposto especifica um conjunto de funções, referidas como MIS (*Modeling Interface Specification*), necessárias para a comunicação entre o ambiente de configuração e o modelador. Esse conjunto de funções deve ser suficiente para permitir a realização das tarefas referentes à especificação dos atributos requisitados para a simulação. No caso de se utilizar um outro modelador geométrico, é necessário somente reescrever as funções MIS para este novo modelador, sem nenhuma alteração no restante da aplicação. Os operadores que implementam essas funções são construídos pelos desenvolvedores do modelador, sendo que a maior parte podem ser extraídos diretamente da funcionalidade de modelagem, inerente ao modelador.

Desse modo, evita-se a necessidade de repetição de toda a capacidade de modelagem geométrica pelo sistema, o que representa uma simplificação significante, já que os modeladores são programas complexos (tipicamente mais de cem mil linhas de códigos). Essa vantagem é apreciada em ambientes que utilizem diferentes modeladores para tarefas similares.

Uma outra vantagem obtida com a utilização dessas funções MIS consiste em poder se utilizar, como módulo responsável pelos serviços de modelagem, códigos já existentes ou futuros desenvolvimentos. No caso de códigos existentes, a construção dos operadores que implementam as funções MIS pode, em alguns casos, impor algumas modificações no código, pois nem sempre as informações que devem ser obtidas por estes operadores estão disponíveis diretamente na sua base de dados (exemplo da normal citado acima). No caso de programas novos, a implementação do código do modelador pode ser feita, desde o início, de modo a facilitar a tarefa dos operadores.

As funções necessárias para as tarefas de especificação dos atributos podem ser classificadas em três grupos, de acordo com suas funcionalidades (a lista completa das funções MIS é apresentada em [Silveira 1995]):

- Funções de inicialização - são funções responsáveis por informações fundamentais ao funcionamento do sistema. Essas funções são chamadas automaticamente pelo sistema ao ser inicializado. Por exemplo:

  MisRegGeomEnt () - essa função registra os tipos de entidades geométricas existentes no modelador, conforme será visto no próximo capítulo.

- Funções de seleção - são funções responsáveis pelas informações referentes às entidades selecionadas pelo usuário. Por exemplo:

  MisGetSelect () - essa função retorna uma lista com as entidades geométricas que estão selecionadas.

- Funções de acesso - são funções responsáveis por informações que estão armazenadas na base de dados do modelador. Por exemplo:

  MisGetCoords () - essa função retorna as coordenadas de um determinado nó.

4.3.3 Binding

O módulo *binding* é responsável pela comunicação entre a parte configurável da aplicação e o modelador. A existência deste módulo assegura uma clara independência entre o ambiente de configuração e o modelador geométrico. Na parte configurável, os métodos das *modeling classes* representam a interface para acessar as informações do modelador. No modelador, as funções MIS disponibilizam o acesso a estas informações.

O acesso às funções MIS pela parte configurável da aplicação representa uma extensão da linguagem de configuração para este domínio, conforme explicado no Capítulo 3. A forma como as funções do programa hospedeiro (escritas em C) são acessadas pela parte configurável depende da linguagem de configuração utilizada, e deve ser feita por um módulo independente tanto do modelador como da parte configurável. Caso contrário, exigiríamos que o implementador das funções MIS tivesse acesso à linguagem de configuração, ou que o configurador tivesse acesso à linguagem hospedeira.

Através do módulo *binding*, o sistema de configuração proposto cria uma interface abstrata entre a parte configurável e o modelador geométrico. O módulo *binding* é responsável por estender a linguagem de configuração. Com isso, na parte configurável

da aplicação, acessa-se os dados do modelador através dos métodos das *modeling classes*. No modelador, escreve-se as funções MIS que são posteriormente acessadas pelas *modeling classes* através do módulo *binding*.

Com esta arquitetura, cria-se uma importante independência entre os grupos responsáveis pelo desenvolvimento da aplicação como um todo. Por um lado, o responsável pela implementação do modelador escreve as funções MIS sem conhecer características da linguagem de configuração utilizada. Por outro, o configurador acessa os dados do modelador através de métodos das classes definidas no próprio ambiente de configuração. Assim, uma eventual troca de modelador não invalida o trabalho de configuração, assim como uma eventual troca de linguagem de configuração não invalida as implementações das funções de acesso ao modelador.

4.4 Fluxo de dados

A Fig. 4.5 ilustra o fluxo de dados do sistema. O aspecto unidiredional da comunicação entre os serviços da aplicação e a parte configurável permite ao modelador não armazenar dados nem ter nenhum conhecimento referente aos atributos associados às entidades geométricas. Todo o gerenciamento dos atributos e a associação destes às entidades geométricas são responsabilidades da parte configurável. A arquitetura proposta evidencia uma clara separação entre os serviços da aplicação e a parte configurável [Cowan et al. 1993].

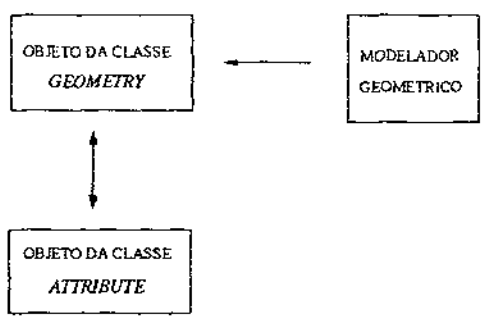

Figura 4.5: Fluxo de dados entre a parte fixa e a parte configurável do sistema.

4.5 Níveis de abstração

O desenvolvimento de uma aplicação configurável deve ser feito sob o enfoque da abstração. O conceito de abstração está estritamente relacionado com o grau de dificuldade na utilização de uma aplicação [Krueger 1992]. Pretende-se, com a arquitetura proposta, tornar a tarefa de especificação de atributos para simulação de mecânica computacional a mais simples possível. A tecnologia da aplicação se refere a modeladores geométricos e geradores de malha de elementos finitos ou de contorno. Os módulos responsáveis por esses serviços são, em geral, bastante complexos, como por exemplo, o código do FRANC3D (Capítulo 2). O acesso a essa tecnologia deve ser feito com alto grau de abstração, pois caso contrário recai-se no problema de acessar diretamente esse código, que representa uma tarefa não trivial.

Nesse contexto, o desenvolvimento de uma aplicação configurável que utiliza a arquitetura proposta envolve a participação de quatro grupos diferentes, com crescente nível de abstração para a realização das tarefas: programadores dos serviços da aplicação, programadores da configuração, usuários avançados e usuários finais. Esses grupos diferem entre si de acordo com seus conhecimentos de programação e tarefas que devem desempenhar ao longo do desenvolvimento de uma aplicação (Fig. 4.6). O sistema proposto para comunicação entre a parte configurável e os serviços da aplicação representa o nível dos programadores da configuração. Portanto, no desenvolvimento de aplicações que utilizem esse sistema, não existem pessoas envolvidas nesse nível, pois o sistema já está implementado. Os profissionais envolvidos serão programadores dos serviços da aplicação, que desejam tornar suas aplicações configuráveis, usuários avançados, que desejam configurar os atributos da aplicação ou usuários finais que simplesmente utilizam a aplicação.

- Programadores dos serviços da aplicação são os responsáveis pela implementação da tecnologia da aplicação. Possuem um alto nível de conhecimento sobre estrutura de dados. São os responsáveis por escreverem as funções MIS. Têm acesso a base de dados do modelador geométrico.

- Programadores da configuração são os responsáveis pela implementação das funcionalidades referentes à configuração da aplicação. Devem estender a linguagem

de configuração, adicionando comandos que possibilitem o acesso à tecnologia da aplicação. São os responsáveis pela extensão da linguagem de configuração, com as funções que fazem a interface entre as *modeling classes* e as funções MIS (funções do *binding*). Para isso devem conhecer a *API* da camada de abstração implementada pelos programadores de aplicação (funções MIS). Não têm conhecimento sobre detalhes de implementação dessa tecnologia. Possuem conhecimento da linguagem de programação e da linguagem de configuração. Podem adicionar uma nova *core class*, aumentando, deste modo, a capacidade de configuração da aplicação. Têm acesso tanto às *core classes* quanto à *API* das funções MIS.

- Usuários avançados, ou usuários mestres, são especialistas na área relacionada com seu trabalho e podem configurar a aplicação para suas necessidades específicas. Devem conhecer detalhes das funcionalidades implementadas pelos programadores da configuração. Possuem conhecimento da linguagem de configuração. São os responsáveis pela configuração da aplicação, feita através da especificação de subclasses das *core classes*, ou seja, utilizando as funcionalidades implementadas pelo configurador da aplicação. Devem saber como criar uma nova classe. Para isto devem conhecer e entender as classes existentes. Têm acesso aos arquivos de configuração. A classe *Attribute* e suas subclasses formam uma base que simplifica a tarefa de especificação dos atributos. Novas classes são adicionadas, de acordo com as aplicações específicas. O usuário avançado não se preocupa com detalhes de implementação e se concentra somente nas tarefas que as aplicações precisa realizar.

- Usuários finais somente utilizam as funcionalidades oferecidas pelo sistema através de uma interface gráfica (botões, listas, etc.), não sendo requerido nenhum conhecimento sobre a linguagem de programação ou de configuração. Podem realizar tarefas como criar um atributo novo ou selecionar um atributo de uma lista existente para aplicar a uma determinada entidade geométrica. Podem também consultar as propriedades de um atributo ou consultar os atributos aplicados a uma determinada entidade. Deve-se ressaltar que criar um atributo novo significa criar uma nova instância de uma determinada classe de atributo. Esses usuários não têm acesso às funções que criam classes novas, ou seja, não é permitido a configuração do sistema; tudo funciona como se fosse parte do código do modelador. Devem

saber escolher o arquivo de configuração adequado às suas necessidades.

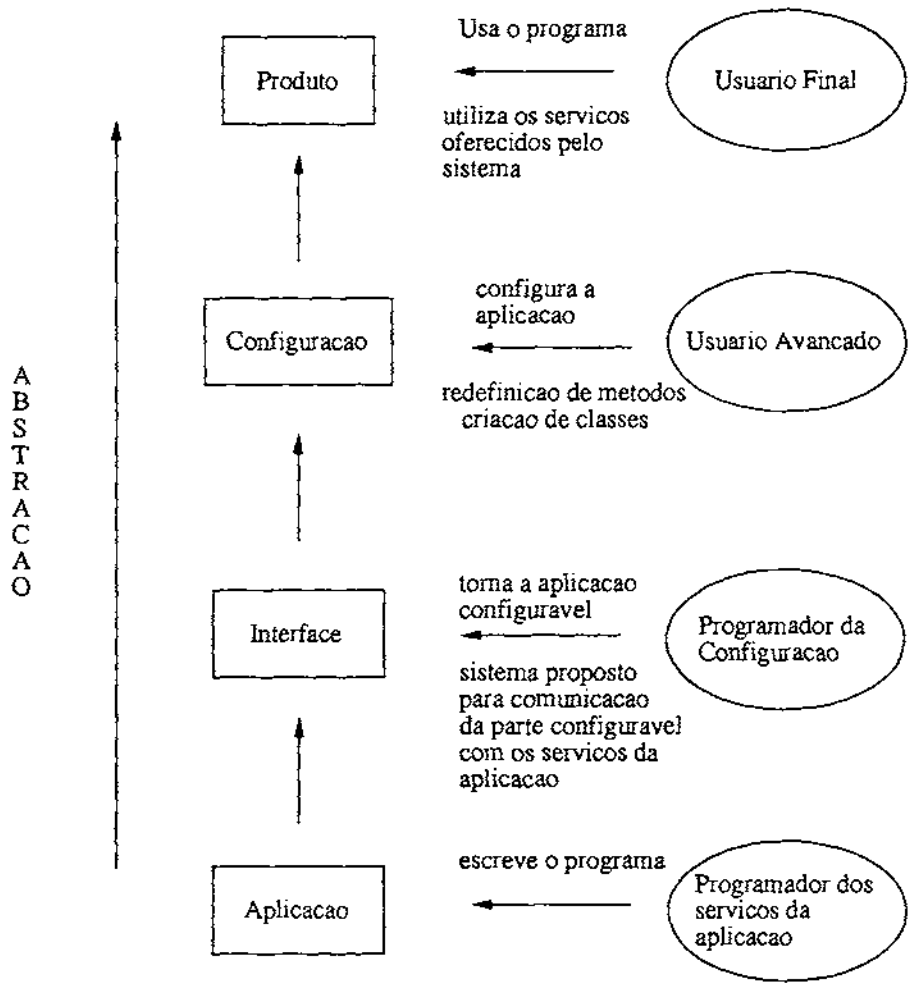

Figura 4.6: Níveis de abstração no processo de desenvolvimento de uma aplicação.

4.6 Construção automática das classes referentes às entidades dos modelos geométrico e numérico

Esta seção descreve o mecanismo para a criação automática das subclasses das classes *Geometry* e *Element*. Conforme explicado anteriormente, a associação entre os atributos e as entidades do modelo geométrico é feita através de instâncias de subclasses da classe *Geometry*, que representam a abstração das entidades geométricas do modelador. A construção dessas subclasses, no entanto, depende do modelador utilizado, pois os tipos de entidades geométricas são específicos para cada modelador. Desse modo, as subclasses

de *Geometry* não podem fazer parte da biblioteca básica das *core classes*, pois não fariam sentido no caso de se utilizar um modelador diferente. Essas classes também não podem ser construídas pelo usuário configurador, que não possui nenhum conhecimento sobre os tipos das entidades do modelador.

