Sobre Geradores de Aplicação

Carlos Alberto Alves Meira

Orientação:

Prof. Dr. Paulo Cesar Masiero

Dissertação apresentada ao Instituto de Ciências Matemáticas de São Carlos – USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Setembro de 1991

Dedico este trabalho

*Aos meus pais,
pelo apoio, dedicação e incentivo constantes.*

Agradecimentos

Ao Masiero, pela orientação sempre segura, dedicada e objetiva.

Ao Mauro, pela amizade e companheirismo durante todas as atividades dentro e fora do ICMSC-USP.

Ao Gabriel e João Cangussú pela ajuda no desenvolvimento do trabalho.

Aos professores e funcionários do ICMSC-USP, pela atenção que me foi dedicada.

À CAPES pelo apoio financeiro.

Àqueles que de um modo ou de outro contribuíram para a realização deste trabalho.

Resumo

Geradores de Aplicação são ferramentas de software que conseguem automatizar parte substancial do trabalho rotineiro envolvido na atividade de desenvolvimento de software. Eles aceleram o processo de implementação, transformando especificações de alto nível em produtos da aplicação, e permitem, com isso, que os projetistas de sistemas se preocupem apenas com os aspectos relevantes do problema.

Neste trabalho mostra-se como os geradores de aplicação se encaixam no processo de desenvolvimento de software e discute-se aspectos próprios de sua construção. Apresenta-se, também, uma revisão bibliográfica de vários geradores de aplicação existentes e de ferramentas de apoio à construção desses geradores. Uma classificação de geradores de aplicação é proposta, baseada na capacidade do gerador permitir a mudança de domínios e a mudança dos produtos gerados, e, segundo essa classificação, agrupam-se os geradores relatados na revisão bibliográfica. Apresenta-se, também, um gerador de aplicação do tipo mais geral, dentre os propostos na classificação, desenvolvido neste trabalho.

Esse gerador de propósito geral (ou meta-gerador), chamado GEDAI (Gerador com Domínio e Aplicação Instanciáveis), pode ser instanciado para domínios específicos e aplicações específicas dentro desses domínios. Para demonstrar a sua utilização, são apresentados três exemplos de instanciação: para o domínio de sistemas reativos especificados com statecharts, gerando um programa simulador; para o domínio de conjuntos matemáticos, gerando estruturas de dados em C; e, para o apoio ao desenvolvimento do próprio meta-gerador.

Abstract

Application Generators are software tools which can automatize the routine work involved in Software Development. They accelerate software implementation transforming high level specifications in products, thus allowing systems designers only to worry about the relevant aspects of that problem.

The integration of application generators in the software development process is shown and their construction is discussed. Many application generators are surveyed. An application generator framework based on the generator capability of supporting different application domains and application products is proposed. This framework is used to classify the generators surveyed. An application generator, called GEDAI (Domain and Application Instantiable Generator), was developed, whose type is the most general within the framework proposed.

GEDAI can be instantiated to many domains and to many applications within a domain. Three instantiation examples are shown to demonstrate GEDAI's use: the reactive systems domain specified with statecharts, generating a simulation program; the mathematical set domain, generating C data structures; and, supporting the development of GEDAI itself.

Conteúdo

1	Introdução	1
1.1	O que é um Gerador de Aplicação	1
1.2	Motivações para o Presente Trabalho	2
1.3	Objetivos do Trabalho	2
1.4	Organização do Trabalho	3
2	Geradores de Aplicação	5
2.1	Considerações Iniciais	5
2.2	Geradores de Aplicação	5
2.3	Relatos de Aplicações	12
2.3.1	Editores de Programas Orientados pela Sintaxe	12
2.3.2	Extração em Bases de Dados	17
2.3.3	Protocolos de Comunicação	18
2.3.4	Aplicações com Interface Gráfica	19
2.3.5	Sistemas de Informação	20
2.4	Meta-geradores de Aplicação	21
2.4.1	PG2	21
2.4.2	STAGE	22
2.4.3	DRACO	26
2.5	Avaliação da Bibliografia	27
3	Geradores de Aplicação: Classificação e Construção	29
3.1	Considerações Iniciais	29
3.2	Uma Classificação de Geradores de Aplicação	29
3.3	GEDAI: Gerador com Domínio e Aplicação Instanciáveis	35
3.3.1	O Módulo de Interface	35

3.3.2	O Analisador de Especificações	36
3.3.3	O Gerador de Produtos	40
3.4	Considerações Finais	48
4	Instanciação do Meta-gerador GEDAI	50
4.1	Considerações Iniciais	50
4.2	Um Gerador de Aplicação para Sistemas Reativos	50
4.2.1	A Linguagem de Especificação e o Analisador de Especificações	53
4.2.2	O Produto e a Descrição de Produto	56
4.3	Um Gerador de Aplicação para a Teoria dos Conjuntos	63
4.3.1	A Linguagem de Especificação e o Analisador de Especificações	63
4.3.2	O Produto e a Descrição de Produto	66
4.4	Ferramentas de Suporte Obtidas com a Própria Técnica de Geração	68
4.5	Passos a serem Seguidos na Instanciação de GEDAI	72
4.6	Considerações Finais	74
5	Conclusão	75
5.1	Considerações Gerais	75
5.2	Contribuições do Trabalho Realizado	76
5.3	Sugestões de Futuras Pesquisas	76
	Bibliografia	78
	A Aspectos Gerais da Implementação	82
	B Gramática da Linguagem de Descrição de Produtos (LDP)	84
	C Gramática da Linguagem de Especificação de Statecharts (LES)	87
	D Gramática da Notação BNF Estendida	91

Lista de Figuras

1.1	Visão Superficial de um Gerador de Aplicação	3
2.1	Relacionamento num Ambiente com o Suporte de Geradores de Aplicação . . .	6
2.2	Modelo de Geradores de Aplicação	7
2.3	Processo de Geração de um Editor ALOE	16
2.4	Arquitetura da Abordagem Dirigida por Tabela	22
2.5	Construção de um Gerador com STAGE	23
2.6	Visão Geral de um Editor Orientado pela Sintaxe	27
3.1	Variação da Aplicação	31
3.2	Variação do Domínio	32
3.3	Arquitetura do Interpretador da LDP	47
4.1	Statechart do Mostrador de um Relógio Digital	52
4.2	Arquitetura do Gerador de Simuladores de Statecharts (GSS)	53

Lista de Tabelas

3.1	Modelo para Classificação de Geradores	30
3.2	Classificação de Geradores Existentes	34
3.3	Operações sobre o TAD Árvore Sintática	38
3.4	Categoria dos Símbolos Lógicos da LDP	47
A.1	Tamanho em Linhas de Código Fonte das Partes do Projeto	83

Capítulo 1

Introdução

1.1 O que é um Gerador de Aplicação

Nas primeiras décadas da era computacional, a principal preocupação em relação ao desenvolvimento de sistemas baseados em computador era o desenvolvimento do hardware utilizado nesses sistemas. Com o grande avanço tecnológico ocorrido na área de microeletrônica, o poder computacional aumentou sensivelmente, a um custo cada vez mais reduzido. Como consequência desse avanço, aumentou a demanda por sistemas de software que utilizassem esses recursos em toda a sua potencialidade.

A corrida desenfreada para acompanhar essa demanda e a ausência de uma forma mais organizada e sistematizada no processo de desenvolvimento de software, resultaram, muitas vezes, na produção de sistemas de baixa qualidade. Além da baixa qualidade, vários outros problemas ocorreram, entre eles o alto custo de manutenção dos sistemas, dando origem ao termo “Crise de Software” [Pre87]. A Engenharia de Software surgiu, então, como uma disciplina para o desenvolvimento de software, com o objetivo de estabelecer e usar princípios de engenharia a fim de obter-se economicamente sistemas confiáveis e eficientes.

Na Engenharia de Software, os custos e a produtividade de software são dois aspectos importantes. Os custos atuais para o desenvolvimento são considerados excessivos e a produtividade, em geral, é considerada baixa, devido a fatores como, por exemplo, a habilidade individual e a complexidade do produto [Fai85]. Outro aspecto importante é a qualidade dos sistemas produzidos. O ideal, então, seria encontrar meios para reduzir os custos e ao mesmo tempo aumentar a produtividade, mantendo um bom nível de qualidade do software.

Os Geradores de Aplicação são ferramentas de apoio por computador ao desenvolvimento de software que, além de reduzirem os custos e aumentarem a produtividade, ajudam a melhorar a qualidade dos sistemas produzidos. Essencialmente, um gerador de aplicação é um utilitário que, a partir de uma especificação em alto nível de um problema implementável, transforma automaticamente essa especificação na implementação do problema. O objetivo é automatizar a produção de software, tirando o processo de

implementação das mãos dos projetistas de sistemas e deixando-o a cargo do computador.

As fases de projeto e implementação consomem uma grande parte do tempo de desenvolvimento e introduzem detalhes específicos que não estão relacionados com as funções que devem ser executadas pelo sistema. Os geradores de aplicação permitem que os projetistas de sistemas não se preocupem com esses detalhes e passem a se preocupar somente com os aspectos relevantes da aplicação.

Os geradores de aplicação são indicados quando a implementação de certas aplicações torna-se um processo semelhante e repetitivo. Sendo assim, pode-se definir uma linguagem de mais alto nível e, conseqüentemente, mais significativa que uma linguagem de propósito geral, para especificar essas aplicações, e proporcionar, com o gerador de aplicação, uma maior rapidez no processo de implementação.