Por outro lado, o programador dos serviços da aplicação, que é quem possui esse conhecimento, não pode ser o responsável pela construção das classes, pois não tem acesso à parte configurável (linguagem de configuração, *core classes*, etc.). Analogamente, as subclasses da classe *Element*, que representam a abstração dos tipos de elementos da discretização, são dependentes do modelador.

Sendo assim, essas classes devem ser criadas automaticamente pelo sistema através de um mecanismo que mantenha a independência entre o modelador e a parte configurável. Este mecanismo é implementado acrescentando-se, ao conjunto das funções MIS, funções responsáveis por cadastrar os tipos de entidades do modelador. O sistema de configuração executa estas funções e, a partir das informações fornecidas pelo registro das entidades, cria automaticamente as correspondentes subclasses, que podem, então, ser acessadas pelo usuário configurador. Detalhes da implementação deste mecanismo serão explicados no próximo capítulo.

Capítulo 5

Um sistema extensível para gerenciamento de atributos

Este capítulo descreve um sistema que implementa a arquitetura proposta no capítulo anterior. O sistema, denominado ESAM (*Extensible System of Attribute Management*), consiste de um conjunto de funções que oferecem os serviços para configuração da aplicação, de uma coleção de classes que representa a biblioteca básica da parte configurável da aplicação (*core classes*) e de um conjunto de funções responsáveis pela interface entre as *core classes* e o modelador.

Os serviços do sistema são oferecidos através de um conjunto de rotinas que devem ser incorporadas à aplicação. A aplicação define sua interface para que os usuários tenham acesso a esses serviços (através de *callbacks* de eventos, por exemplo). Esses serviços representam a funcionalidade do sistema, como a possibilidade de especificação de um novo atributo ou a associação desse atributo a uma entidade geométrica. Os serviços, entretanto, não incorporam a semântica da aplicação. Este capítulo descreve algumas funcionalidades desses serviços. Uma descrição detalhada dos serviços é feita em [Silveira 1995].

As *core classes* representam uma abstração das entidades envolvidas no processo de geração de um modelo numérico, ou seja, as entidades geométricas e da malha (*modeling classes*) e os atributos de simulação (*attribute classes*).

As *modeling classes* permitem que o usuário defina métodos correspondentes a ações sobre as entidades geométricas, tais como a aplicação de um atributo a uma face ou a

validação de uma operação de subdivisão de uma aresta. Desse modo, o configurador da aplicação pode controlar o comportamento das entidades geométricas com um alto grau de abstração, pois não precisa acessar o código que implementa essas entidades no modelador.

As classes derivadas da classe *Attribute* formam uma biblioteca que representa uma base para um grande domínio de simulações. Essa biblioteca pode ser expandida através da construção de novas classes, derivadas das classes existentes, de acordo com a necessidade de cada simulação específica.

O acesso à hierarquia das classes existentes é feito através de uma ferramenta interativa que permite ao usuário percorrer toda a estruturação de classes do sistema, com suas respectivas variáveis e métodos. Essa ferramenta, denominada *Browser*, permite ainda a construção de novas classes derivadas de classes de atributos, ou a redefiniçâo de métodos de classes de geometria.

A Fig. 5.1 ilustra o sistema ESAM sendo utilizado em uma aplicação. O retângulo tracejado representa os componentes que fazem parte do sistema.

5.1 Serviços oferecidos

Os serviços oferecidos pelo ESAM são funções, implementadas em C, que incorporam o ambiente de configuração de atributos à aplicação e têm como objetivo permitir ao usuário final acessar às funcionalidades do ambiente de configuração, tais como especificar um atributo, aplicar um atributo a uma face, etc. Esses serviços podem ser acionados, por exemplo, através de *callbacks* de eventos de interface. Os serviços podem ser agrupados da seguinte maneira:

- **Serviços de inicialização e finalização.** Esses serviços são responsáveis por tarefas como inicializar as variáveis utilizadas pelo sistema, criar automaticamente as classes referentes às entidades geométricas (através das funções MIS de registro das entidades geométricas), criar as instâncias das classes *Model* e *Numerical Model*, entre outras. Deve ser criada uma instância para cada modelo geométrico ou numérico gerado pela aplicação.

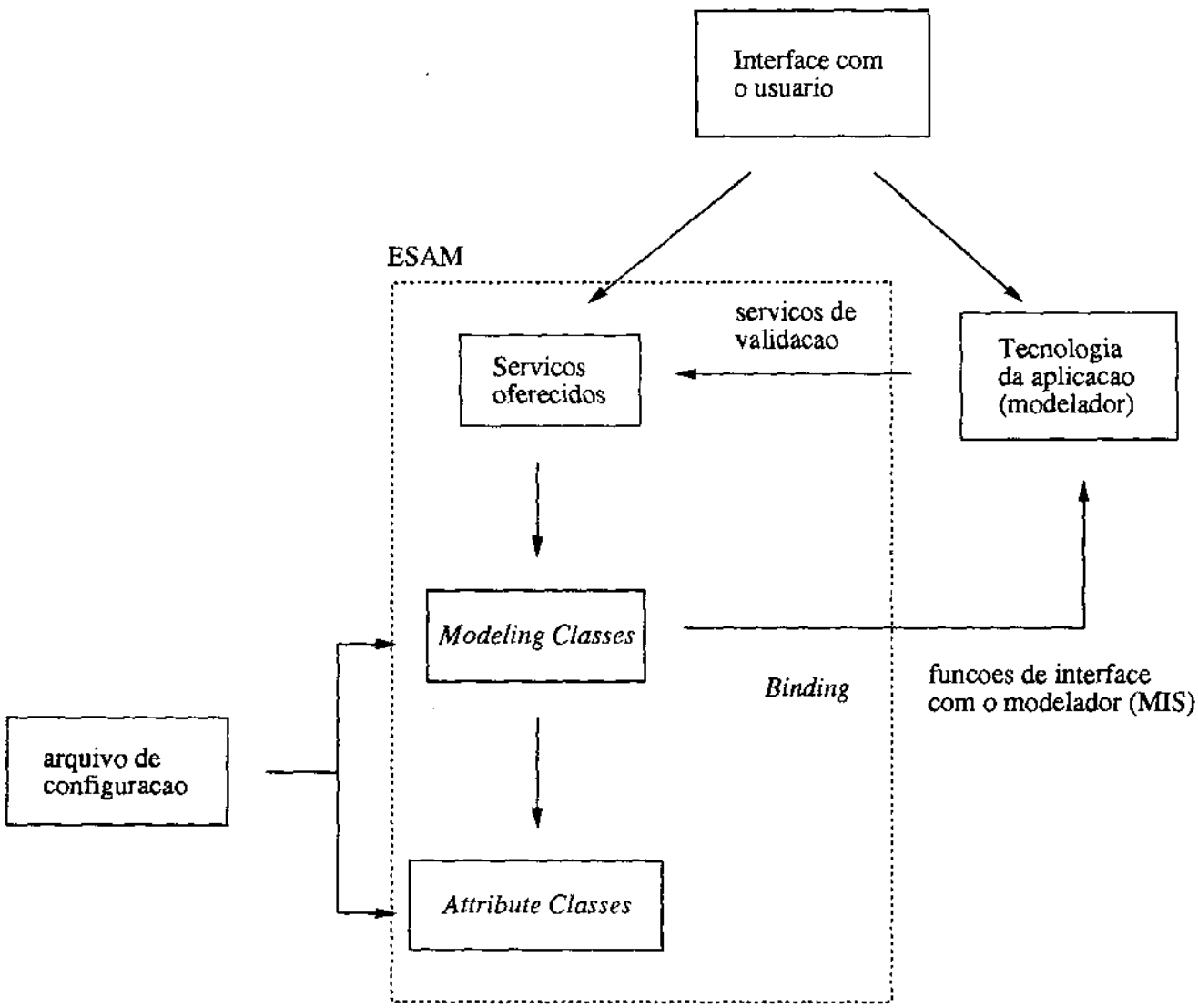

Figura 5.1: Aplicação que utiliza o ESAM.

- **Serviços de especificação.** Esses serviços são os responsáveis por tarefas como criar um tipo novo de atributo, modificar um tipo existente, especificar um atributo como o atributo corrente, aplicar um atributo a uma entidade geométrica ou consultar um atributo de uma entidade.

- **Serviços de validação.** Esses serviços são responsáveis pela validação, no que se refere ao gerenciamento de atributos, de ações sobre as entidades, como a operação de juntar ou subdividir entidades.

- **Serviços de entrada e saída.** Esses serviços são os responsáveis por tarefas como carregar o arquivo de configuração desejado, armazenar um determinado modelo e recuperá-lo posteriormente (mecanismo de persistência) ou salvar o modelo numérico em um formato específico (exportação).

- **Serviços de cliente.** Esses serviços são chamados pela aplicação sempre que uma determinada ação sobre uma entidade geométrica for efetuada, como por exemplo, a criação de uma aresta.

5.2 Mecanismo para a construção das classes de geometria

Conforme mencionado no capítulo anterior, as classes referentes às entidades geométricas do modelador são criadas automaticamente pelo sistema. Isso é feito através de uma função MIS (*Modeling Interface Specification*) de inicialização, chamada por um serviço de inicialização do sistema. Essa função, escrita pelo programador do modelador, registra cada tipo de entidade geométrica e de malha do modelador, fornecendo o nome (vai ser o nome da classe criada pelo sistema) dado por uma cadeia de caracteres, a dimensão geométrica (representa a superclasse da classe criada) e o identificador da entidade no contexto do modelador. A partir dessas informações, o sistema constrói a classe com o nome fornecido e armazena a correlação entre a classe criada e o identificador da entidade. Desse modo, toda vez que o sistema requisitar informações sobre uma entidade do modelador, o identificador da entidade é fornecido para o sistema, que pode determinar, então, a classe da entidade consultada no contexto do ambiente de configuração. O mesmo mecanismo é utilizado para a construção das classes referentes

aos tipos de elementos da discretização. Nesse caso, a função de inicialização registra os tipos de elementos fornecendo o nome, o número de nós do elemento e o grau do polinômio de interpolação do elemento (1 - linear, 2 - quadrático, etc.). Um exemplo do código C da função de inicialização para registrar entidades geométricas e' mostrado abaixo (exemplos de utilização dessas funções são mostradas na próxima seção):

```
void RegGeomEntities ( void (*reg_ent)(char *, int, int) )
{
 ...

 reg_ent ("nome", id, dim);   /* funcao do sistema que cria automaticamente
                                 uma subclasse da classe "dim". */

 ...
}
```

5.3 Descrição das modeling classes

5.3.1 Classe Model

Cada instância da classe *Model* representa um modelo geométrico específico. Esta classe contém as informações existentes em um modelo geométrico, ou seja, geometria e atributos. Um objeto desta classe contém uma lista de objetos da classe *Geometry* e uma lista de objetos da classe *Attribute*. Os métodos associados a esta classe implementam os serviços oferecidos pelo sistema, referentes ao modelo geométrico (serviços de especificação, de validação e de persistência, por exemplo) descritos anteriormente.

Suponha, por exemplo, que o serviço para criação de um atributo tenha sido associado (como *callback*) a um botão de uma determinada aplicação. Ao ser acionado pelo usuário, esse botão dispara o serviço, que por sua vez, chama o método da classe *Model* para criar um atributo, que finalmente dispara o diálogo de interface mostrado na (Fig. 5.2).

5.3.2 Classe Geometry

Esta classe implementa a abstração das entidades geométricas para o sistema de configuração de atributos. A dimensão das entidades de um modelo geométrico são: 0 para um ponto, 1 para uma curva, 2 para uma superfície e 3 para um volume. Desse modo,

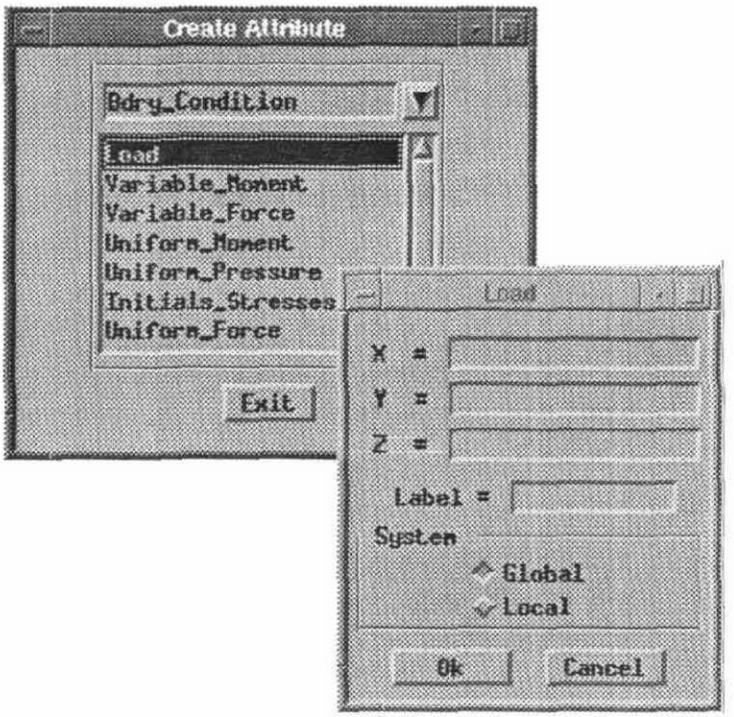

Figura 5.2: Diálogo disparado pelo método para criação de um atributo.

existem subclasses de *Geometry* referentes a cada dimensão.

A associação entre as entidades geométricas e os atributos é feita através de classes referentes a essas entidades, criadas automaticamente pelo sistema, conforme explicado anteriormente. Essas classes são específicas para cada modelador. Considere, por exemplo, que existem as entidades vértice, aresta e região em um determinado modelador bidimensional, e que os identificadores dessas entidades, no contexto do modelador, são respectivamente 5, 4 e 1. Nesse caso, a função de inicialização escrita pelo programador da aplicação consiste de:

```
void RegGeomEntities (void (*reg_ent)(char *, int, int) )
{
 reg_ent ("Vertex", 5, 0 ); /* cria uma classe vertex, subclasse da classe 0D
                               e relaciona o numero 5 com a classe vertex */

 reg_ent ("Edge", 4, 1 );   /* cria uma classe edge, subclasse da classe 1D
                               e relaciona o numero 4 com a classe edge */

 reg_ent ("Region", 1, 2 ); /* cria uma classe region, subclasse da classe 2D
                               e relaciona o numero 1 com a classe region */
}
```

A estruturação das *geometry classes*, nesse caso, é mostrada na Fig. 5.3:

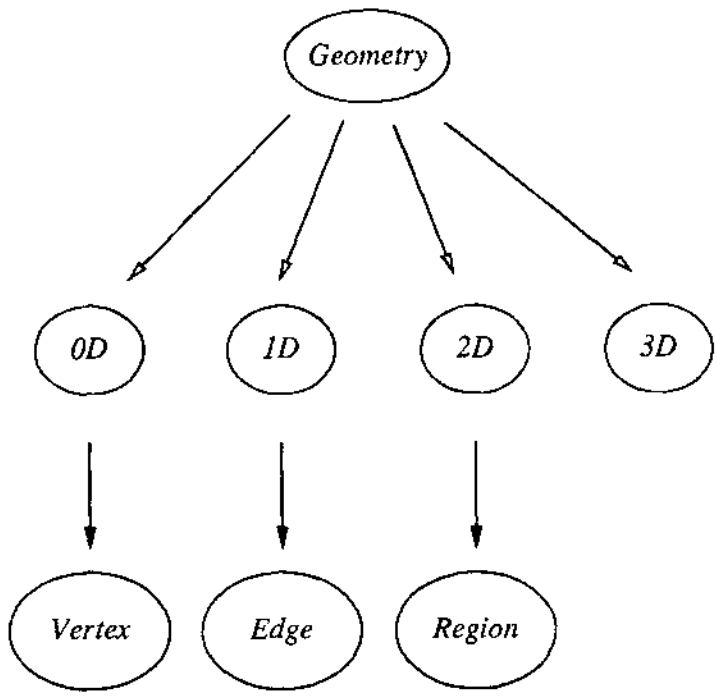

Figura 5.3: Exemplo de organização das *geometry classes*.

A classe *Geometry* implementa métodos para a manipulação da lista de atributos da entidade geométrica, como, por exemplo, adicionar e retirar um atributo dessa lista, métodos para aplicação de atributos na entidade geométrica ou para consulta dos atributos da entidade geométrica.

Existem, ainda, métodos para validação de algumas operações sobre as entidades geométricas, como por exemplo, métodos para validar a junção de entidades geométricas (duas faces podem representar regiões com materiais distintos que não podem ser combinados).

Esses métodos possuem uma implementação padrão (*default*) e são herdados por todas as subclasses de *Geometry*. Por exemplo, a implementação padrão do método *Attach* simplesmente adiciona o atributo corrente na lista de atributos do objeto em questão. Entretanto, o usuário configurador pode redefinir o método da subclasse *Vertex* para que, somente adicione o atributo caso este não tenha sido previamente colocado, ou seja, antes de adicionar o atributo na lista, verifique se já existe algum atributo do mesmo tipo associado ao vértice selecionado.

5.3.3 Classe Numerical Model

Esta classe representa uma abstração, no contexto do ambiente de configuração, do modelo numérico utilizado na simulação. Uma instância desta classe é criada para cada análise a ser efetuada e contém as informações necessárias para complementar a especificação do modelo numérico, ou seja, contém referências às entidades da discretização e um meio de acessar os atributos associados à essas entidades.

Esta classe implementa os métodos para acesso aos atributos aplicados sobre o modelo numérico. Esses métodos retornam listas de todos os nós ou elementos que possuem um determinado tipo de atributo ou listas com todos os atributos de um determinado tipo aplicados sobre nós ou elementos. Esses métodos são usados, por exemplo, quando se deseja obter a lista de todos os elementos de uma malha que possuem material isotrópico ou todos as cargas aplicada em nós.

5.3.4 Classe Node

Os objetos desta classe contêm referência para o objeto da classe *Geometry* correspondente à entidade geométrica à qual o nó da discretização está associado.

Esta classe implementa métodos para se obter as informações necessárias para a criação do modelo numérico para uma análise numérica (através de um arquivo, por exemplo). Para isso, existem métodos para capturar as coordenadas de um nó ou acessar os atributos aplicados no nó. Esse método de acesso aos atributos retorna uma lista com todos os atributos de um determinado tipo aplicado no nó.

5.3.5 Classe Element

A subclasses desta classe se referem aos tipos de elementos existentes no modelador e são criadas automaticamente pelo sistema, conforme explicado anteriormente. Essas subclasses são específicas para cada modelador. Considere, por exemplo, que existem os tipos Q4, Q8, T3 e T6. Nesse caso, a função de inicialização que deve ser escrita pelo programador da aplicação é:

```
void RegElemTypes (void (*register_elem)(char *, int, int) )
{
 reg_elem ("Q4", 4, 1);  /* cria a classe Q4 ( 4 n\'os, interpola\c c\~ao linear) */
 reg_elem ("Q8", 4, 2);  /* cria a classe Q8 ( 4 n\'os, interpola\c c\~ao quadr\'atica) */
 reg_elem ("T3", 3, 1);  /* cria a classe T3 ( 3 n\'os, interpola\c c\~ao linear) */
 reg_elem ("T6", 3, 2);  /* cria a classe T6 ( 3 n\'os, interpola\c c\~ao quadr\'atica) */
}
```

As informações sobre o número de nós do elemento e o grau do polinomio de interpolação são utilizados internamente pelo sistema. A estruturação das *Element classes*, nesse caso, é mostrada na Fig. 5.4:

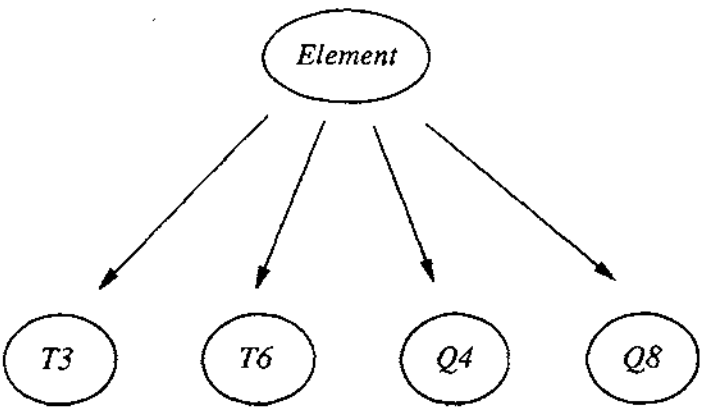

Figura 5.4: Exemplo de organização das *element Classes*.

A classe *Element* também implementa métodos para se obter as informações necessárias para a criação do modelo numérico para uma análise numérica. Exemplos são métodos para obter a conectividade de um elemento ou acessar os atributos aplicados no elemento ou em parte do elemento.

O método que fornece a conectividade do elemento assume uma numeração sequencial no sentido anti-horário, fornecida pelo modelador, através de uma função MIS. Esse método, no entanto, pode ser redefinido pelo usuário configurador, nas subclasses de *Element*, como por exemplo fornecer a conectividade do elemento no sentido horário.

Os métodos para acesso aos atributos retornam uma lista com todos os atributos de um determinado tipo aplicado no elemento ou com todas as partes do elemento que possuem um determinado tipo de atributo. Esses métodos são usados, por exemplo, para se retornar todos os lados de elementos que possuem uma carga distribuída.

5.3.6 Classe Element Feature

Esta classe existe para facilitar a identificação de atributos aplicados a partes de elementos. Desse modo, os métodos são referentes a informações de atributos, como métodos que retornam a conectividade da parte do elemento que contém um atributo ou métodos que retornam a lista de atributos de um determinado tipo aplicados à parte do elemento. Esses métodos são usados, por exemplo, para se obter uma carga distribuída aplicada em um lado do elemento.

5.4 Descrição das attribute classes

A classe *Attribute* representa a superclasse dos atributos de simulação necessários à complementação do modelo geométrico. Subclasses derivadas dessa classe devem ser criadas para atender aos requisitos específicos de cada simulação. Nesta versão, foram concebidas classes que definem os objetos comuns a um grande domínio de simulações. Essas classes representam uma base a partir da qual as classes específicas para cada tipo de problema devem ser construídas.

Atributos podem ser organizados em 5 categorias: propriedades de material, propriedades geométricas, parâmetros para controle de análise, parâmetros para controle de discretização e condições de contorno. Propriedades de material são propriedades físicas inerentes do material que está sendo analisado (constantes clásticas, por exemplo). Propriedades geométricas consistem de informações geométricas sobre a idealização do modelo que não podem ser obtidas do modelador. Em uma idealização de cascas, a espessura é uma propriedade geométrica, já que no modelador a casca é armazenada como uma face com uma descrição de superfície associada – por definição a descrição de uma superfície não menciona espessura. Parâmetros para controle de análise controlam a criação do modelo numérico para a análise, como por exemplo, definindo se a análise é linear ou não linear, se é de elementos finitos ou de contorno, e definindo os tipos de elementos utilizados. Parâmetros para controle de discretização controlam os algoritmos de geração de malha. A densidade da malha, transição e topologia da malha (elementos triangulares ou quadrilaterais) são alguns exemplos. Exemplos de condições de contorno são restrições a movimento e cargas aplicadas, como deslocamentos prescritos e forças de

superfícies. A Fig. 5.5 ilustra a estruturação da hierarquia da classe *Attribute*.

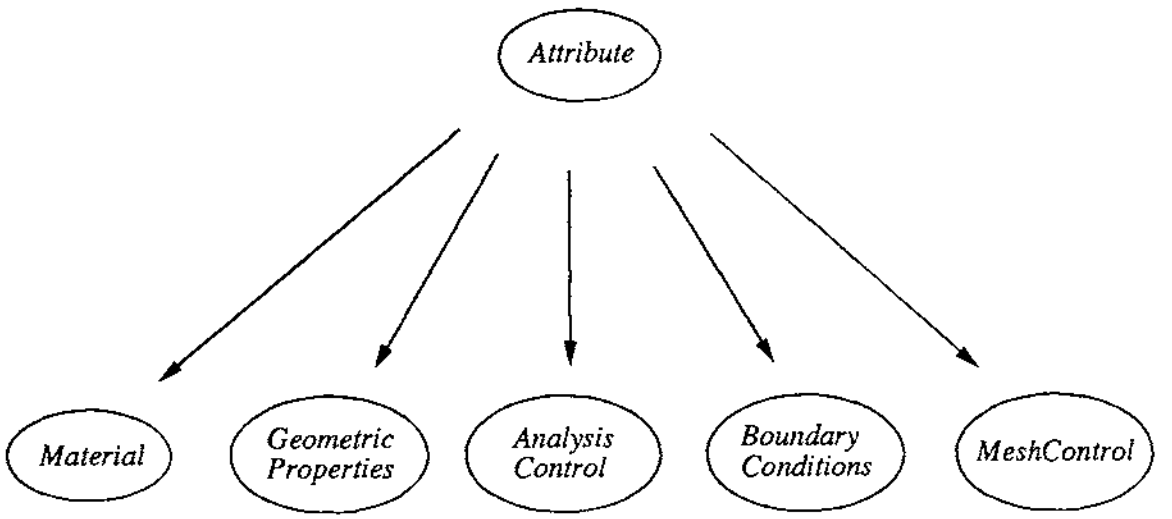

Figura 5.5: Organização das *attribute classes*.

As instâncias dessas classes contêm as propriedades pertinentes para o seu tipo e também uma lista de entidades geométricas (objetos da classe *Geometry*) nas quais estão associados. As instâncias têm associadas, ainda, um rótulo (*label*), para servir de referência para o usuário, implementado através de uma cadeia de caracteres. O *label* de cada instância pode ser definido explicitamente pelo usuário ou pode ser implicitamente construído pelo sistema.

Qualquer método específico a um determinado atributo pode ser criado pelo usuário configurador. Existem métodos, no entanto, que devem ser escritos para cada novo tipo criado. Esses métodos incluem: um método para a captura dos dados que definem o atributo (especifica o diálogo para a definição do atributo), um método para validação dos dados especificados e métodos para transportar os valores do ambiente de configuração para a interface gráfica e vice-versa. Esses métodos serão ilustrados na próxima seção.