1.2 Motivações para o Presente Trabalho

Os geradores de aplicação têm sido utilizados em domínios de aplicação bem definidos. Quando é identificada uma área de aplicação, o gerador é construído especificamente para criar produtos nessa área. Com isso, a construção desses geradores tem sido feita de forma isolada e desorganizada, e a conseqüência é que, apesar dos geradores de aplicação existentes atingirem o mesmo objetivo – a tradução de uma especificação para uma implementação – há pouca semelhança na forma com que esse objetivo é alcançado. Além disso, a construção de geradores de aplicação é trabalhosa [Cle88].

A falta de uma forma organizada e a complexidade na construção de geradores de aplicação faz com que o desenvolvimento de um gerador para um novo domínio seja um grande esforço. Esse esforço, aliado à incerteza do sucesso do gerador de aplicação, muitas vezes acaba desestimulando o início ou até mesmo a conclusão do desenvolvimento do gerador.

Sendo assim, um processo sistemático para o desenvolvimento de geradores de aplicação e a existência de meios para facilitar e agilizar esse desenvolvimento justifica-se, principalmente, pelo potencial de ganho em termos de produtividade e qualidade dos produtos gerados e pela possibilidade de construção de protótipos de geradores de aplicação.

1.3 Objetivos do Trabalho

O objetivo deste trabalho é organizar e facilitar a construção de geradores de aplicação, tendo como base a generalização desses geradores. Apresenta-se um gerador de aplicação de propósito geral e ferramentas de apoio a esse gerador, que permitem que um gerador de aplicação específico seja obtido de maneira mais fácil e rápida que a convencional.

Neste trabalho os geradores de aplicação são divididos em duas partes: uma parte dependente do domínio de aplicação e outra independente do domínio. Uma visão

Figura 1.1: Visão Simplificada de um Gerador de Aplicação

simplificada dessa divisão é apresentada na figura 1.1.

O Analisador de Especificações é a parte dependente do gerador e deve ser construído após identificar-se o domínio de aplicação e definir-se a forma da especificação de entrada. Ele é a interface entre o meio externo e o gerador, que deve analisar as especificações de entrada e produzir as informações necessárias para o módulo seguinte.

O Gerador de Produtos é a parte independente do gerador e, basicamente, é constituído de rotinas que obtêm as informações passadas pelo Analisador de Especificações e produzem o produto final. A grande vantagem do Gerador de Produtos ser independente é que ele pode ser reutilizado no desenvolvimento de todo novo gerador que se queira construir.

O gerador de propósito geral que se apresenta, além de incluir o módulo Gerador de Produtos, oferece ferramentas de apoio à construção do módulo Analisador de Especificações.

1.4 Organização do Trabalho

Este trabalho contém cinco capítulos. Neste capítulo foi introduzido o conceito de geradores de aplicação, mostrando algumas de suas vantagens. Também, foram apresentadas a motivação e o objetivo do trabalho desenvolvido. Além dos capítulos, este trabalho inclui a bibliografia consultada e vários apêndices.

No capítulo 2 é feita a revisão bibliográfica, onde os geradores de aplicação são tratados com maiores detalhes. São apresentados vários relatos do uso de geradores de aplicação em diversas áreas e são apresentadas algumas ferramentas de apoio ao desenvolvimento desses geradores. Ao final do capítulo é feita uma avaliação da bibliografia consultada.

No capítulo 3 é proposta uma classificação de geradores de aplicação e, segundo essa classificação, são organizados os geradores apresentados no capítulo 2. Também, neste capítulo, é apresentado o gerador de aplicação de propósito geral que foi o projeto desenvolvido neste trabalho.

No capítulo 4 são apresentados exemplos de geradores específicos construídos

No capítulo 4 são apresentados exemplos de geradores específicos construídos com o apoio do gerador geral desenvolvido. Ferramentas de apoio ao uso do próprio gerador geral estão incluídas entre esses geradores.

Finalmente, no capítulo 5, são apresentadas as conclusões gerais e algumas contribuições que, acredita-se, são decorrentes deste trabalho. São propostas, também, novas pesquisas e desenvolvimentos a serem investigados no futuro, como decorrência deste trabalho.

Capítulo 2

Geradores de Aplicação

2.1 Considerações Iniciais

Na literatura encontram-se vários exemplos de geradores de aplicação. Eles são de diferentes tipos, variando tanto na forma da especificação de entrada quanto no processo de geração de código.

Neste capítulo é dada uma visão geral do que se entende por Gerador de Aplicação e, também, é apresentada uma abordagem metódica para a sua construção, com o objetivo de ilustrar o processo de desenvolvimento dos geradores. Esse processo é exemplificado na área de ambientes de desenvolvimento de software com relação a editores de programas orientados pela sintaxe.

Vários outros relatos do uso de geradores de aplicação são apresentados, para exemplificar outras áreas de aplicação e mostrar a utilidade desses geradores. São apresentadas também ferramentas para o desenvolvimento de geradores de aplicação, chamadas de geradores de geradores ou meta-geradores. Ao final do capítulo é feita uma avaliação da bibliografia consultada.

2.2 Geradores de Aplicação

Tipicamente, durante a fase de análise de requisitos, o analista de sistemas deve interagir com o cliente para especificar “o que” o sistema deve fazer. Feito isso, o resultado da análise é passado para o projetista de sistemas, que é quem define “como” o sistema deve executar as funções especificadas.

Num ambiente com o suporte de geradores de aplicação surgem dois novos personagens: o **analista de domínios** e o **projetista de domínios**. O analista de domínios é quem especifica os requisitos do gerador, analisando as necessidades e os requisitos de uma série de problemas na área de interesse. O projetista de domínios toma essa especificação e a implementa no gerador. O projetista de sistemas, agora com o auxílio do gerador de aplicação, recebe a especificação do sistema e a transforma numa imple-

mentação. A figura 2.1 [Cle88] mostra como ficam as várias funções num ambiente típico com o suporte de geradores de aplicação.

Figura 2.1: Relacionamento num Ambiente com o Suporte de Geradores de Aplicação

Os geradores de aplicação transformam especificações em produtos da aplicação. Eles operam de maneira semelhante a um compilador, traduzindo informações de alto nível para informações de mais baixo nível. As especificações descrevem o problema ou tarefa a ser feita pela aplicação. Essas especificações podem estar na forma de um diálogo interativo, onde selecionam-se as opções de uma série de menus; podem aparecer na forma gráfica, através de diagramas; ou podem ser escritas numa linguagem textual, às vezes chamada de linguagem de quarta geração, linguagem orientada pela aplicação ou meta-linguagem. Qualquer que seja a forma utilizada, as especificações são usadas pelo gerador de aplicação para criar um ou mais produtos: geralmente estruturas de dados, subrotinas ou segmentos de código.

Seria ideal, na teoria, que o trabalho do projetista de sistemas se reduzisse a fornecer corretamente a especificação para o gerador de aplicação. De fato, se o gerador pudesse produzir o sistema completo, o papel do projetista de sistemas simplesmente desapareceria. Entretanto, na prática, os geradores de aplicação geralmente criam apenas parte de um sistema, deixando o resto do trabalho para o projetista.

A razão pela qual apenas partes de sistemas são criadas por geradores é que as aplicações podem não ser totalmente representadas por uma notação simples. Existem aspectos da aplicação que podem não ser descritos eficientemente, e é preferível ter-se uma representação parcial simples da aplicação, mesmo que restrita, do que uma representação totalmente abrangente, mas complexa. Em [Lev86], Levy afirma que nem sempre a automação completa do processo de produção de software é a solução mais econômica, e sugere critérios para avaliar percentualmente a porção do sistema a ser gerada.

Os geradores de aplicação são normalmente compostos por três módulos básicos: um módulo de interface, um módulo analisador de especificações e um módulo gerador de produtos [Luk86]. Esses módulos podem estar entrelaçados num único módulo ou podem

ser construídos de forma independente, com a saída de um servindo de entrada para o outro. Isso pode ser visto como uma analogia aos compiladores de um único passo e de vários passos, respectivamente. A figura 2.2 mostra como os três módulos se relacionam entre si.

Figura 2.2: Modelo de Geradores de Aplicação

O módulo inicial, de interface com o usuário, pode não aparecer explicitamente em todos os geradores e, nesse caso, a especificação de entrada é textual, podendo ser fornecida num arquivo preparado por algum editor de texto. Quando ele existe, torna mais amigável o uso do gerador, escondendo possíveis detalhes e dificuldades da linguagem de especificação. Em [Luk86], Luker sugere um Módulo de Diálogos como Módulo de Interface e identifica alguns pontos a serem observados no projeto de módulos desse tipo, embora não exista nenhuma disciplina estabelecida para o projeto de diálogos.

O Analisador de Especificações é responsável pelas análises sintática e semântica das entradas e deve produzir como resultado estruturas de dados intermediárias, que são utilizadas pelo módulo seguinte. Essa representação interna pode ser composta de diversas formas de tabelas, árvores sintáticas de derivação, etc.

O módulo Gerador de Produtos é encarregado de gerar o produto desejado pelo usuário. Para isso, ele possui um conjunto de rotinas padrão para acesso às estruturas de dados geradas pelo analisador de especificações e reconhece descrições de produtos em uma linguagem própria, que deve indicar as manipulações necessárias na estrutura de dados intermediária e o formato final do produto.