5.4.1 Criação de um novo tipo de atributo

Esta seção mostra um exemplo de criação de um novo tipo de material, com os métodos mencionados na seção anterior. A nova classe (subclasse de *Material*) se refere a um material isotrópico, que possui como parâmetros as constantes referentes ao módulo de elasticidade e ao coeficiente de *poisson*. A sintaxe mostrada abaixo é a sintaxe da

linguagem Lua [Figueiredo et al 1994, Ierusalimschy et al 1995] e do *toolkit* EDG [Celes 1994]. As variáveis E e NU correspondem a parâmetros do diálogo utilizado para a captura dos dados do objeto. A variável self corresponde ao objeto corrente. A Fig. 5.6 mostra o diálogo construído pelo método CrtDlg.

```
ISOTROPIC  = crtclass { name = "Isotropic", parent = MATERIAL,
                    vars = { "e", "nu" } }

function ISOTROPIC:CrtDlg()   -- metodo que constroi o dialogo para
                              -- capturar os parametros do atributo.

  E  =  field { prompt=" E  = ",attributes='SIZE=75x12'  }
  NU =  field { prompt=" NU = ",attributes='SIZE=75x12'  }
end

function ISOTROPIC:Validate() -- metodo para validacao dos parametros
                              -- capturados.
  if !isNumber(E.nvalue)      or
     E.nvalue < 0             or
     !isNumber(NU.nvalue)     or
     NU.nvalue < 0            or
  then
     return nil
  end
  return 1
end

function ISOTROPIC:ValueToDlg() -- metodo que coloca nos campos do dialogo
                                -- os valores das variaveis do objeto.
  E:set(self.e)
  NU:set(self.nu)
end

function ISOTROPIC:ValueFromDlg() -- metodo que transporta os valores fornecidos
                                  -- pelo usuario para as variaveis do objeto.
  self.e = E.nvalue
  self.nu = NU.nvalue
end
```

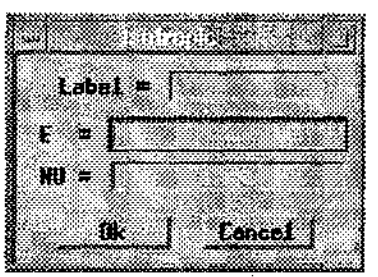

Figura 5.6: Diálogo para capturar os parâmetros do objeto da classe Isotrópico.

5.5 Browser

O *Browser* do ESAM serve para a configuração da aplicação, ou seja, para a criação de novas classes ou redefinição de classes ou métodos existentes, de acordo com as necessidades específicas. O *Browser* é uma ferramenta de navegação que permite o usuário percorrer a hierarquia das classes do sistema. Através dessa ferramenta, pode-se obter diversos tipos de informações sobre as estruturas das classes disponíveis e de diversas formas diferentes. O *Browser* permite explorar uma classe desde a sua definição (nomes das variáveis e dos métodos que compõem a classe) até a implementação de cada um de seus métodos. É possível, ainda, se obter informações sobre como as classes se relacionam entre si.

Browsers são importantes para o re-uso de componentes. O *browser* permite que o usuário verifique o que existe disponível em uma biblioteca. Se não existir uma classe que possua todas as necessidades requisitadas, podem existir partes de classes (métodos e variáveis) que juntas forneçam a solução procurada. Ou ainda, pode-se procurar classes que se aproximam das características desejadas e estender essas classes, i.é. criar subclasses, para atender ao comportamento esperado.

Conforme explicado anteriormente, existem duas hierarquias no sistema: uma das classes referentes às entidades do modelador (*modeling classes*) e uma das classes referentes aos atributos (*attribute classes*).

O *Browser* trabalha com o conceito de objeto selecionado corrente e possui três colunas. Cada coluna contém uma lista com classes ou métodos sobre a qual podem ser efetuadas algumas ações. As ações referem-se sempre ao item da lista que está selecionado: Na primeira coluna, define-se qual das duas hierarquias se deseja acessar (*Modeling ou Attribute*). Nesse instante, são mostradas as subclasses imediatamente abaixo da classe escolhida. Essas subclasses não podem ser modificadas pelo configurador. A ação disponível nessa coluna consiste na criação de uma subclasse da classe selecionada (botão *Add Class*). A segunda coluna contém as listas de subclasses da classe selecionadas na primeira coluna. A subclasse criada é adicionada nessa lista da segunda coluna. As ações oferecidas por essa coluna consistem em se criar uma subclasse (botão *AddSubclass*) ou se modificar um método de uma classe selecionada (botão *ModifyMethod*). A terceira

coluna contém a lista de métodos da classe selecionada na segunda coluna. Os métodos que contêm uma marca correspondem aos métodos definidos na classes corrente. Os métodos que não possuem uma marca são métodos herdados das superclasses da classe corrente. Todas as listas são atualizadas automaticamente pelo *browser*.

As Fig. 5.8 e Fig. 5.7 ilustram as funcionalidades descritas acima. Deve-se ressaltar que, para o configurador da aplicação, fica transparente o fato de as subclasses de *Geometry* e *Element* serem construídas automaticamente pelo sistema, parecendo que existem no arquivo de configuração do mesmo modo que as subclasses de *Attribute*. Para essas classes construídas automaticamente, no entanto, não é permitido se criar subclasses, somente redefinir os seus métodos. Isso se deve ao fato dessas classes se referirem às entidades geométricas, não gerenciadas pelo configurador. Desse modo, seria inconsistente permitir o configurador criar subclasses dessas classes.

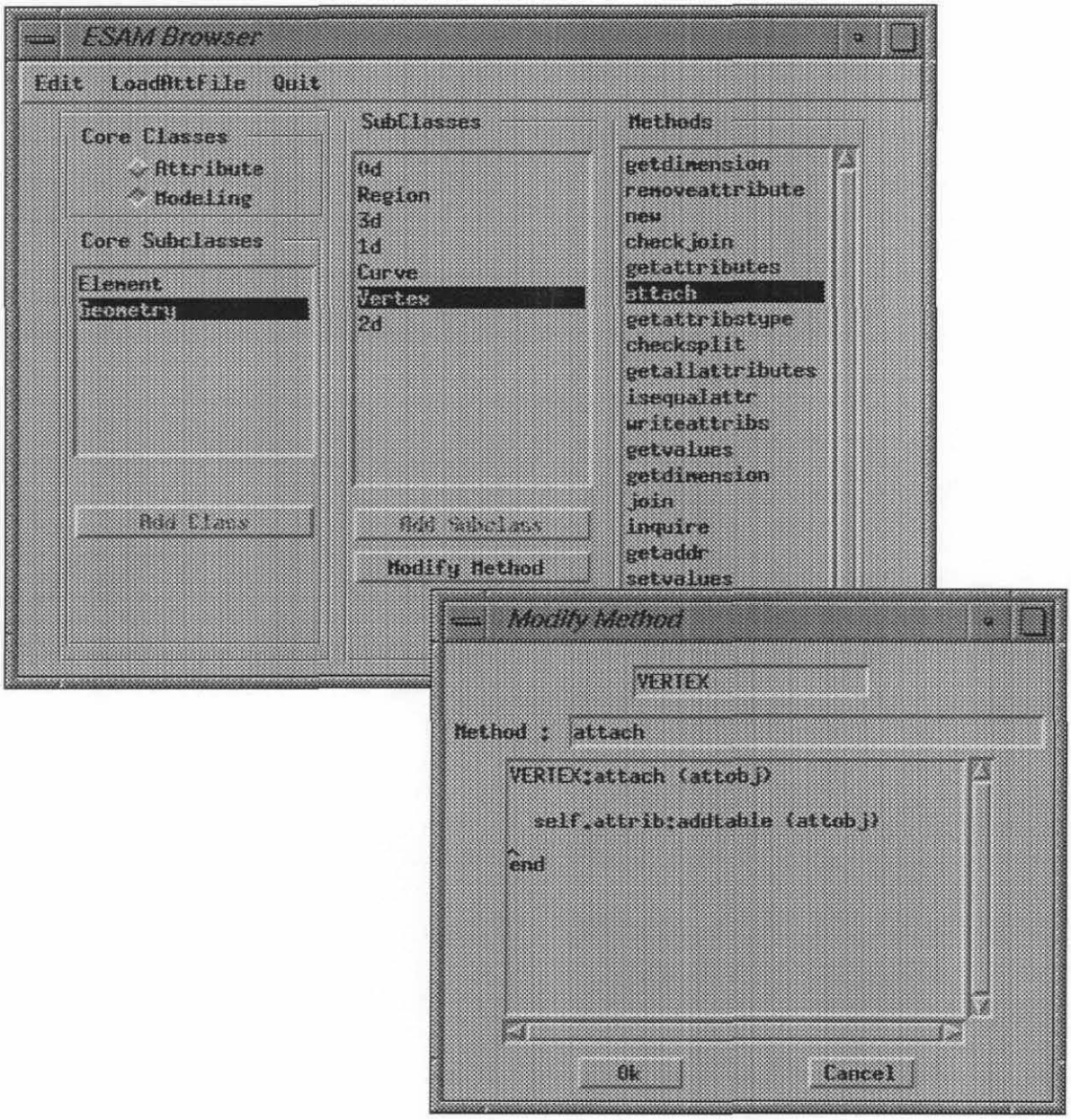

Figura 5.7: Browser: tarefa de redefinição do método *attach* da classe Vertex.

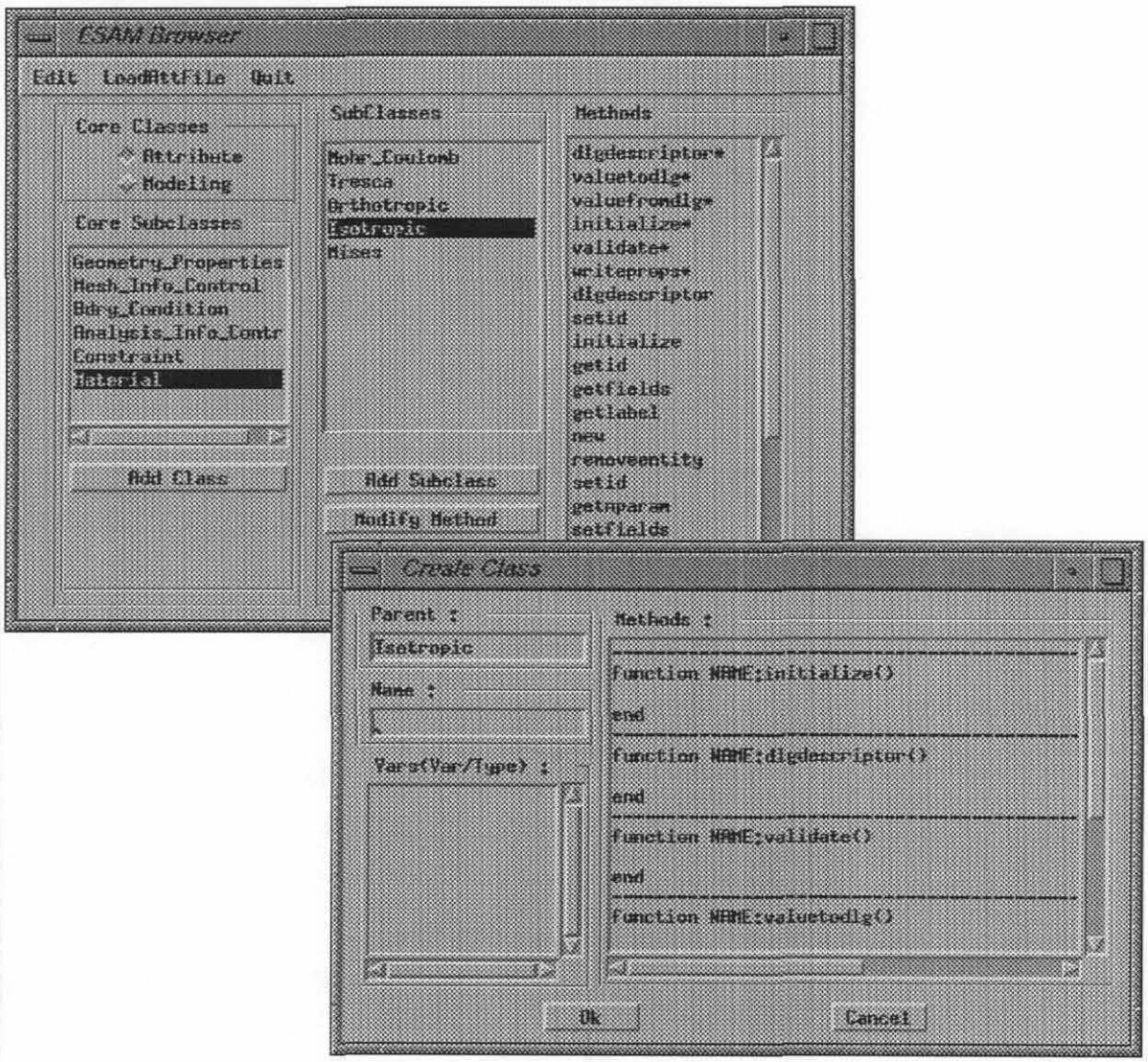

Figura 5.8: Browser: tarefa de criação de uma nova subclasse da classe *Isotropic*.

Capítulo 6

Implementação e utilização do sistema ESAM

Este capítulo descreve duas aplicações desenvolvidas com a arquitetura proposta, implementadas em dois ambientes diferentes. Uma aplicação consiste do programa FRANC3D [Potyondy-Ingraffea 1992; Martha 1989] e da versão do sistema ESAM que utiliza a linguagem Tcl e o *toolkit* Tk [Ousterhout 1994]. Uma outra aplicação consiste do modelador EDP [Cavalcanti 1992], do gerador de malhas AMVIEW [Cavalcante 1994] e da versão do sistema ESAM implementada com a linguagem Lua [Ierusalimschy *et al* 1994, Figueiredo *et al* 1994] e o *toolkit* EDG [Celes 1994].

Apesar de bastante diferentes, ambas linguagens de configuração (Tcl e Lua) oferecem algumas funcionalidades básicas que são importantes para a implementação da arquitetura: ambas linguagens consistem de uma biblioteca em C que pode ser ligada a diversas aplicações. As aplicações podem acoplar códigos escritos na linguagem de configuração, permitindo prototipagem rápida e acesso programável pelo usuário à tecnologia da aplicação (serviços de modelagem).

Conforme mencionado no Capítulo 3, não existe o conceito de programa principal, ou seja, a linguagem depende de um programa hospedeiro escrito em C, que pode, por sua vez, chamar funções da linguagem de configuração para serem executadas, ler e atribuir variáveis no ambiente de configuração e ainda registrar novas funções de C para serem chamadas nesse ambiente. Além disso, as linguagens oferecem as construções fundamentais para definição de funções e controle de fluxo (*if statements, loops*, etc.). Os detalhes

de implementação específicos das linguagens de configuração não serão abordados.

O ambiente de configuração é todo baseado no paradigma de programação orientada a objetos, no qual o usuário configurador tem acesso a uma biblioteca de classes, podendo redefinir métodos das classes existentes ou criar classes novas. A configuração é feita em um arquivo escrito na linguagem de configuração.

6.1 Uma versão configurável do FRANC3D

Tornar o programa FRANC3D (descrito no Capítulo 2) configurável foi a motivação inicial desta tese, conforme mencionado na introdução desta tese, e desse modo esta foi a primeira aplicação desenvolvida com a arquitetura proposta.

Nessa aplicação, foi utilizada uma versão do sistema ESAM, implementada com a linguagem de configuração Tcl, o *toolkit* Tk e uma biblioteca que suporta programação orientada a objetos, denominada Odc, desenvolvida na Universidade de Cornell pelo grupo CFG (*Cornell Fracture Group*). Essa biblioteca consiste de uma extensão da linguagem C e é baseada nos conceitos apresentados em [Cox 1991]. A linguagem Tcl foi estendida com comandos para a criação de classes e envio de mensagens entre objetos. Esses comandos correspondem a funções que acessam as funções da biblioteca Odc, ou seja, fazem interface entre a linguagem Tcl e a biblioteca Odc. Desse modo, essa biblioteca Odc teve que ser estendida para suportar a criação de classes em tempo de execução. Devido ao fato de Tcl somente trabalhar com *strings* (cadeia de caracteres), um conjunto de funções foi desenvolvido para auxiliar na interface com a biblioteca Odc.

Deve-se observar que, no momento da implementação desta versão do sistema ESAM, não existia nenhuma extensão de Tcl, de domínio público, que suportasse a definição de classes, objetos e métodos. Recentemente, foi distribuída uma biblioteca denominada TclObject [IXILimited 1995]. O autor, no entanto, não possui conhecimento suficiente sobre esta biblioteca para afirmar que a teria utilizado, caso existisse no momento do desenvolvimento do ESAM.

6.2 Uma versão configurável do EDP e AMVIEW

De modo a utilizar o sistema ESAM no desenvolvimento de aplicações no TeCGraf (Grupo de Tecnologia em Computação Gráfica da PUC-Rio), foi decidido implementar o ESAM com a linguagem de configuração Lua e o *toolkit* EDG, a fim de manter a compatibilidade com as aplicações existentes no grupo.

Lua é uma linguagem de extensão desenvolvida no TeCGraf projetada para ser usada como linguagem de configuração . Além das funcionalidades básicas apresentadas no início deste capítulo, Lua oferece um poderoso mecanismo para descrição de objetos, através dos chamados construtores de tipo [Figueiredo et al 1994, Ierusalimschy et al 1995]. Lua suporta o paradigma de programação orientada a objetos, permitindo a definição de métodos e herança (simples e múltipla) entre objetos. Ainda, em Lua o programador não precisa se preocupar com o gerenciamento de memória. Com estruturas internas dinâmicas e coleta automática de "lixo", o programador usa livremente as diversas opções de estruturação de dados e deixa a cargo da própria linguagem as tarefas de re-alocação e liberação de memória.

Utilizando-se essa nova versão do sistema ESAM, foi desenvolvida uma aplicação configurável para simulação bidimensional adaptativa de mecânica computacional, utilizando-se o modelador EDP e o gerador de malhas AMVIEW. O EDP é um editor de subdivisões planares, baseado na estrutura de dados HED [Mäntylä 1988]. O AMVIEW é um gerador adaptativo, automático, de malhas planas, baseado em técnicas de subdivisão espacial recursiva (*quadtree e bitree*) [Cavalcante 1994].

Essa aplicação, implementada por Silveira (1995), é capaz de, a partir da definição geométrica do modelo, gerar automaticamente uma malha, construir o modelo numérico (malha mais atributos) e efetuar a análise, até a solução convergir para uma precisão especificada pelo usuário. Os atributos podem ser configurados para o tipo de análise desejado, através do sistema ESAM, que também é o responsável pelo mapeamento dos atributos da geometria para a malha de elementos finitos e pela criação do modelo (arquivo) para a análise (Fig. 6.1). Essa aplicação representa uma contribuição na direção da total automação do processo de simulação em mecânica computacional.

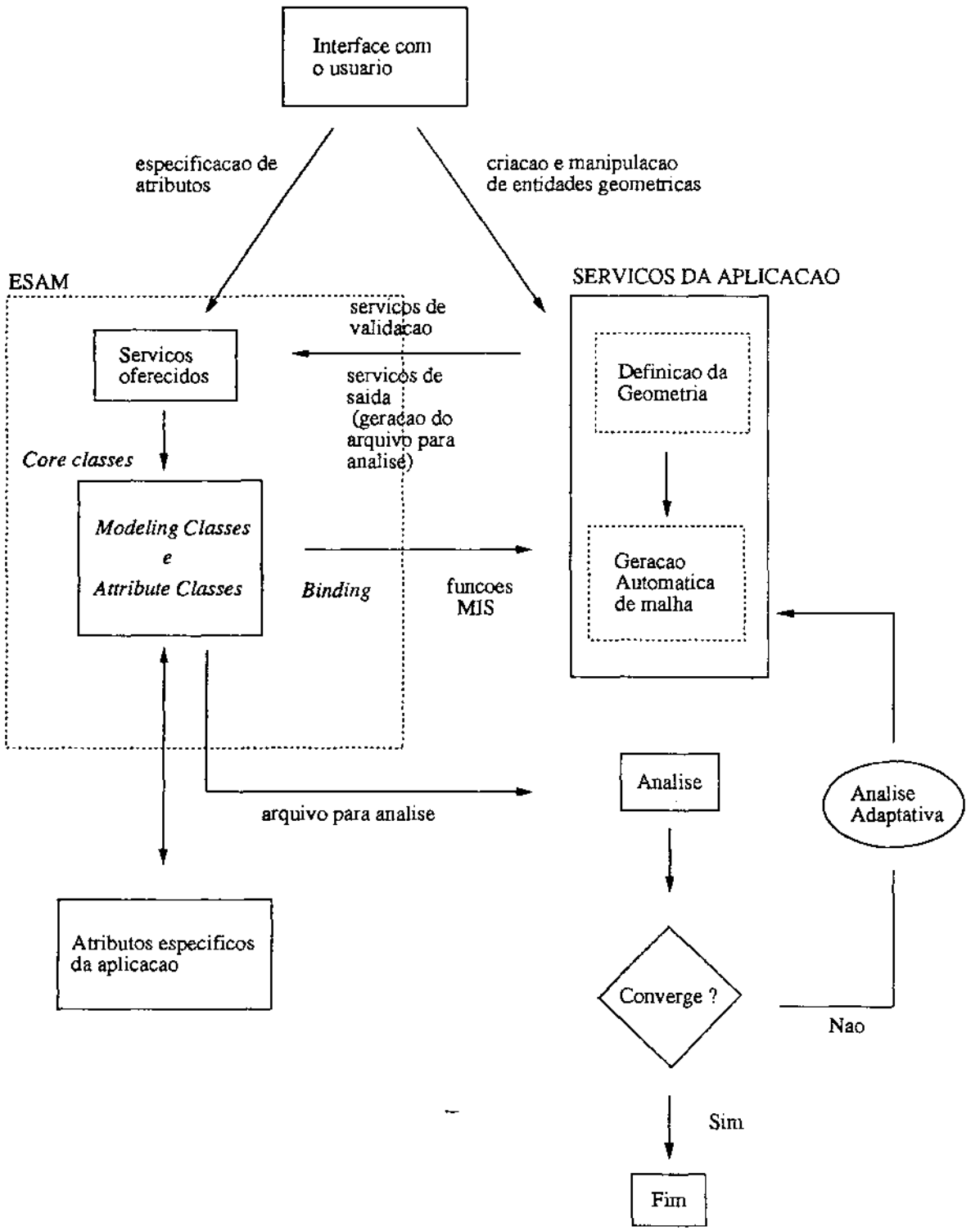

Figura 6.1: Utilização do sistema ESAM em uma aplicação para simulação adaptativa bidimensional de elementos finitos.

6.3 Duas experiências com a arquitetura proposta

As duas aplicações descritas acima serviram como uma forma de validação da arquitetura proposta, pois foram implementadas em dois ambientes completamente distintos. Ao mesmo tempo, isso serviu como plataformas de testes, auxiliando no desenvolvimento da arquitetura. Apesar de possuírem características distintas, ambos ambientes permitiram a implementação das funcionalidades da arquitetura.

O ambiente da linguagem Tcl necessitou de um esforço maior de implementação, devido ao fato de Tcl não oferecer suporte para programação orientada a objetos. No caso da linguagem Lua não foi preciso nenhuma biblioteca adicional, como no caso de Odc. A sintaxe de Tcl também não se mostrou tão fácil de ser assimilada e manipulada pelo usuário configurador da aplicação como foi no caso de Lua. O *toolkit* Tk, no entanto, oferece bem mais flexibilidade do que o *toolkit* EDG para a construção dos diálogos de interface com o usuário utilizados pelo ambiente de configuração e também para a construção dos diálogos dos atributos para a captura de suas propriedades.

Deve-se ressaltar, ainda, que o par Lua/EDG, ao contrário de Tcl/Odc/Tk, oferece uma portabilidade muito maior, permitindo a utilização do sistema ESAM com aplicações implementadas em diversas plataformas (Unix, DOS, MS-Windows, etc.).

6.4 Uma experiência de re-uso com o sistema ESAM

Conforme mencionado no Capítulo 4, a arquitetura proposta permite, devido a utilização das funções de interface (MIS), a implementação do sistema ESAM em aplicações existentes, mesmo que essas aplicações não tenham sido concebidas originalmete para serem configuráveis. Para isso é necessário, somente, que se escrevam essas funções MIS para a aplicação de interesse.

Nesse sentido, o sistema ESAM foi implementado no programa para geração de malhas de superfície MG [TeCGraf 1995], desenvolvido no TeCGraf. Esse programa não contém, em seu código original, nenhum suporte para tarefas referentes à especificação de atributos.

A implementação do ESAM no programa MG foi bastante simples. A tarefa de escrever

as funções de interface consumiu dois dias do programador do MG (que não precisou de nenhum conhecimento sobre a linguagem de configuração Lua ou sobre o *toolkit* EDG). Deve-se ressaltar, que para um usuário do programa, tudo funciona como se os serviços de especificação dos atributos, oferecidos pelo ESAM, fizessem parte do código original (Fig. 6.2).