Quando as três atividades, correspondentes aos três módulos, estão implementadas num único módulo, as estruturas de dados intermediárias não precisam ser expli-

tamente criadas e as informações sobre o produto normalmente estão embutidas no código do gerador.

Os geradores de aplicação oferecem muitas vantagens. Além de aumentarem a produtividade na fase de desenvolvimento de software, os geradores facilitam a fase de manutenção, devido à modularidade e à uniformidade do código gerado. Ainda, para corrigir um erro de especificação, ao invés de se ter que modificar diretamente o código, basta modificar a especificação de entrada e fornecê-la novamente ao gerador.

A especificação, por não conter detalhes de implementação, é mais fácil de ser escrita, lida e modificada. Sendo assim, permite-se que aplicações sejam geradas e mantidas por pessoas sem o conhecimento de linguagens de programação. Um outro ponto forte dos geradores de aplicação é que eles, por acelerarem o processo de codificação, facilitam a construção de protótipos e o teste de especificações alternativas. Outra vantagem é a ausência de erros na tradução da especificação para a implementação.

Assim como os geradores de aplicação oferecem vantagens, eles também causam algumas dificuldades. Eles podem ser usados efetivamente apenas em poucas situações e, frequentemente, reconhecer onde poder usá-los ocorre em fase adiantada do desenvolvimento, desestimulando o reinício do mesmo.

Os geradores de aplicação são difíceis de serem construídos, pois requerem linguagens de especificação cuidadosamente projetadas, um conhecimento íntimo do domínio da aplicação e a habilidade de projetar unidades de software genéricas e confiáveis. O analista de domínios deve decidir qual é a forma da especificação de entrada, e isso requer tanto o conhecimento de projeto de linguagens, como o conhecimento das convenções usadas pelos clientes. Também, é ele quem especifica os requisitos do gerador e, portanto, deve conhecer e entender as necessidades e os problemas dos clientes.

Finalmente, é difícil encaixá-los no processo de desenvolvimento de software: Quem deve considerá-los? Se não estão disponíveis, deve-se construí-los? Como afetam o cronograma? É difícil comparar os benefícios a longo prazo contra os custos a curto prazo de se construir geradores de aplicação.

Em [Cle88], Cleaveland sugere um método para construção de geradores dividido em sete passos. Na verdade, os passos se entrelaçam num único processo global, mas é mais fácil focalizar cada um separadamente. Todos os passos, exceto o último, que é de competência do projetista de domínios, são de competência do analista de domínios, e são os seguintes:

1. Reconhecimento de Domínios.
2. Definição dos Limites do Domínio.
3. Definição do Modelo Básico.
4. Definição das Partes Variante e Invariante.
5. Definição da Entrada da Especificação.
6. Definição dos Produtos.

7. Implementação do Gerador.

Reconhecimento de Domínios: uma vez em uso, o gerador de aplicação parece ser uma ferramenta óbvia e natural para o desenvolvimento de software. Entretanto, reconhecer onde usá-lo, isto é, reconhecer o domínio de aplicação, é o passo mais difícil durante sua construção.

Uma solução para o reconhecimento de domínios é a identificação de padrões. Padrões ocorrem ao nível de código, como grandes comandos seletivos (*case*), como código repetitivo ou como rotinas similares. Padrões podem ocorrer, também, em níveis mais altos, na forma de programas, projetos ou arquiteturas semelhantes.

Definição dos Limites do Domínio: este passo determina o alcance do gerador, ou seja, quais características devem ser incluídas e quais devem ser excluídas.

Definir os limites é definir a abrangência do domínio, isto é, a gama de problemas que o gerador pode solucionar. Como em qualquer projeto de engenharia, existem compromissos entre a abrangência do domínio e o poder do gerador, que é a porção da aplicação gerada automaticamente. Um domínio muito abrangente provoca um decréscimo no poder do gerador, tal que, provavelmente, desenvolver uma aplicação com o gerador não será muito mais rápido do que desenvolvê-la sem ele. Por outro lado, encurtando a abrangência aumenta o poder do gerador, mas para um conjunto limitado de problemas.

Quando determinam-se os limites do domínio, deve-se, também, prever direções futuras. É interessante adotar uma abordagem evolucionária e possibilitar que as características e a forma de especificação sejam extensíveis. Por exemplo: a geração de novos produtos pode requerer informações adicionais de especificação.

Definição do Modelo Básico: um gerador, provavelmente, é mais compreensível, consistente e completo se ele for baseado num modelo básico matemático.

O modelo estabelece a base para se definir consistentemente a semântica do domínio: cada característica pode ser explicada em termos do modelo; a informação da especificação é expressa em termos do modelo; e, o projeto do produto é baseado nele. Além disso, cada produto pode ser verificado e validado com respeito ao modelo.

Alguns modelos básicos comuns são: conjuntos, grafos direcionados e modelos computacionais como, por exemplo, máquinas de estados finitos.

Definição das Partes Variante e Invariante: uma parte variante é algo sob o controle do projetista de sistemas, enquanto que uma parte invariante é algo que não deve ser modificado.

Decidir o que é variante e o que é invariante é análogo a parametrizar uma subrotina, para aumentar o seu grau de reutilização. Os parâmetros da subrotina são as partes variantes, que o programador especifica quando a subrotina é chamada. A subrotina também tem partes invariantes, que não podem ser parametrizadas e, desse modo, não podem ser modificadas: O programador da subrotina é quem deve decidir o que parametrizar.

As partes variantes de um gerador geralmente correspondem à especificação do sistema e as partes invariantes são suposições fixas do domínio e da implementação, que são detalhes de projeto com os quais os usuários não devem se preocupar.

Definição da Entrada da Especificação: a entrada para os geradores pode tomar várias formas. A melhor forma depende da aplicação. Algumas das formas mais comuns são:

- Uma linguagem ou notação textual que expresse todas as informações necessárias. O gerador analisa a entrada e obtém a informação para a construção do produto. Esse método é útil quando existe muita informação ou quando as especificações incluem expressões complexas. Um inconveniente, entretanto, é que o projeto de linguagens é uma tarefa difícil.
- Muitos geradores comerciais utilizam diálogos e abordagens interativas (menus, linguagem de consulta e manipulação direta de objetos na tela) para fornecer informações de especificação. Mudar a especificação significa invocar o diálogo novamente, com a suposição de que o gerador lembra o diálogo anterior. Esse método é útil quando for possível selecionar, através de um menu, a maior parte da informação do problema.
- Diagramas são uma abordagem menos comum para se obter informações de especificação. Esse método é útil para especificações que são naturalmente ilustrativas, como: redes, fluxogramas ou grafos.

Definição dos Produtos: o produto de um gerador pode ser um programa, uma documentação, etc. Geralmente, o gerador formata a informação da especificação numa forma que outras ferramentas possam usar. Essas ferramentas incluem formatadores de texto, compiladores e até mesmo outros geradores.

O produto de um gerador deve ser útil para as diversas aplicações do domínio e, portanto, deve possuir um projeto genérico. A estrutura e a organização da implementação devem ser genéricas a fim de atender às necessidades de todas as aplicações do domínio.

Existem duas abordagens básicas para a geração de código: a abordagem dirigida por código e a dirigida por tabela. A abordagem dirigida por código gera código com os dados embutidos. Por exemplo, se a especificação é formada pelas palavras “um, dois, tres”, e supõe-se que o programa gerado imprima essas palavras, a abordagem dirigida por código poderia gerar o seguinte segmento de código, supondo-se que a linguagem de programação alvo seja C:

```
printf("um\n");  
printf("dois\n");  
printf("tres\n");
```

A abordagem dirigida por tabela, por outro lado, gera código que coloca os dados numa estrutura e, depois, tem acesso a eles através dessa estrutura. Essa abordagem poderia gerar para a mesma especificação o seguinte código:

```
#define num_dados 3

char *dados[] = {
    "um",
    "dois",
    "tres"
};

for (j=0;j<num_dados;++j)
    printf("%s\n",dados[j]);
```

Implementação do Gerador: o último passo consiste em implementar o gerador. O projetista de domínios determina como implementar o gerador a partir dos requisitos fornecidos pelo analista de domínios.

Basicamente, o projetista escreve um programa que traduz a forma da especificação para o produto desejado. O gerador, como a maioria dos tradutores, incluirá um analisador léxico, um analisador sintático, um analisador semântico e um gerador de produto, que produz a saída desejada. Como esses componentes são convencionais e repetitivos, pode-se utilizar ferramentas, e é o que geralmente se faz, para gerá-los.

Os geradores podem ignorar o estilo e a forma de codificação porque os programas gerados não devem ser modificados e raramente são lidos. Entretanto, testar e depurar programas às vezes requer a leitura dos mesmos. Assim, embora um bom estilo de codificação não seja necessário, pode ser vantajoso tornar os programas gerados fáceis de ler.

Ainda com respeito à construção de geradores de aplicação, em [Lev86], Levy apresenta um método para desenvolvimento de sistemas, onde pode-se visualizar, de uma maneira mais geral, o processo de construção de geradores. Levy sugere um método de “Metaprogramação”. Ele define Metaprogramação como a criação de programas de aplicação por meio da construção de programas que produzem programas. O método é dividido em três grandes fases: a fase inicial, que decide o que é necessário para o sistema; a fase de construção das ferramentas e a fase de produção.