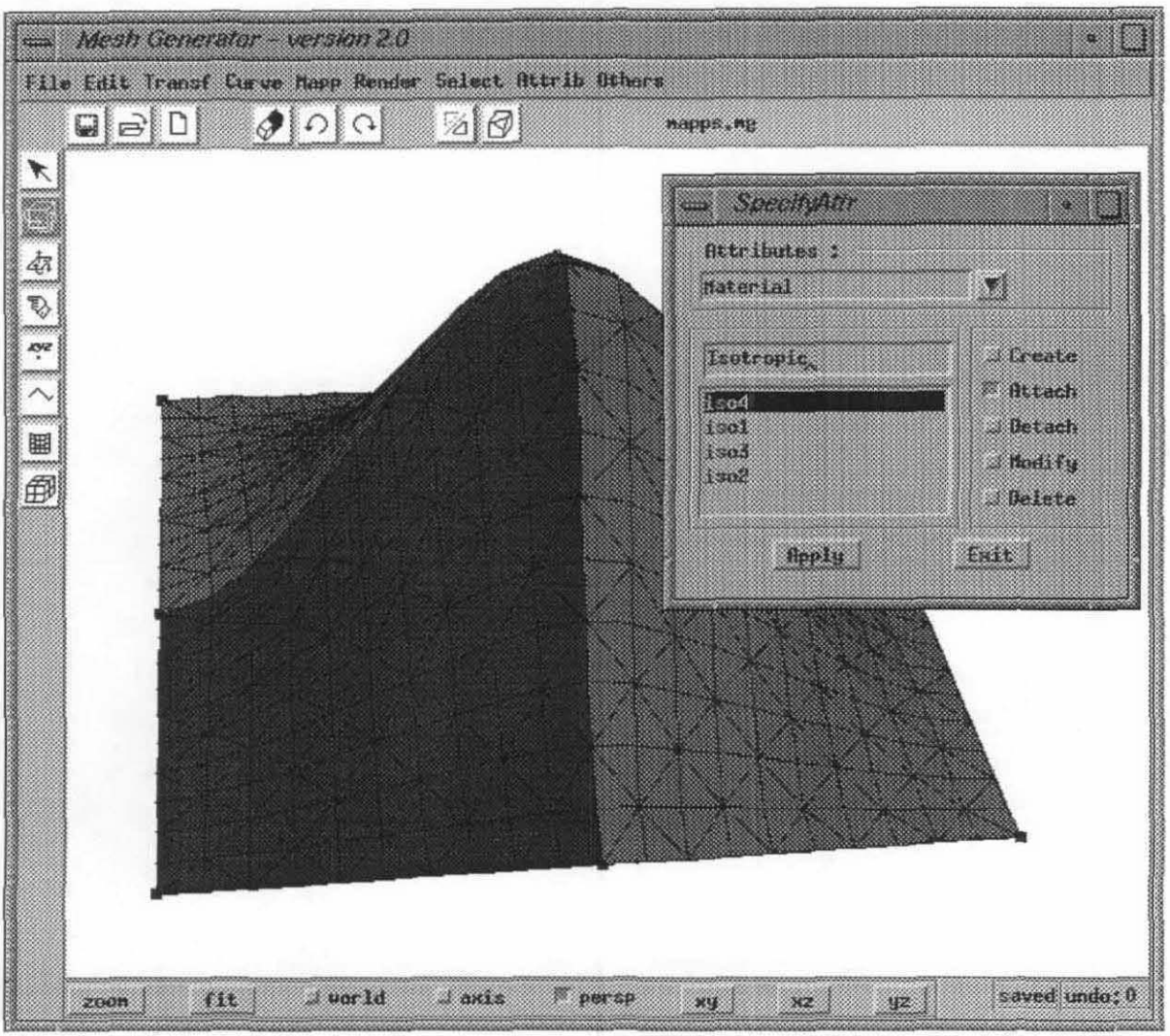

Figura 6.2: Ilustração da utilização do sistema ESAM para especificação de atributos no gerador de malhas MG.

6.5 Implementação do serviço para aplicação de um atributo a uma entidade geométrica

Esta seção mostra a implementação da funcionalidade referente à aplicação de um atributo a uma entidade geométrica no modelador EDP (mencionado anteriormente), nos diversos níveis de abstração do sistema ESAM. O atributo é aplicado a todas as entidades que estão selecionadas no momento que se dispara a função para realizar essa operação. Esse exemplo ilustra o tipo de tarefa requisitado em cada nível (é mostrado parte do código das funções e métodos (Linguagens C e Lua)).

- Nível do programador da aplicação: a informação requisitada pelo sistema ESAM ao modelador para aplicação de um atributo, consiste em saber qual é a lista de entidades geométricas selecionadas. Essa informação depende do modo como o modelador foi implementado e, portanto, é obtida através de uma função de interface MIS (código C), que deve ser escrita pelo programador da aplicação (programador do EDP, no caso).

```
void MisGetSelect( int *n, void ***ent, int **type)
{             /* retornar uma lista "ent" de entidades selecionadas.  */
 ...

 while( faces != NULL )          /* lista de faces selecionadas. */
 {
  fl = (Tface *) faces->f;
  (*ent)[cont]  = (void*)fl;     /* identificador da face corrente */
  (*type)[cont] = 5;             /* 5 corresponde ao tipo FACE */
  faces = faces->next;
  cont++;
 }

 ...                             /* o mesmo para vertices e arestas. */
}
```

- Nível do programador da configuração: esse nível corresponde à implementação do sistema ESAM, ou seja, as funções e métodos desse nível fazem parte da funcionalidade oferecida pelo sistema e, portanto, estão disponíveis para os clientes do sistema. As funções correspondentes aos serviços oferecidos pelo sistema são escritas em C e os métodos correspondentes às *core classes* são escritos em Lua.

```
void AttachAttr()          /* servico oferecido pelo sistema ( codigo C ). */
{
 ...

 method = lua_getfield(lo_model,"attach"); /* acessa metodo attach de Lua */
 lua_pushobject(lo_model); /* coloca objeto na pilha de Lua */
 lua_callfunction(method); /* dispara o metodo attach (de Lua) da classe Model */

 ...
}

function MODEL:attach()  -- metodo Attach da classe Model (codigo Lua).
  ...
  entlist = getentsel()  -- lista de entidades selecionadas.
                         -- (esse comando corresponde a funcao do binding
                         --  responsavel por chamar a funcao MIS GetEntSel).
  while entlist do
    ...
    classe = table[entlist.type] -- classe correspondente ao tipo da entidade
                                 -- selecionada (table[5] = FACE). Ver a
                                 -- funcao MisGetSelect acima.
    obj = classname:new()        -- cria o objeto, caso ainda nao exista.
    obj:attach(curattr)          -- metodo attach do objeto criado.
    curattr:append(obj)          -- adiciona objeto geometrico na lista
                                 -- de entidades do atributo corrente.
    ...
  end
  ...

end

function GEOMETRY:attach( attobj ) -- metodo attach da classe Geometry.

  self.attribs:addtable(attobj)    -- adiciona attobj na lista de atributos
                                   -- da entidade geometrica (a variavel self
end                                -- corresponde ao objeto corrente).
```

- Nível do usuário avançado: esse nível corresponde a configuração da aplicação, ou seja, nesse nível são criadas novas classes ou redefinidos métodos de classes existentes. Neste caso, o usuário avançado pode redefinir o método *attach* das subclasses de *Geometry*. No caso de *Vertex*, por exemplo, não faz sentido aplicar material.

```
function VERTEX:attach( attrobj )  -- redefinicao do metodo GEOMETRY:attach

 if not self.attribs:lista(attrobj) then  -- testa para ver se o atributo e do
                                          -- tipo material, antes de adiciona-lo
  self.attribs:addtable(attrobj)          -- na lista.
 end
end
```

- Nível do usuário final: esse nível corresponde aos usuários comuns da aplicação, que utilizam as funcionalidades do sistema. Neste caso o usuário seleciona na lista do diálogo apresentado, o atributo que ele deseja aplicar nas entidades selecionadas (Fig. 6.3).

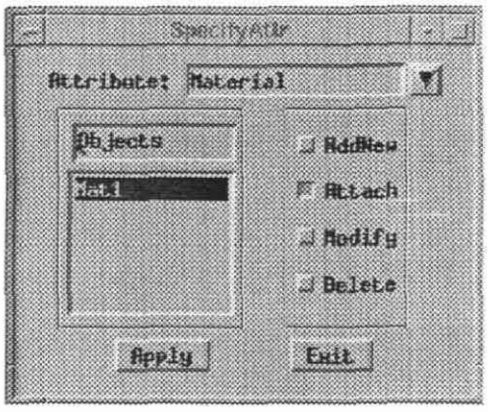

Figura 6.3: Diálogo de interface com o usuário disparado pelo sistema.

6.6 Ilustração da aplicação EDP/AMVIEW

Esta seção ilustra um exemplo da aplicação EDP/AMVIEW mencionada anteriormente. A intenção deste exemplo é mostrar a possibilidade de se utilizar a aplicação para diversos tipos de análise e a simplicidade para se especificar os atributos. Nesse sentido, são feitas duas análises distintas da mesma estrutura. Uma análise de tensões, linear-elástica, e uma análise de temperatura. Os atributos específicos de cada análise estão mostrados na Fig. 6.4. O modelador EDP gera a geometria (Fig. 6.5) a qual se aplica os atributos referentes a cada análise (Fig. 6.6). A malha para ambas análises é a mesma e é gerada pela aplicação (Fig. 6.7).

Pretende-se também, ilustrar o funcionamento de alguns métodos mencionados no capítulo anterior. Para isso, mostra-se um trecho do método (em lua) que escreve o arquivo neutro a ser utilizado como arquivo de entrada para a análise de tensões (Fig. 6.8 e Fig. 6.9). De modo a se ter um arquivo neutro simples, foi utilizado um exemplo bastante pequeno. A intenção do exemplo, no entanto, não é mostrar a capacidade de modelagem da aplicação, mas sim as funcionalidades do sistema ESAM.

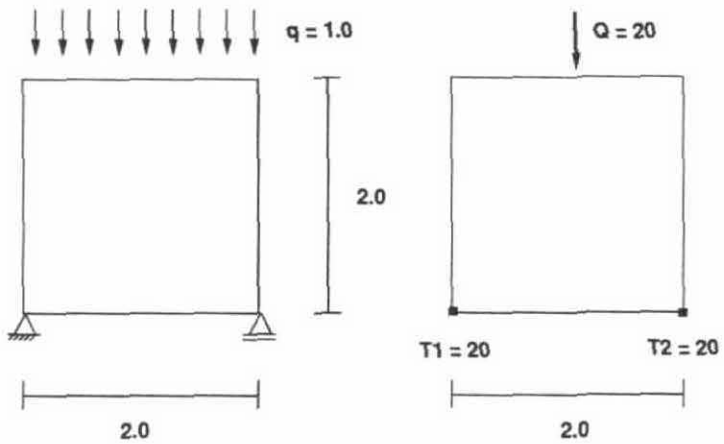

Figura 6.4: Ilustração da estrutura com os atributos referentes aos dois tipos de análise.

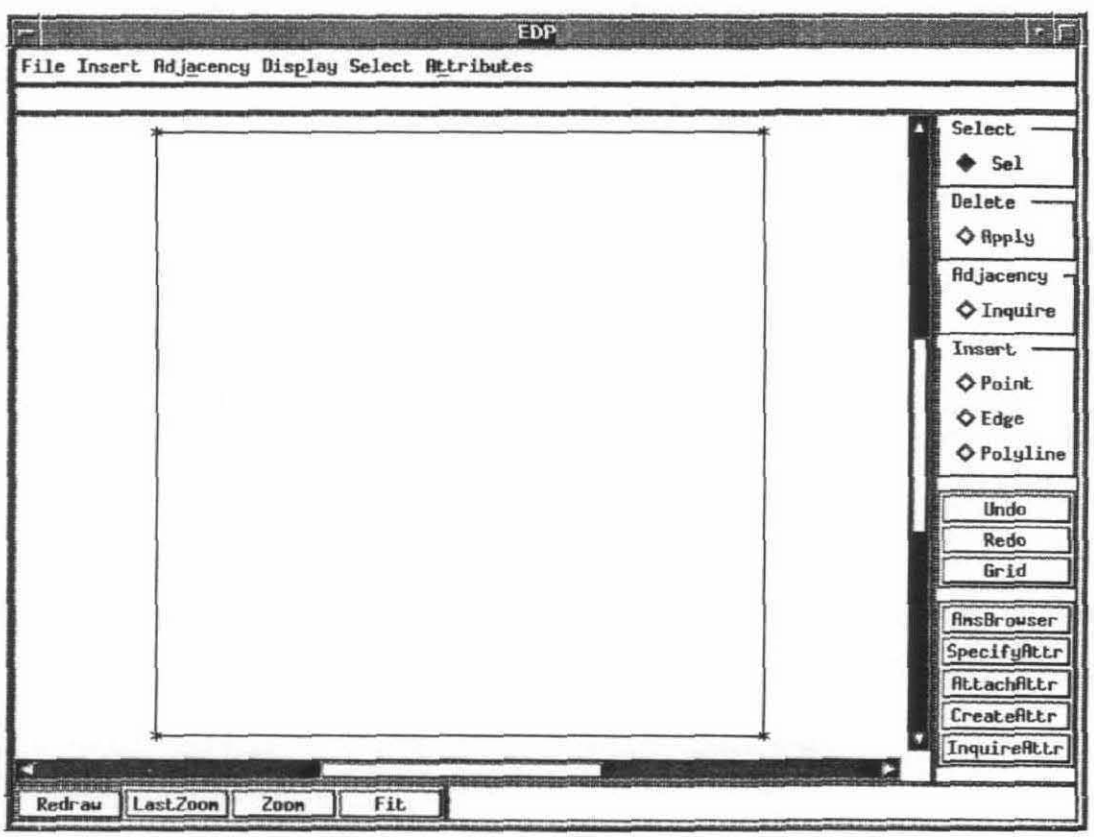

Figura 6.5: Geometria gerada no EDP.

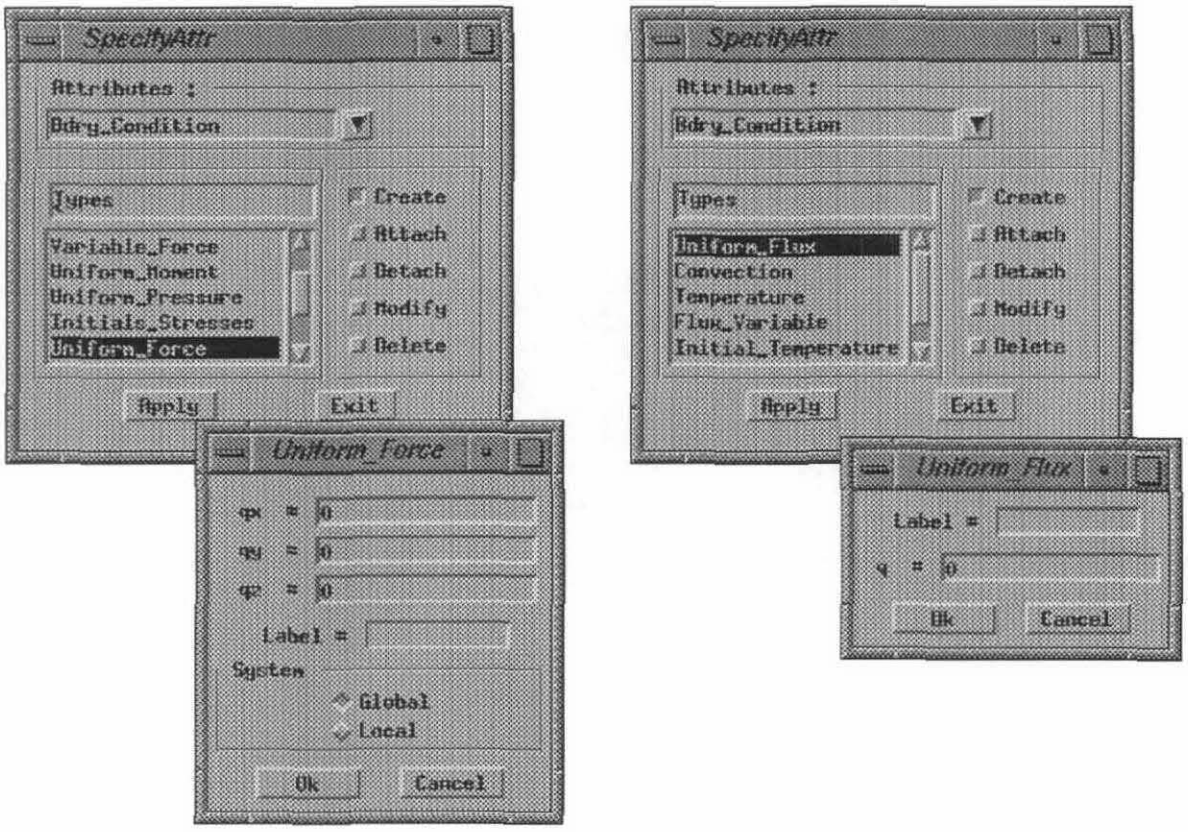

Figura 6.6: Diálogos para especificação dos atributos para os dois tipos de análise.

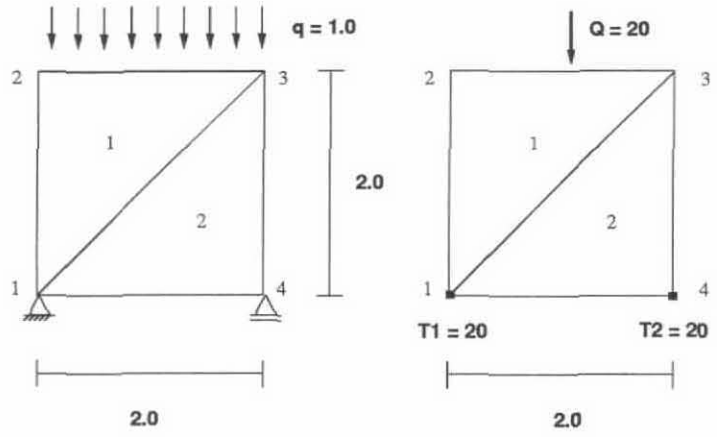

Figura 6.7: Malha gerada pelo sistema.

```
%HEADER
'File created by amview program'

%HEADER.ANALYSIS
'Plane_Stress'

%NODE
4

%NODE.COORD
4
1       0.000000        0.000000        0.000000
2       0.000000        2.000000        0.000000
3       2.000000        2.000000        0.000000
4       2.000000        0.000000        0.000000

%NODE.SUPPORT
2
1       1       1       0       0       0       0
4       0       1       0       0       0       0

%ELEMENT
2

%ELEMENT.T3
2
1       1       1       1       1       3       2
2       1       1       1       1       4       3

%LOAD
1
1  'Caso Unico'

%LOAD.CASE
1

%LOAD.CASE.LINE.FORCE.UNIFORM
1
1       3       2       0
 0.00   -1.00    0.00

%END
```

```
%HEADER
'File created by amview program'

%HEADER.ANALYSIS
'Temperature2d'

%NODE
4

%NODE.COORD
4
1       0.000000        0.000000        0.000000
2       0.000000        2.000000        0.000000
3       2.000000        2.000000        0.000000
4       2.000000        0.000000        0.000000

%ELEMENT
2

%ELEMENT.T3
2
1       1       1       1       1       3       2
2       1       1       1       1       4       3

%LOAD
1
1  'Caso Unico'

%LOAD.CASE
1

%LOAD.CASE.NODAL.TEMPERATURE
2
1   20.0
2   20.0

%LOAD.CASE.LINE.HEAT.FLUX.UNIFORM
1
1   2   3   20.0

%END
```

Figura 6.8: Arquivos neutros gerados para as duas análises.

```
function NUMMODEL:writeStructNF()

 ...
 write ( "%NODE\n" )
 write ( nodemap:getsize() )     -- numero total de nos do modelo numerico.
                                 -- nodemap contem a lista dos nos.

 write ( "%NODE.COORD\n" )         -- no caso de temperatura seria
                                   -- LOAD.CASE.NODAL.TEMPERATURE.
 write ( nodemap:getsize() )

 local n,v = next(nodemap,nil)
 while n do
   write ( v:getid() )       -- id do no corrente.
   coords = v:getcoords()    -- coordenadas do no corrente.
   write(coords.x)
   write(coords.y)
   write(coords.z)
  n,v = next(nodemap,n)
 end

 nodefixmap = getnodewithattribs("fixity")  -- lista de nos que possuem
                                               o atributo "fixity".
 write ( "%NODE.SUPPORT" )
 write ( nodefixmap:getsize() )             -- numero de nos com "fixity".

 local n,v = next(nodefixmap,nil)
 while n do
   write ( v:getid() )                      -- id do no corrente.
   fix = v:getattribstype("fixity")         -- atributo aplicado em um no.
                                            -- no caso de temperatura seria
                                            -- "temp" no lugar de "fixity".
   write ( fix )
   n,v = next(nodefixmap,n)
 end

 ...  -- idem ao NODE para imprimir os elementos
      -- idem de NODE.SUPPORT para LOAD.CASE.LINE.FORCE.UNIFORM.
      -- no caso de temperatura o label seria LOAD.CASE.LINE.HEAT.FLUX.UNIFORM.

end
```

Figura 6.9: Trecho do código que gera o arquivo neutro para análise de tensões.

Capítulo 7

Conclusão

Visando a automação do processo de simulação numérica, muito esforço tem sido concentrado na parte de modelagem geométrica, não existindo muita atenção para os aspectos referentes à especificação dos atributos. Esta tese trata desses aspectos.

A arquitetura proposta apresenta uma estratégia para se obter aplicações configuráveis em mecânica computacional, ou seja, um mecanismo que permite a um usuário, com algum conhecimento, definir os tipos de atributos específicos ao seu problema, sem a necessidade de recompilação e re-ligação do programa e sem a necessidade de conhecer os detalhes do código. Desse modo, as aplicações podem ser utilizadas em diversos tipos de problema.

Os sistemas normalmente utilizados para se realizar uma simulação numérica são complexos e, principalmente, do modo como são usualmente implementados, não podem ser utilizados em problemas diferentes dos quais foram originalmente concebidos. Conforme apresentado no Capítulo 2, esses sistemas são bastante grandes e possuem estruturas de dados bastante complexas. Por outro lado, existem aspectos desses sistemas que são comuns a diferentes tipos de simulações, ou seja, independem do tipo de problema, e que poderiam ser reaproveitados em outros tipos de problemas. Para isso é necessário somente a definição de novos tipos de atributos.

A partir dos conceitos discutidos no Capítulo 3 sobre re-uso de *software*, programação orientada a objetos e configuração de aplicações, apresenta-se, no Capítulo 4, uma arquitetura para a configuração de atributos de simulação de mecânica computacional.

De modo a viabilizar a implementação do mecanismo proposto e estabelecer uma total independência entre a parte configurável da aplicação e a parte referente à modelagem, é apresentado um sistema responsável pela comunicação entre essas duas partes, conforme mostrado no Capítulo 5.

As principais características e vantagens do procedimento proposto são resumidas a seguir:

- A arquitetura proposta define uma clara independência entre os serviços da aplicação (modelagem) e o ambiente de configuração de atributos, o que permite um melhor aproveitamento das funcionalidades oferecidas por cada um. Isso é conseguido através do fluxo uni-direcional na interface entre ambas as partes e torna o sistema bastante flexível, pois possibilita o reaproveitamento desses serviços para diversos fins específicos. Isso significa uma aplicação poder ser configurada para atender a diversos tipos de simulações.

- A especificação de funções para a interface com os serviços da aplicação (MIS) possibilita a utilização do ambiente de configuração com qualquer modelador geométrico, sem que haja nenhuma alteração no restante do sistema. Para isso, é necessário somente escrever essas funções para o novo modelador. Isso representa uma grande vantagem em um ambiente onde existem diversos modeladores similares.

- Devido a essas funções de interface, é possível que aplicações existentes tornem-se configuráveis, mesmo não tendo sido concebidas originalmente para isso. Nesse caso, por não terem sido projetados com essa finalidade, a implementação dessas funções nas aplicações pode, em alguns casos, impor algumas modificações no código existente, pois nem sempre as informações requisitadas por essas funções estão disponíveis diretamente na sua base de dados. Esse esforço, no entanto, não representa nenhuma desvantagem se considerada as vantagens obtidas pela incorporação do ambiente de configuração à aplicação. No caso de aplicações novas, a implementação do código pode ser feita, desde o início, de modo a facilitar a implementação dessas funções de interface.

- A classe *Attribute* e suas subclasses formam uma biblioteca que auxilia a tarefa de especificação dos atributos. Novas classes são criadas pelo configurador da

aplicação (usuário avançado), derivadas a partir dessa base, de acordo com as necessidades específicas das aplicações. Desse modo, o usuário não se preocupa com detalhes de implementação do modelador e se concentra somente nas tarefas que as aplicações precisam realizar. Essas classes novas são criadas em um arquivo de configuração, escritas na linguagem de configuração. O fato de esse arquivo de configuração ser interpretado permite a sua modificação sem que seja necessário a recompilação e re-ligação do sistema.

- É oferecido ao usuário avançado a possibilidade de redefinir os métodos das classes referentes às entidades geométricas e aos elementos da malha de elementos finitos ou de contorno. Esses métodos correspondem a ações associadas aos atributos dessas entidades. Desse modo, o configurador da aplicação pode controlar o comportamento do modelador, no que se refere a ações relacionadas com os atributos.

- Para o usuário final, as funcionalidades de especificação e associação dos atributos se apresentam como se fossem parte da aplicação original. Além disso, a abstração dos nós e elementos da malha, oferecida pelas classes *Node* e *Element*, faz com que as informações pareçam existir no formato esperado (desejado) e os detalhes de implementação do modelador são escondidos do usuário e do configurador da aplicação.

- Devido ao fato dos objetos atributo serem especificados independentemente dos objetos geométricos, a arquitetura apresentada permite que o ambiente de configuração proposto seja utilizado em outras aplicações, tais como sistemas de informações geográficas, além de simulação em mecânica computacional. Para isso, é necessário somente a criação de subclasses da classe *Attribute*, referentes ao domínio de interesse da aplicação. Desse modo, a estratégia proposta pode ser vista como uma estratégia para configuração de modeladores geométricos em geral, sendo a aplicação do sistema proposto à mecânica computacional um caso particular.

7.1 Contribuições

As idéias e procedimentos utilizados nesta tese, como por exemplo, re-uso de *software*, programação orientada a objetos, utilização de linguagens de extensão para a con-

figuração de aplicações e uma interface para a comunicação com uma base de dados de uma aplicação não são novidades. A estratégia de se integrar diferentes ferramentas em um mesmo ambiente, de modo a se obter melhores resultados, também não é nova. A arquitetura proposta, entretanto, representa uma estratégia original, no sentido que apresenta uma combinação de idéias e procedimentos, referentes a áreas distintas, em um ambiente para configuração de aplicações em mecânica computacional. O mecanismo para viabilizar a implementação dessa arquitetura também representa uma proposta original.

Este trabalho representa um avanço em simulações de mecânica computacional, pois proporciona um ambiente onde é possível o re-uso da tecnologia da aplicação e novas implementações podem ser feitas com extrema rapidez e relativa simplicidade. O mecanismo proposto é uma contribuição na direção da automação do processo de simulação numérica.

7.2 Direções futuras

Conforme mencionado no corpo deste trabalho, esta tese apresenta um mecanismo para a configuração e associação de atributos a uma descrição geométrica fornecida por algum modelador. Esse mecanismo, no entanto, não possui nenhum formalismo para a definição dos atributos propriamente ditos. Os exemplos apresentados ao longo desta dissertação, que se referem à especificação de atributos, ilustram apenas alguns protótipos implementados com o intuito de testar a arquitetura proposta.