O método pode ser resumido como segue: apenas os requisitos necessários para a construção de um protótipo são fixados inicialmente, sendo que esse protótipo tem a arquitetura do sistema final, em termos da estrutura dos dados e da estrutura de controle; uma vez completado o protótipo e fixada a forma do sistema final, é construído um conjunto de ferramentas (geradores de aplicação), que permitem que o sistema real seja construído diretamente da especificação.

A fase de construção das ferramentas começa logo que a forma do produto foi determinada e o software necessário para produzi-lo foi identificado. Se foi determinado na fase inicial, por exemplo, que o modelo abstrato de representação do software é uma máquina de estados finitos, então, durante a fase de construção das ferramentas, será construído um conjunto de programas que produzirá o código final a partir de uma especificação de estados finitos.

2.3 Relatos de Aplicações

Nesta seção serão apresentados vários relatos do uso de geradores de aplicação em diferentes áreas de aplicação.

2.3.1 Editores de Programas Orientados pela Sintaxe

Uma área de aplicação de grande interesse para os geradores de aplicação é a área de Ambientes de Desenvolvimento de Software [Dar87]. A meta principal de ambientes de desenvolvimento de software é oferecer o suporte de ferramentas para facilitar o processo de desenvolvimento de software. Dart classifica os ambientes em quatro categorias: ambientes centrados em linguagens, ambientes orientados por estruturas, ambientes *toolkit* e ambientes baseados em métodos.

Os ambientes centrados em linguagens são construídos em torno de uma linguagem, proporcionando ferramentas adequadas para essa linguagem; os ambientes orientados por estruturas incorporam técnicas que permitem a manipulação direta de estruturas sintáticas; os ambientes *toolkit* oferecem uma coleção de ferramentas, independentes de qual seja a linguagem, para o suporte a tarefas do tipo *programming-in-the-large*, tais como gerenciamento de configuração e controle de versões do sistema; e por último, os ambientes baseados em métodos dão suporte a um determinado método de desenvolvimento de software.

No contexto deste trabalho, o interesse é por ambientes orientados por estruturas, pois a independência das técnicas utilizadas, com respeito a linguagem de programação, conduz à noção de geradores de tais ambientes. Primeiramente, serão vistas algumas das características de ambientes orientados por estruturas, para depois tratar-se da geração de tais ambientes.

O elemento principal de ambientes orientados por estruturas é o editor orientado pela sintaxe da linguagem de programação. Nesse tipo de editor, os programas são vistos como composições hierárquicas de estruturas, ao invés de simples seqüências de caracteres, palavras e linhas. Essas estruturas são as construções sintáticas da linguagem.

A edição é orientada pela gramática que define a linguagem, e os programas são mantidos internamente numa estrutura de árvore. Essa árvore é a árvore sintática de derivação do programa, onde os nós representam os símbolos não terminais e terminais da gramática. Os nós representando os símbolos não terminais identificam as construções sintáticas da linguagem, e os nós representando os símbolos terminais são os identificadores e as expressões definidas pelo usuário. O nó raiz da árvore de derivação é o símbolo inicial da gramática.

Na derivação de um programa, a utilização de uma produção da gramática é representada por uma subárvore da árvore sintática. A raiz da subárvore é o símbolo não terminal do lado esquerdo da produção e os filhos desse nó são os símbolos terminais e/ou não terminais do lado direito.

Programas completos são representados por uma árvore onde todos os nós in-

intermediários são símbolos não terminais e todos os nós folhas são símbolos terminais da gramática. Por outro lado, programas incompletos são árvores onde alguns dos nós folhas são símbolos não terminais. Esses símbolos devem ser expandidos, utilizando-se as produções associadas a eles, para completar a árvore e, conseqüentemente, tornar o programa completo.

Apesar de serem mantidos internamente como árvores, os programas são vistos pelos usuários como seqüências de caracteres, palavras e linhas, como se fossem arquivos textos. Os usuários só percebem a diferença através das manipulações sobre os programas, que são dirigidas pelas construções sintáticas da linguagem. Inserções e remoções são vistas como inserções e remoções de estruturas inteiras, e os movimentos do cursor de edição são realizados de uma estrutura para outra.

Nos programas incompletos, os símbolos não terminais não expandidos são identificados para os usuários. São pontos de extensão (*placeholders*), onde o usuário deve posicionar o cursor e inserir as construções sintáticas desejadas. Geralmente, os pontos de extensão são apresentados em destaque no programa, para serem melhor visualizados.

A criação de um programa começa com a expansão dos pontos de extensão associados aos símbolos não terminais do lado direito da produção que tem como lado esquerdo o símbolo inicial da gramática. A expansão desses pontos de extensão pode introduzir novos pontos de extensão, que devem ser expandidos, e assim sucessivamente, até que sejam substituídos por símbolos terminais. A construção do programa termina quando todos os pontos de extensão forem substituídos por símbolos terminais.

A grande vantagem dos editores orientados pela sintaxe é que eles produzem programas sintaticamente corretos. São dois os motivos para que os programas produzidos estejam sintaticamente corretos: primeiro, como são feitas inserções de construções sintáticas inteiras, as palavras reservadas e os símbolos especiais da linguagem são inseridos automaticamente pelo editor (as palavras reservadas e os símbolos especiais são pré-definidos pelo editor e, portanto, estão sintaticamente corretos); segundo, a inserção num ponto de extensão só é permitida para aquelas construções que estão associadas a ele, isto é, para as produções que tem como lado esquerdo o símbolo não terminal associado ao ponto de extensão. Assim, impede-se que os usuários insiram construções sintáticas em lugares impróprios. Por exemplo, num ponto de extensão onde é esperada a inserção de uma expressão não é permitida a inserção de um comando de atribuição.

Visto como funcionam os editores orientados pela sintaxe, agora serão tratados alguns dos aspectos relacionados com a construção de tais editores. No começo, a construção desses editores era complicada e trabalhosa, e para cada linguagem de programação, para a qual se desejasse um editor orientado pela sintaxe, iniciava-se um novo projeto. Era necessário, portanto, encontrar um meio mais simples de construí-los.

Com a experiência adquirida na construção dos primeiros editores, verificou-se, entretanto, que muitos dos aspectos relacionados com editores orientados pela sintaxe são independentes da linguagem de programação para a qual se deseja o editor. Visualizava-se, então, uma área (ou domínio) para o emprego de geradores de aplicação: os editores orientados pela sintaxe. A partir daí, começaram a surgir os primeiros geradores de editores orientados pela sintaxe. A idéia básica é que, a partir de descrições de linguagens

de programação, em forma de gramática, o gerador consiga produzir instâncias de editores para essas linguagens.

Na literatura existem vários relatos de aplicações de geradores de editores orientados pela sintaxe. Abaixo, são apresentados alguns exemplos, identificando-se aspectos próprios de cada um.

SYNTHESIZER GENERATOR

O gerador de editores orientados pela sintaxe *Synthesizer Generator* [Rep84A] foi desenvolvido após se ter construído o *Cornell Program Synthesizer* (CPS), que é um ambiente de programação interativo com facilidades integradas para a criação, edição, execução e depuração de programas. O gerador, independentemente da linguagem de programação, através de descrições externas de linguagens, cria ambientes semelhantes ao CPS.

A criação de programas no CPS [Tei81A, Tei81B] é análoga à derivação de sentenças, com respeito a uma gramática livre de contexto. Essa gramática é incorporada numa coleção de gabaritos (*templates*), definidos para cada tipo de comando da linguagem. Um gabarito é uma sequência de caracteres, símbolos de pontuação e caracteres de controle, que define a forma de apresentação na tela dos comandos. Por exemplo, o gabarito de um comando condicional pode ser visto pelo usuário no seguinte formato:

```
IF (<condicao>)
  THEN <comando>;
  ELSE <comando>;
```

onde <condicao> e <comando> são os pontos de extensão.

O editor do CPS é uma mistura de editor de estruturas em árvore e editor de textos. Os gabaritos são gerados por comandos, mas as expressões e as atribuições são construídas textualmente, ou seja, um caracter por vez. A interface textual para as expressões é bem mais simples e menos monótona do que ter que derivá-las através de comandos. Com a interface textual, entretanto, surge o problema de garantir-se a correção sintática, mas isso é resolvido analisando as expressões, por meio de um analisador sintático, logo após o término de suas construções.

No CPS, não só a edição é facilitada pelo tratamento de programas como composições hierárquicas de estruturas. Outras atividades, como execução e depuração, também o são. A execução segue a edição, com o código sendo gerado para cada unidade de programa editada, eliminando o passo de compilação. Desse modo, as unidades sintáticas do editor correspondem às unidades computacionais do interpretador, e essa característica proporciona ao programador várias opções para o acompanhamento da execução e para a depuração do programa.

Vistas as características do editor do CPS, agora pode-se ver como o *Synthesizer Generator* gera editores semelhantes a ele. A especificação da linguagem de programação fornecida ao gerador é dada na forma de "Gramática de Atributos" [Knu68, Dem81]. Gramática de atributos é um formalismo que estende o poder descritivo de

gramáticas livres de contexto, permitindo a incorporação de relações semânticas da linguagem. Para cada símbolo não terminal da gramática estão associados atributos, e para cada produção estão associadas equações semânticas envolvendo esses atributos. Essas equações semânticas definem o significado das construções da linguagem.

Durante a edição, os programas são representados por árvores de derivação consistentemente atribuídas. Modificações nos programas reestruturam a árvore de derivação, e essa reestruturação afeta os valores de alguns atributos, podendo comprometer a consistência nos relacionamentos entre eles. Com a gramática de atributos permite-se o reestabelecimento automático dessa consistência através da “Análise Incremental”, que atualiza os valores desses atributos [Rep83].

Na especificação em forma de gramática de atributos de um editor estão definidos: a sintaxe abstrata, as equações semânticas, o formato de apresentação na tela do programa e a interface de entrada com o usuário. A sintaxe abstrata é definida por regras gramaticais de uma formulação de gramática livre de contexto. Essas mesmas regras são usadas para definir os atributos que compõem as equações semânticas. Com isso, além da uniformidade proporcionada, permite-se a construção de especificações modulares. O formato de apresentação (*unparsing scheme*) é definido para cada produção, consistindo de uma sequência de cadeias de caracteres, nomes de atributos e nomes de símbolos não terminais. Para especificar a interface, deve-se fornecer a sintaxe concreta de entrada, junto com equações semânticas. Essas equações semânticas definem a tradução da sintaxe concreta para a sintaxe abstrata, e é esse mecanismo de tradução que permite definir interfaces estruturais e/ou textuais na proporção desejada.

À parte o seu uso na geração de editores de linguagens de programação, *Synthesizer Generator* tem sido usado para criar outras ferramentas que manipulam dados estruturados, como por exemplo: calculadoras, provadores de teoremas e formatadores de texto [Rep84A,Rep84B].

GERADOR ALOE

O projeto GANDALF [Hab86,Not85], além de facilitar e reduzir o custo de construção de ambientes de desenvolvimento de software, gera ambientes que se caracterizam principalmente: pela integração das atividades envolvidas no processo de desenvolvimento, pela uniformidade da interface e pela interação com o usuário. Semi-automaticamente, o Sistema Gandalf [Ell85] gera ambientes com essas características.

A geração desses ambientes é executada com o auxílio do gerador ALOE de editores orientados pela sintaxe. A visão em alto nível desse processo é mostrada na figura 2.3 [Hab86]. Tabelas da linguagem, representando as sintaxes abstrata e concreta, são construídas numa forma abstrata. Essas tabelas são transformadas pelo ALOEGEN, um editor que auxilia a construção de novos editores, em tabelas na linguagem GC (Gandalf C), que depois são compiladas. Separadamente, rotinas de ação são definidas em GC e compiladas. Por último, os códigos objetos para as tabelas e para as rotinas de ação são ligados com o código objeto do núcleo, para produzir o editor.

Os editores ALOE são editores de estruturas de árvore. Os comandos fornecidos para a manipulação dessas estruturas são de dois tipos: comandos construtivos, depen-

Figura 2.3: Processo de Geração de um Editor ALOE

dentes da linguagem, e comandos independentes da linguagem. Os últimos formam o núcleo, e são compartilhados por todos os editores ALOE. Comandos típicos são: comandos para a remoção de estruturas ou para a movimentação do cursor. Cada estrutura da linguagem possui um comando construtivo, que formata na tela as palavras chaves da estrutura e identifica os pontos de extensão.

Na descrição da linguagem, que serve como entrada para o gerador, devem ser especificadas a sintaxe abstrata, a sintaxe concreta e os nomes das rotinas de ação, que são chamadas enquanto a árvore de derivação é manipulada.

A gramática da sintaxe abstrata descreve: os “operadores”, que incluem os terminais e não terminais; e as “classes”, que indicam o conjunto de operadores que podem substituir determinados pontos de extensão.

A sintaxe concreta descreve a representação na tela de cada operador. A representação visual de um programa é derivada da árvore sintática pelo *unparser*, que é uma função do editor que mapeia a árvore de representação para a representação textual correspondente. Um exemplo de descrição de sintaxe concreta pode ser:

WHILE = while Q1 do QN Q2

onde Q1 e Q2 identificam os nós filhos do operador WHILE, e QN indica a inserção de uma nova linha na saída.

As rotinas de ação implementam o comportamento em tempo de execução, que se deseja realizar no ambiente que está sendo criado. Rotinas de ação podem ser usadas para vários propósitos, incluindo checagem de tipos e geração de código.

Como exemplos de ambientes Gandalf, gerados seguindo o processo acima, podem ser citados [Hab86]: *Gandalf Prototype* (GP), que é um protótipo com todas as características de ambientes Gandalf; GNOME, que é um conjunto de ambientes de programação usado para fins educacionais; e o próprio ambiente ALOEGEN.

Outras aplicações interessantes de sistemas geradores de editores orientados pela sintaxe incluem: Mentor, que utiliza uma linguagem descritiva, chamada Metal, para construir ambientes multilíngüe [Don80,Don84]; PSG (*Programming System Generator*), que gera editores totalmente híbridos, permitindo a edição de texto ou estrutura em qualquer ponto [Bah85,Bah86]; e POE, onde os editores automaticamente reparam erros sintáticos de programas criados textualmente [Fis84].

Um outro tipo de editor de certo modo sensível à sintaxe são os editores diagramáticos, especializados em técnicas diagramáticas como, por exemplo, técnicas de apoio às fases de análise e projeto, tais como os Diagramas de Fluxo de Dados e os Diagramas de Entidade e Relacionamento. Esses editores criam os diagramas apenas com os elementos definidos pela técnica, que devem ser especificados ao editor de alguma forma.

Os editores diagramáticos também podem ser gerados. No projeto AMADEUS (Ambiente e Metodologia Adaptáveis de Desenvolvimento Unificado de Software) existe uma ferramenta, chamada Editor Diagramático Generalizado (EDG), que é um gerador desse tipo de editor [Mra90].

O EDG suporta a descrição de técnicas diagramáticas, possíveis de serem abstraídas como grafos, através de gramáticas de atributos, e depois oferece facilidades na edição dos diagramas. O usuário interage com o EDG através do Editor Diagramático Específico (EDE), que permite somente a manipulação de diagramas de técnicas já especificadas. A interface do EDE é uniforme para todas as técnicas suportadas.

2.3.2 Extração em Bases de Dados

A aquisição de dados é uma parte fundamental do processo de análise, existindo uma grande variedade de fontes de dados com tamanhos diferenciados. No caso de grandes conjuntos de dados, é preciso estabelecer um meio eficiente para o analista extrair subconjuntos úteis desses dados, com o mínimo de esforço e conhecimento de programação. Também, é necessário proporcionar ao analista uma interface uniforme, escondendo as diferenças na forma de acesso de cada fonte.

Em [Bel85], Belanger apresenta um conjunto de ferramentas que permitem que programas sejam gerados, em várias linguagens de programação, para a extração de dados em diferentes bases de dados. Essas ferramentas formam um sistema que facilita a construção dos geradores de programas e será tratado mais adiante. Especificamente, são apresentados dois geradores de programas: um para a extração em bases de dados IMS (*Information Management System*), chamado IMX (*IMS-to-UNIX*), e outro para a extração em fitas geradas por computadores IBM, chamado TTU (*Tape-to-UNIX*). IMX gera programas nas linguagens COBOL, JCL e DL/I, enquanto TTU gera programas na linguagem C.

Nas duas ferramentas proporciona-se ao usuário uma interface uniforme e de alto nível. Por exemplo, qualquer usuário que queira obter dados de fitas usando TTU deve seguir os seguintes passos: primeiro, o usuário ou o administrador deve criar um dicionário (um arquivo descrevendo a fita), contando com o apoio de um editor orientado pela sintaxe da linguagem definida para descrever as fitas; uma vez criado o dicionário,

os campos e os registros a serem selecionados podem ser especificados através de uma linguagem de alto nível. Com essas informações TTU gera o programa que executa as operações desejadas. O dicionário da fita, depois de criado, fica à disposição para novas consultas.

Tanto IMX quanto TTU geram programas eficientes, provavelmente mais eficientes do que seriam os programas construídos manualmente por programadores.

2.3.3 Protocolos de Comunicação

O desenvolvimento de técnicas de descrição formal para a especificação de protocolos de comunicação tem evoluído muito com a pesquisa e o desenvolvimento de técnicas formais e ferramentas para projeto, validação, implementação e testes de protocolos. Um grande número de compiladores e interpretadores, baseados nessas técnicas, tem sido desenvolvido para automatizar a implementação de protocolos.

Em [Vuo88], Vuong apresenta as idéias básicas para se implementar semi-automaticamente protocolos de comunicação, utilizando um **Compilador Estelle-C**. Estelle (*Extended State Transition Language*) é uma técnica de descrição formal de protocolos.

O compilador lê uma especificação de protocolo na linguagem Estelle e gera código na linguagem C. Esse segmento de código é depois completado pela incorporação de detalhes dependentes do sistema e de rotinas de suporte pré-definidas. A porção dependente do sistema é a interface entre a máquina de protocolo especificada e o seu ambiente de trabalho. Por exemplo: as interações com o sistema operacional.

Os analisadores léxico e sintático para o compilador foram gerados pelos utilitários padrões do sistema UNIX: LEX (*LEXical Analyzer Generator*) e YACC (*Yet Another Compiler-Compiler*), respectivamente. A manipulação de erros, o gerenciamento de tabelas e a geração de código foram embutidos na gramática de entrada para o utilitário YACC.