Sendo assim, uma extensão natural dessa arquitetura, consiste na proposição de um formalismo para a definição dos atributos propriamente ditos e no desenvolvimento de uma ferramenta para especificação de atributos baseada nesse formalismo. Alguns protótipos nesse sentido, foram implementados e testados pelo autor, no começo do desenvolvimento deste trabalho. Esses protótipos, entretanto, não foram concluídos, devido a mudança de direção do objetivo da tese. As idéias testadas se mostraram promissoras e devem continuar a serem exploradas.

O desafio de se buscar um formalismo para a definição dos atributos, consiste em se definir o significado de um atributo, ou seja, que informações são necessárias para a

definição de um atributo de maneira geral. Informações a respeito da distribuição do atributo sobre a geometria à qual é aplicado e sobre o sistema de coordenadas no qual o atributo é definido são fundamentais.

A distribuição do atributo pode ser especificada através de uma função escrita em termos de qualquer parâmetro. Os parâmetros podem ser simples variáveis, outros atributos, funções escritas pelo usuário, funções matemáticas, etc. Deve existir um interpretador para essas funções, que permita a avaliação do atributo quando requisitado. Deve existir o conceito de diferentes escopos, ou seja, parâmetros definidos para mais de um atributo (escopo global) e parâmetros definidos somente para o atributo em questão (escopo local).

Apesar de ser possível a especificação de um atributo em um sistema de coordenadas global, é desejável que seja possível a definição dos atributos em relação a sistemas de coordenadas mais convenientes, como sistemas locais das entidades geométricas, sistemas de coordenadas cilíndricas, sistemas paramétricos, etc.

Desse modo, para a definição de um atributo, deve existir uma ferramenta como um editor de atributos. Esse editor deve permitir, de uma maneira simples e flexível, a especificação das informações necessárias. Para a definição das funções de distribuição e definição do sistema de coordenadas deve existir um avaliador de expressões (um protótipo também foi implementado pelo autor). Esse editor deve incluir também mecanismos para a transformação entre os diferentes sistemas de coordenadas.

No sentido de se especificar um formalismo para a definição de atributos, Wong (1994) propõe uma estratégia baseada na idéia de tratar todo atributo como sendo um tensor e apresenta um gerenciador para a especificação dos atributos. Esse procedimento, no entanto, é muito genérico e, por isso, não muito simples de usar. Isso é devido ao fato de o gerenciador ter sido concebido no contexto de um sistema para multi-análises (temperatura, análise de tensões locais e globais e etc.), onde é o responsável pela comunicação de dados entre os diversos módulos do sistema. Desse modo, as tarefas desempenhadas por esse gerenciador são mais complicadas do que a simples configuração e associação dos atributos à descrição geométrica do objeto. Não existe, também, nenhuma ferramenta gráfica para auxiliar a especificação dos atributos.

Outros aspectos referentes à especificação dos atributos e ao ambiente de configuração

proposto, que não foram abordados são:

- *Feedback* do atributo: atualmente o sistema somente oferece a possibilidade de enviar ao modelador uma lista de entidades. O modelador pode fazer o que quiser com essas entidades, como por exemplo, realçá-las com alguma cor. Um problema maior ocorre para se dar um *feedback* do atributo propriamente dito, tal como desenhar uma carga. O problema é que o responsável pelo desenho, normalmente é a parte referente aos serviços da aplicação (o modelador) que não tem conhecimento sobre os atributos a serem desenhados. Por outro lado, o ambiente de configuração não possui conhecimento sobre o sistema gráfico do modelador ou sobre os objetos de interface onde o atributo deve ser desenhado. Uma solução intermediária deve ser estudada.

- Controle de versão: esse problema ocorre, por exemplo, quando se modifica a definição de uma classe e se deseja carregar um arquivo que contém objetos correspondentes à definição antiga. O sistema atual não tem nenhum mecanismo formal para manter a compatibilidade de objetos entre diferentes versões.

- Mecanismo de persistência: estudar uma melhor forma de se verificar a consistência entre os objetos geométrico e a geometria do modelo. Como a geometria é salva pelo modelador e os objetos geométrico são salvos pelo ambiente de configuração, deve existir um mecanismo para se checar essa consistência. Os objetos geométricos somente fazem sentido se a geometria do modelo for correspondente, já que os objetos atributo se referem às entidades geométricas do modelador.

- Como o sistema para gerenciamento de atributos é independente do tipo de análise, tem que ser genérico o suficiente para suportar uma grande variedades de análises e requisitos. Desse modo, deve existir um mecanismo para organizar os atributos em grupos. No sistema atual, já existe a estrutura para isso, mas falta uma melhor formalização.

- Outros problemas são: melhorar o *browser* (com algoritmos mais eficientes de busca), especificar mais funções de interface MIS e mecanismo para *undo*.

Capítulo 8

Referências Bibliográficas

K. J. Bathe, *Finite Element Procedures in Engineering Analysis*, Prentice Hall, 1982.

B. Baumgart, "Winged-Edge Polyhedron Representation", *Stanford Artificial Inteligence*, report no.CS-320, (1972).

B. Beckman, "A Scheme for little languages in interactive graphics", *Software, practice & experience* **21** (1991) 187–207.

J. Bentley, "Programming Pearls: Little Languages", *Communications of the ACM* (Aug 1986).

D. Betz, "Embedded Languages", *Byte* **13** (12) (Nov 1988) 409–416.

J. B. Cavalcante Neto, *'Simulação Auto-Adaptativa Baseada em Enumeração Espacial Recursiva de Modelos Bidimensionais de Elementos Finitos*, Dissertação de Mestrado, Departamento de Engenharia Civil, PUC-Rio, 1994.

P. R. Cavalcanti, P. C. P. Carvalho, L. F. Martha, "Criação e Manutenção de Subdivisões Planares", *Anais do SIBGRAPI IV* (1991) 13-24.

P. R. Cavalcanti, "Criação e Manutenção de Subdivisões do Espaço", Tese de Doutorado, Departamento de Informática, PUC-Rio, 1992.

W. Celes Filho, "Sistema EDG – Manual de Programação", TeCGraf – Grupo de Tecnologia em Computação Gráfica, PUC-Rio, PETROBRÁS/CENPES/SEPROC, 1994.

W. Celes Filho, *Modelagem Configurável de Subdivisões Planares Hierárquicas*, Tese de Doutorado, Departamento de Informática, PUC-Rio, 1995.

S. Chang, *Principles of Visual Programming Systems*, Prentice Hall, 1990.

B. J. Cox, A. J. Novobilsky, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Company, 2nd Edition, 1991.

D. D. Cowan, R. Ierusalimschy, T. M. Stepien, "Programming Environments for End Users", *Anais do 12th World Computer Congress*, **A-14** (1992) (54–60) .

D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, T. M. Stepien, " Abstract Data Views", *Structured Programming*, **14** (1) (Jan 1993) 1–13.

L. H. de Figueiredo, R. Ierusalimschy, W. Celes Filho, "The design and implementation of a language for extending applications", Anais do XXI Semish, 1994.

P. M. Finnigan, A.Kela, J.E.Davis, "Geometry as a Basis for Finite Element Automation", *Engineering with Computers* **5** (1989) 147–160.

B. W. R. Forde, "An application of selected artificial intelligence techniques to engineering analysis" *Ph.D. thesis*, University of British Columbia, Vancouver, B.C. 1989.

C. M. Hoffmann, "Geometric and Solid Modeling: An introduction", Morgan Kaufmann Publishers, 1989.

R. Ierusalimschy, L. H. de Figueiredo, W. Celes Filho, "Reference manual of the programming language Lua", *Monografias em Ciência da Computação* **4/94**, Departamento de Informática, PUC-Rio, 1994.

IXILimited, "IXI ObjectTcl", 1995.

G. E. Krasner, S. T. Pope, " A Cookbook for Using The Model-View-Controller User Interface Paradigm in Smalltalk-80", *JOOP* (Aug 1988) 26–49.

C. W. Krueger, "Software Reuse", *ACM Computer Surveys* **24** (2) (Jun 1992)

J. R. Levine, D. Brown, "Lex & Yacc, 2nd edition", O'Reilly and Associates, 1992

M. Mäntylä, "An Introduction to Solid Modeling", Computer Science Press, 1988

L. F. Martha, "Topological and Geometrical Modeling Approach to Numerical Discretization and Arbitrary Fracture Simulation in Three-Dimensions", *Ph.D. Thesis*, Cornell University, 1989.

Microsoft Corporation, "Visual Basic Programmer's Guide, version 3.0", 1993

PDA Engineering, "Patran 3", 1993.

C. Petzold, "Programando para Windows 3.1", Microsoft Press, 1993.

D. O. Potyondy, A. R. Ingraffea, " A Methodology for Simulation of Curvilinear Crack Growth in Pressurized Fuselages", *proceedings of the International Workshop on Structural Integrity of Aging Airplanes*, (April 1992).

D. O. Potyondy, "A software framework for simulating curvilinear crack growth in pressurized thin shells", *Ph.D. Thesis*, Cornell University, Ithaca, New York 1993.

D. O. Potyondy, P. A. Wawrzynek, A. R. Ingraffea, " Discrete Crack Growth Analysis Methodology for Through Cracks in Pressurized Fuselage Structures", *International Journal for Numerical Methods in Engineering* **38** (10) (1995) 1611–1633.

D. Pountain, C. Szyperski, "Extensible Software Systems", *Byte* **19** (5) (May 1994) 57–62.

J. K. Ousterhout, *Tcl and the Tk ToolKit*, Addison-Wesley, 1994.

A.A.G. Requicha, "Representations of Solid Objects: Theory, Methods and Systems", *ACM Computer Surveys* **12** (4) (1980) 438–464.

A.A.G. Requicha e H.B.Voelcker, "Solid Modeling: A Historical Summary and Contemporary Assessment" *IEEE Computer Graphics and Applications* **2** (2) (1982) 9–24.

A.A.G. Requicha e J.R.Rossignac, "Solid Modeling and Beyond" *IEEE Computer Graphics and Applications* **12** (5) (1992) 31–44.

E. S. Silveira, "Um Sistema Configurável para Simulação Adaptativa Bidimensional de Mecânica Computacional", *Dissertação de Mestrado* , Departamento de Engenharia Civil, PUC-Rio, 1995.

M. S. Shephard, C. N. Tonias C. N. e T. J. Weidner, "Attribute Specification for Finite Element Models", *Computers & Graphics* **6** (2) (1982) 83–91.

M. S. Shephard, P. M. Finnigan, "Integration of Geometric Modeling and Advanced Finite Element Preprocessing", *Finite Elements in Analysis and Design* **4** (1988) 147–162.

M. S. Shephard, "The Specification of Physical Attribute Information for Engineering Analysis", *Engineering with Computers*, **4** (1988) 145–155.

M. S. Shephard, P. M. Finnigan, "Toward Automatic Model Generation", *State-of-the-Art Surveys on Computational Mechanics* **ASME** (1989) 335–366.

Spatial Technology Inc, "ACIS 3D", 1994.

TeCGraf, "MG - Mesh Generator, versão 2.0, Manual do Usuário", *Grupo de Tecnologia em Computação Gráfica*, PUC-Rio, convênio PETROBRÁS/CENPES, 1994.

R. Valdés, "Little languages, big questions", *Dr. Dobb's Journal* **16** (9) (Sep 1991) 16–25.

J. Udell, "ComponentWare", *Byte* **19** (5) (May 1994) 46–56.

P. A. Wawrzynek "Discrete Modelling of Crack Propagation: Theoretical Aspects and Implementation Issues in Two and Three Dimensions", *Ph.D. Thesis*, Cornell University, Ithaca, New York 1991.

K. Weiler, "Topological Structure for Geometric Modeling: Discretization and Arbitrary Fracture Simulation in Three-Dimensions", *Ph.D. Thesis*, Rensselaer Polytechnic Institute, Troy, New York, 1986.

V. S. Wong, "Qualification and Management of Analysis Attributes with Application to Multi-Procedural Analyses for Multichip Modules", *Master Thesis*, Rensselaer Polytechnic Institute, Troy, New York, 1994.

"Uma Estratégia para Desenvolvimento de Aplicações Configuráveis em Mecânica Computacional". Tese de Doutorado apresentada por MARCELO TÍLIO MONTEIRO DE CARVALHO em 29 de Junho de 1995 ao Departamento de Engenharia Civil da PUC-Rio e aprovada pela Comissão Julgadora, formada pelos seguintes professores:

Prof. Luiz Fernando Campos Ramos Martha (Orientador)
Departamento de Engenharia Civil / PUC-Rio

Prof. Marcelo Gattass
Departamento de Informática / PUC-Rio

Prof. Roberto Ierusalimschy
Departamento de Informática / PUC-Rio

Prof. Eurípedes do Amaral Vargas Júnior
Departamento de Engenharia Civil / PUC-Rio

Prof. Luiz Henrique de Figueiredo
IMPA

Prof. Paul Andrew Wawrzynek
Cornell University, USA

Visto e permitida a impressão,

Rio de Janeiro, 02/08/1995

Coordenador dos Programas de Pós-Graduação
do Centro Técnico Científico