Vuong, para avaliar a utilidade do compilador Estelle-C, realizou a implementação de um protocolo de transporte ISO (*International Standard Organization*) classe 2. O protocolo de transporte ISO proporciona um mecanismo confiável e eficiente para a troca de dados entre processos de sistemas diferentes. Subseqüentemente, o protocolo foi reimplementado manualmente, proporcionando a comparação entre as duas abordagens. Ambas as implementações foram testadas rodando num VAX 11/750, bem como em várias estações de trabalho SUN, sob controle do sistema operacional UNIX 4.2 BSD.

A vantagem de se usar a abordagem semi-automática é que o código gerado é bem estruturado, a concordância com a especificação é praticamente garantida e as propriedades dependentes do sistema são isoladas, tornando-se fácil a manutenção e a portabilidade da implementação. Por outro lado, a implementação manual obtém vantagens do ambiente de trabalho, reduzindo significativamente a quantidade de código e processamento.

Vuong sugere que uma boa prática é começar com a implementação semi-automática, porque economiza o tempo de desenvolvimento. Se o código gerado não for eficiente,

parte-se então para a implementação manual. A experiência adquirida durante a fase de automação é de grande valia para obter-se uma implementação manual otimizada.

2.3.4 Aplicações com Interface Gráfica

Nesta seção serão apresentados três exemplos de geradores de aplicação baseados em linguagens de especificação gráficas.

TAGS

O sistema de ferramentas TAGS (*Technology for the Automated Generation of Systems*) é baseado numa linguagem de especificação gráfica e executável, apropriada para sistemas de informação e sistemas embutidos [Lew90]. A linguagem permite uma descrição gráfica detalhada de todos os requisitos de entrada, processamento e saída do sistema.

O sistema é composto por três elementos básicos [Sie85]: a linguagem de especificação IORL (*Input/Output Requirements Language*), um conjunto de ferramentas automatizadas e a metodologia TAGS. Essa metodologia é baseada no “Paradigma Centrado em Automação” [Boe83].

Os símbolos gráficos da linguagem definem explicitamente todos os componentes, as interfaces, o fluxo de dados, a lógica de controle, os processos, as variáveis e as restrições de tempo, formando uma hierarquia que estabelece os relacionamentos entre todas as entidades do sistema.

Uma especificação TAGS pode ser analisada e compilada automaticamente, produzindo uma simulação executável do projeto. TAGS gera código fonte na linguagem Ada e roda em estações de trabalho gráficas IBM, SUN, DEC e HP/Apollo, com sistema operacional UNIX ou AEGIS.

MICROSTEP

Microstep é um gerador de aplicação projetado para o processamento de dados e para aplicações dirigidas principalmente pelo fluxo de dados entre os processos [Lew90]. Ele não dá apoio à construção de sistemas de tempo real. Sua linguagem de especificação é baseada em diagramas de fluxo de dados, similares aos diagramas de Yourdon [You78].

Os diagramas de fluxo de dados, os objetos de dados (estruturas e formatos) e os processos são criados com editores gráficos; o controle do fluxo e a lógica também são especificados; a especificação pode conter chamadas para outras especificações, permitindo a construção de grandes sistemas; e, código na linguagem de programação alvo pode ser incluído, permitindo a execução de tarefas difíceis de serem especificadas no Microstep.

O gerador de aplicação, a partir do diagrama de especificação, gera código na linguagem C, que é compilado e ligado automaticamente. Microstep roda em microcomputadores PCs com processadores 286 ou 386, sob o controle do sistema operacional MS-DOS.

GERADOR DE GERENCIADORES DE SISTEMAS REATIVOS

O terceiro exemplo de gerador de aplicação baseado numa linguagem de especificação gráfica é o Gerador de Gerenciadores de Sistemas Reativos (abreviado neste trabalho por G.G.S.R.) [Fig91]. Esse gerador recebe como entrada uma especificação criada através de Estadogramas (*Statecharts*¹) e produz um programa em C, funcionalmente equivalente à especificação. Em particular, trata-se de uma ferramenta útil no desenvolvimento de gerenciadores de diálogos em interfaces homem-máquina.

Na verdade, a entrada para o gerador propriamente dito é uma descrição textual de um estadograma, produzida por um editor gráfico de bolhas (componentes de um estadograma). Esse editor gráfico é um módulo independente do módulo gerador e ainda não está pronto, estando em fase inicial de desenvolvimento. Entretanto, considerou-se aqui que o G.G.S.R. são os dois módulos, o editor gráfico e o módulo gerador, integrados.

2.3.5 Sistemas de Informação

Os primeiros geradores de aplicação surgiram na área de Sistemas de Informação e tinham como base um sistema de gerenciamento de bases de dados, ao qual proporcionavam, principalmente, uma linguagem de alto nível para a manipulação dos dados da base e para a geração automática de relatórios [Hor85]. Essas linguagens, chamadas Linguagens de Quarta Geração, evoluíram e atualmente abrangem várias outras aplicações nessa área, dentre as quais podem ser citadas [Mar85]:

- Geração de Telas.
- Geração de Diálogos.
- Geração de Gráficos.
- Manipulação de Regras Lógicas e Aritméticas.

Esses geradores de aplicação normalmente misturam as atividades de análise sintática e geração e/ou interpretação de código, como a maioria dos compiladores de linguagens de programação convencionais, e, portanto, não fica clara a divisão entre essas atividades. Exemplos de geradores de aplicação que utilizam linguagens de quarta geração são: Nomad, Ramis, Focus e as várias versões do Dbase [Hor85].

Mesmo as linguagens de quarta geração, para aplicações bem simples, como uma busca sequencial de um arquivo, mostrando cada registro na tela, requerem construções de baixo nível, similares às de linguagens algorítmicas, obrigando o programador a ter em mente a estrutura da máquina. Segundo Setzer, em [Set91], uma solução para este problema é a Programação Super-estruturada (SSP – *Superstructured Programming*), que permite ao programador pensar apenas na estrutura do problema.

Uma especificação em SSP de uma tarefa não reflete a estrutura da máquina, mas reflete uma organização das atividades – bem próximas às atividades humanas –

¹A técnica *Statechart* será descrita mais adiante neste trabalho.

necessárias para executar a tarefa. Sendo assim, especificações em SSP são mais simples, claras e compreensíveis e, naturalmente, podem constituir a documentação do problema.

Especificações em SSP também permitem a geração automática de código em linguagens de programação tradicionais. Geradores de aplicação utilizando descrições em SSP têm sido utilizados principalmente no processamento de dados comerciais, diminuindo sensivelmente os esforços de teste e manutenção e proporcionando documentações atualizadas. Segundo Setzer, SSP pode também produzir código para software básico, como compiladores e sistemas operacionais.

Um exemplo de gerador de aplicação que recebe especificações em SSP é o sistema I-M-E, que utiliza, em parte, Tabelas de Decisão para especificar processamento de dados comerciais [Set91]. As especificações para o I-M-E procuram organizar as atividades de processamento de dados nos seguintes passos, que devem ser executados sequencialmente:

1. Passo de Inicialização (**I** – *Initialization Step*), é o passo de preparação dos dados. Ele é executado uma única vez e pode ser vazio, isto é, pode não ser necessário.
2. Passo Principal (**M** – *Main Step*), que descreve como cada elemento do conjunto de dados deve ser processado. Ele é repetido para cada elemento do conjunto de dados, mas isto deve estar implícito na especificação.
3. Passo de Finalização (**E** – *Ending Step*), que, eventualmente, realiza um processamento final, depois que todo elemento do conjunto de dados for processado. Ele também é executado uma única vez e pode ser vazio.

O sistema I-M-E, então, a partir de uma especificação que descreva os passos acima, cria um programa em Pascal que executa o processamento desejado. I-M-E é um sistema experimental, de uso acadêmico, que roda em computadores IBM-PC. A versão atual não suporta ainda a geração de relatórios.

2.4 Meta-geradores de Aplicação

Para facilitar a construção de geradores de aplicação, têm sido utilizados geradores de analisadores léxicos e analisadores sintáticos. Além desses, novas ferramentas, chamadas de ferramentas para o desenvolvimento de geradores de aplicação, têm sido desenvolvidas com o intuito de reduzir o esforço necessário na construção de geradores. Nesta seção serão apresentadas três dessas ferramentas, que também podem ser chamadas de “Geradores de geradores” ou “Meta-geradores.”

2.4.1 PG2

Na seção 2.3.2 foram apresentados os geradores IMX e TTU de programas para a extração de dados. Esses geradores foram construídos com o apoio de um sistema de geração de código dirigido por tabela, chamado PG2, o qual pode ser “montado” para

gerar código para diferentes modos de extração [Bel85]. No caso dos geradores IMX e TTU, existe uma tabela para a extração em bases de dados IMS e uma tabela para a extração em fitas geradas por computadores IBM, respectivamente. Essa abordagem de geração de código dirigido por tabela possui a arquitetura mostrada na figura 2.4.

Figura 2.4: Arquitetura da Abordagem Dirigida por Tabela

Primeiro, um arquivo descrevendo a base de dados fonte deve ser criado, o qual também fornece informações sobre parte do processo de geração de código. Depois, o analista requisita a porção dos dados a ser extraída numa linguagem de alto nível, que é analisada sintaticamente e semanticamente, produzindo uma “árvore de recuperação atribuída.” O gerador de código percorre essa árvore e traduz o pedido do usuário num programa equivalente, usando uma tabela que contém uma abstração dos programas de extração.

A abstração do programa alvo, dada num arquivo separado, é escrita numa linguagem especial, chamada KS, que estabelece diretivas ao gerador de código sobre a manipulação da árvore de recuperação. Ela é construída uma única vez para cada sistema de base de dados.

A árvore de recuperação produzida pelos analisadores sintático e semântico é semelhante à árvore de derivação produzida pelo *front end* (análise léxica, sintática e semântica) de alguns compiladores. Interpretar e efetuar expansões na abstração de programa, com a árvore de recuperação como base, é análogo a uma combinação dos métodos interpretativo e dirigido por tabela, usados em geradores de código redirecionáveis.

A abordagem dirigida por tabela permite que as descrições das bases de dados e que as estruturas dos programas gerados sejam facilmente modificadas. Permite-se também a incorporação de “eficiências” no código, que normalmente não seriam utilizadas por um programador para fazer o mesmo trabalho.

2.4.2 STAGE

Na seção anterior os geradores construídos são específicos para o domínio de bases de dados. Em [Cle88], é apresentada uma ferramenta, chamada STAGE, que é utilizada na construção de geradores para qualquer área de aplicação. Ela tem sido aplicada com

sucesso tanto em áreas convencionais quanto em áreas não convencionais, e entre algumas aplicações podem ser citadas: interfaces com usuário, ferramentas de teste, aplicações baseadas em máquinas de estados finitos e tradutores de projeto de hardware.

STAGE também é um gerador de aplicação e, como todo gerador, recebe como entrada uma especificação e produz como saída um produto, que no caso é um novo gerador de aplicação. A entrada para a ferramenta STAGE é dividida em duas partes (figura 2.5): (1) uma descrição fonte em forma de gramática, que descreve a entrada para o gerador; e (2) uma descrição de produto que descreve a saída do gerador.

Figura 2.5: Construção de um Gerador com STAGE

A partir dessas descrições, STAGE produz o gerador de aplicação, que é uma união de partes produzidas automaticamente: um analisador léxico, um analisador sintático, estruturas de dados, rotinas de construção da árvore de derivação, rotinas para o percurso da árvore e rotinas de geração de produto.

Para exemplificar o uso de STAGE seja o seguinte exemplo: suponha que exista uma coleção de aproximadamente 1000 palavras classificadas em conjuntos e deseja-se implementar esses conjuntos em estruturas de dados na linguagem C. A especificação de entrada para o gerador é dada na seguinte forma, considerando a seguinte parte da coleção de palavras:

```
cores {vermelho, azul, verde}  
cidades {piracicaba, campinas}  
insetos {formiga, aranha, mosca, abelha}
```

onde cores, cidades e insetos são os nomes dos conjuntos. A implementação é feita com arranjos (*arrays*), e para a especificação acima tem-se o seguinte produto:


```

int nro_conjuntos = 3;

char *nome_conjunto[] = {
    "cores",
    "cidades",
    "insetos",};

int tamanho_conjunto[] = {3,2,4,};

char *conjunto_cores[] = {
    "vermelho",
    "azul",
    "verde",};

char *conjunto_cidades[] = {
    "piracicaba",
    "campinas",};

char *conjunto_insetos[] = {
    "formiga",
    "aranha",
    "mosca",
    "abelha",};

char **valores_de_conjunto[] = {
    conjunto_cores,
    conjunto_cidades,
    conjunto_insetos,};

```

Identificadas as formas de entrada e saída do gerador de aplicação, agora deve-se definir a descrição fonte e a descrição de produto necessárias para a construção desse gerador, utilizando a ferramenta STAGE. A descrição fonte é uma gramática para a linguagem de especificação:

```

%grammar

*espec: (def_conjunto+)

def_conjunto: (nome_conjunto "{"
                elemento+ separado-por ","
                "}"
            )

nome_conjunto: < [a-zA-Z0-9_]+ >

```

elemento: < [a-zA-Z0-9_]+ >

%product tabela.c

A última linha significa que o gerador produzirá um único produto chamado tabela.c. A descrição de produto é dada num arquivo gabarito:

%template

```
int nro_conjuntos =  
    %d(length(def_conjunto(top)));
```

```
char *nome_conjunto[] = {  
    %forall n:NOME_CONJUNTO %loop  
        "%(tok(n))",  
    %endloop};
```

```
int tamanho_conjunto[] = {  
    %forall s:DEF_CONJUNTO  
        %loop %d(length(elemento(s))),  
    %endloop};
```

```
%forall s:DEF_CONJUNTO %loop  
    char *conjunto_%(tok(nome_conjunto(s)))[] = {  
        %forall e:ELEMENTO in s %loop  
            "%(tok(e))",  
        %endloop};  
%endloop
```

```
char **valores_de_conjunto[] = {  
    %forall n:NOME_CONJUNTO %loop  
        conjunto_%(tok(n)),  
    %endloop};
```

As frases em itálico começando com % descrevem como obter dados da árvore de derivação para a produção da saída. Por exemplo: a frase *%d(length(def_conjunto(top)))* produzirá o número de definições de conjuntos encontradas ao se percorrer a árvore de derivação a partir do topo; a frase *%tok(n)* produz o símbolo (*token*) associado ao nó *n*; e a frase *%forall n:NOME_CONJUNTO %loop* percorre a árvore de derivação visitando todos os nós do tipo *NOME_CONJUNTO*. O restante do texto em romano é colocado diretamente no produto.

A partir dessas duas descrições, STAGE produz um gerador que aceita especificações de conjuntos de palavras na linguagem de especificação definida pela gramática, e as transforma em estruturas de dados na linguagem C (especificação fonte e produto, respectivamente, na figura 2.5).

2.4.3 DRACO

A reutilização de software na construção semi-automática de sistemas segue duas abordagens básicas, que se diferenciam pelas linguagens utilizadas na representação do processo de desenvolvimento: uma única linguagem que representa o desenvolvimento desde a especificação até o código executável (*wide-spectrum language*), ou uma linguagem para cada fase do desenvolvimento (*narrow-spectrum language*). A abordagem DRACO para a construção de software a partir de componentes reutilizáveis é diferente dessas duas porque propõe uma linguagem diferente, chamada “linguagem de domínio”, para a descrição de aplicações em diferentes áreas [Fre87, Nei84]. Especificamente, a abordagem DRACO está relacionada com a reutilização de informação de análise e projeto.

Para que a abordagem DRACO possa ser utilizada em uma determinada área de interesse é necessário que ela tenha o conhecimento do “domínio” da área. O domínio é descrito através de objetos e operações relevantes na área de interesse, e é criado pelo analista de domínios, que examina as necessidades e os requisitos de uma coleção de sistemas na área. Esses objetos e operações são fornecidos ao projetista de domínios, que especifica diferentes implementações desse domínio em termos de outros domínios já conhecidos por DRACO.

A informação de análise é reutilizada toda vez que um sistema puder ser descrito em termos de objetos e operações de uma linguagem de domínio. Uma vez que um sistema tenha uma representação num domínio, os projetistas de sistemas podem interagir com DRACO, refinando a descrição do sistema até o código executável. Nesse refinamento são tomadas decisões entre diferentes implementações especificadas pelos projetistas de domínios, representando a reutilização de informações de projeto.

As fases de análise e projeto do domínio são as mais importantes e as mais difíceis, e produzem como resultado uma descrição do domínio que deve ser oferecida ao mecanismo DRACO. Uma descrição de domínio possui cinco partes:

Parser: ele define a sintaxe externa e a representação interna do domínio. A sintaxe externa é descrita em notação BNF estendida com mecanismos de controle, tais como recuperação de erros sintáticos. A forma interna é uma árvore com um nome de atributo e dados em cada nó. Essa forma interna contém os dados realmente manipulados por DRACO.

Prettyprinter: ele descreve como obter a sintaxe externa de todos os fragmentos de programas possíveis na forma interna. Isso é necessário para que DRACO possa interagir com os usuários na linguagem de domínio e possa identificar partes incompletas de programas.

Transformações: na abordagem DRACO elas são realizadas de fonte para fonte, isto é, na mesma linguagem de domínio. Essas transformações representam otimizações ao nível da linguagem de domínio e nunca tomam decisões de implementação.

Componentes: eles especificam a semântica do domínio e são eles que tomam as decisões de implementação, existindo um componente para cada objeto e operação

do domínio. Cada componente consiste de um ou mais “refinamentos”, que representam diferentes implementações para o objeto ou operação, e cada refinamento é uma redefinição da semântica do componente em termos de uma ou mais linguagens conhecidas por DRACO.

Procedimentos: procedimentos específicos do domínio são utilizados em circunstâncias onde uma transformação específica a ser realizada é algorítmica por natureza.

Portanto, a base da abordagem DRACO é o uso de **análise de domínios** para produzir **linguagens de domínios**, as quais podem ser **transformadas** com o propósito de otimização e implementadas através de **componentes de software**, cada qual contendo múltiplos **refinamentos**, que tomam decisões de implementação, redefinindo o problema em outras linguagens.

2.5 Avaliação da Bibliografia

O objetivo da seção 2.3.1, onde foram descritos os editores de programas orientados pela sintaxe, foi tentar mostrar a evolução cronológica da implantação de geradores de aplicação: primeiro, o desenvolvimento manual de sistemas numa área de aplicação desconhecida; segundo, a identificação, através da experiência adquirida na área, da possibilidade do uso de geradores de aplicação; e por último, a implantação definitiva de geradores de aplicação na área.

Apesar de possuírem algumas características diferentes, os geradores de editores orientados pela sintaxe seguem um mesmo processo geral de construção, mostrado na figura 2.6. Inicialmente, é dada uma especificação da linguagem de programação em forma de gramática. Essa especificação é transformada pelo gerador de aplicação em tabelas, que são utilizadas pelo núcleo, direcionando o editor segundo as características próprias da linguagem.

Figura 2.6: Visão Geral de um Editor Orientado pela Sintaxe

O núcleo é comum a todos os editores gerados. Ele define a interface com o usuário, estabelecendo primitivas que são adaptadas de acordo com a especificação da

linguagem. Nele estão incluídos comandos para a movimentação do cursor, comandos para a remoção de estruturas e o mecanismo de tradução da representação abstrata (árvore de derivação) para a representação concreta (texto).

A geração das tabelas pode ser dividida em duas partes: a geração da tabela para a sintaxe abstrata e a geração da tabela para a sintaxe concreta. Pode-se pensar, então, em dois geradores de aplicação: um gerador que aceita a descrição da sintaxe abstrata, produzindo uma tabela para essa sintaxe, e um gerador semelhante para a sintaxe concreta.

Como já foi dito, o núcleo é comum a todos os editores gerados, e isso sugere a noção de que ele é único: uma vez construído ele fica à disposição de todos os editores. Na prática, é isso o que acontece, mas, teoricamente, podem existir núcleos diferentes, com primitivas de interface diferentes.

Surge, então, a idéia de se ter geradores de aplicação que produzam núcleos de editores orientados pela sintaxe. Seria necessário identificar as partes invariantes dos núcleos e definir uma forma para especificar as primitivas de interface com o usuário. Cabe mencionar que, antes de se desenvolver tais geradores, deve-se analisar se seria vantajoso esse desenvolvimento, ou seja, até que ponto esses geradores seriam úteis, considerando o esforço necessário para desenvolvê-los.

Fazendo uma análise dos outros relatos de uso de geradores de aplicação, verifica-se que eles se restringem a gerar produtos específicos para as suas áreas de aplicação, com exceção dos meta-geradores. Apesar disso, depois de prontos, esses geradores confirmaram as suas principais vantagens: redução no custo de desenvolvimento, aumento da produtividade de código e uma boa qualidade do código produzido.

Com relação aos meta-geradores, STAGE e DRACO são mais gerais do que PG2, já que este só gera geradores de programas para o domínio de bases de dados. PG2 e STAGE são semelhantes quanto ao mecanismo de geração de código, interpretando abstrações de produtos criadas através de uma meta-linguagem. Já DRACO pode ser considerado um gerador de geradores com ênfase em reutilização através do refinamento sucessivo de domínios abstratos para domínios mais concretos.

Capítulo 3

Geradores de Aplicação: Classificação e Construção

3.1 Considerações Iniciais

Neste capítulo é proposta uma classificação para geradores de aplicação, considerando o suporte dado pelo gerador a diferentes domínios e a diferentes aplicações dentro desses domínios. Segundo essa classificação, são organizados os geradores de aplicação apresentados no capítulo 2.

Ainda, com respeito à classificação de geradores, é apresentado um gerador do tipo mais geral (suporte a vários domínios e várias aplicações), que foi o projeto desenvolvido neste trabalho. A apresentação do gerador é feita seguindo o modelo básico correspondente ao seu nível de classificação, ressaltando os aspectos que proporcionam a sua generalidade.

O capítulo está organizado da seguinte forma: na seção 3.2, é apresentada a classificação de geradores de aplicação e a organização, segundo essa classificação, dos geradores de aplicação apresentados no capítulo 2; na seção seguinte, seção 3.3, é apresentado o gerador do tipo mais geral que foi desenvolvido, com cada subseção desta apresentando um componente do gerador, segundo o seu modelo básico; e, finalmente, na seção 3.4, são apresentadas as considerações finais do capítulo.

3.2 Uma Classificação de Geradores de Aplicação

Os geradores de aplicação podem ser classificados através de dois eixos de abstração: o suporte dado a diferentes domínios de aplicação e a abrangência das aplicações que podem ser desenvolvidas com o gerador, isto é, a quantidade de diferentes tipos de aplicações que podem ser geradas. Dessa forma, tem-se a matriz mostrada na tabela 3.1,

Generalidade das Aplicações	Domínio de Aplicação	
	ÚNICO	MÚLTIPLO
ÚNICA	UDUA	UDMA
MÚLTIPLA	MDUA	MDMA

Tabela 3.1: Modelo para Classificação de Geradores

onde as siglas utilizadas têm os seguintes significados:

UDUA : Único Domínio - Única Aplicação.

MDUA : Múltiplos Domínios - Única Aplicação.

UDMA : Único Domínio - Múltiplas Aplicações.

MDMA : Múltiplos Domínios - Múltiplas Aplicações.

O adjetivo múltiplo se refere à capacidade do gerador suportar e facilitar a definição de novos domínios (múltiplos domínios) e a definição de novas aplicações dentro desses domínios (múltiplas aplicações). O termo “Domínio de Aplicação” é considerado juntamente com a técnica de especificação utilizada, ou seja, seria considerado como sendo do tipo Único Domínio um gerador que, por exemplo, pudesse gerar aplicações para os domínios de sistemas reativos e de sistemas embutidos, a partir de uma única linguagem de especificação fixa. Lançando mão de outra associação para tentar explicar melhor esse conceito, pode-se considerar que um domínio compreende todas as aplicações que poderiam ser especificadas por uma *little language*, como definido por Bentley, em [Ben86]. A necessidade de definir uma nova *little language* para especificar um problema específico indica a mudança de domínio.

A “Aplicação”, ou produto gerado pelo gerador, define o outro eixo de abstração. Alguns geradores geram apenas uma aplicação específica, isto é, um produto que tem sempre a mesma estrutura interna. É importante notar que, se o produto é um programa, não se está avaliando a sua semântica. O compilador Estelle-C, por exemplo, gera programas que implementam diferentes protocolos, mas eles são programas, essencialmente, com a mesma estrutura. Um gerador que suporta a geração de múltiplas aplicações permite a intervenção do usuário para criar produtos estruturalmente e/ou semanticamente diferentes.

As figuras 3.1 e 3.2 ilustram os quatro níveis de generalidade de geradores, classificados da maneira proposta acima, segundo o modelo geral apresentado no capítulo 2

(a): UDU

(b): UDMA

Figura 3.1: Único Domínio com Variação da Aplicação

(figura 2.2), composto de três módulos básicos: interface, analisador e gerador de produtos. Foi suposto que o resultado produzido pelo analisador de especificações é a árvore sintática de derivação de uma especificação.

No caso dos geradores UDU (figura 3.1(a)), tem-se um gerador de aplicação para um domínio específico, que produz sempre um produto de mesmo tipo. O retângulo envolvendo os dois componentes da direita do modelo indica que a principal tarefa do usuário é a de fornecer a especificação do problema, não tendo acesso ao gerador, para mudar seu comportamento. Nesta situação, o produto foi definido pelo projetista de domínios, utilizando alguma forma de definição do produto desconhecida (ou sem necessidade de ser conhecida) pelo usuário final. Neste tipo de gerador, a árvore sintática não precisa ser explicitamente criada e a descrição do produto geralmente está embutida no código do gerador.

No segundo caso, de geradores do tipo UDMA (figura 3.1(b)), o usuário, além de especificar a sua aplicação, tem a liberdade de especificar diferentes produtos dentro do domínio e, para isso, utiliza uma meta-linguagem para a definição de produtos. Essa meta-linguagem deve ser definida e projetada pelo projetista de domínios, para permitir manipulações na árvore sintática, que é onde estão as informações da aplicação. Além disso, para tornar mais poderosa essa linguagem, geralmente existem mecanismos que permitem a passagem de controle (“escape”) para linguagens de propósito geral – usualmente

(a): MUA

(b): MDMA

Figura 3.2: Múltiplos Domínios com Variação da Aplicação

aquela em que o próprio GA foi desenvolvido – para que seja possível resolver situações que a meta-linguagem não possa resolver.

Nos dois casos anteriores o domínio de aplicação foi previamente analisado e formalizado pelo analista de domínios, segundo alguma técnica de especificação. Quando já existem métodos/técnicas formais desenvolvidas para especificar aplicações dentro de um certo domínio, a tarefa do analista de domínios fica facilitada. Quando isso não ocorre, essa tarefa torna-se bem mais difícil, exigindo habilidade e experiência do analista no domínio alvo. Para ilustrar essa atividade, um experimento sobre a análise e formalização do domínio de sistemas de bibliotecas pode ser encontrado em [Lei90].

O caso envolvendo geradores do tipo MUA, ilustrado na figura 3.2(a), parece ser menos significativo, não tendo sido possível encontrar registro na literatura. Uma forte razão para isso é que a definição de domínios é tarefa difícil. Entretanto, visto sob o ângulo do usuário, este tipo de gerador poderia ser útil por oferecer várias alternativas para o usuário especificar o mesmo problema.

Uma situação hipotética para ilustrar este caso seria um gerador de aplicação que oferecesse ao usuário facilidades para redefinir os módulos de interface e analisador de especificações, para permitir a entrada de problemas de diferentes domínios de aplicação,

por exemplo: Sistemas Concorrentes especificados por Redes de Petri e Sistemas Reativos especificados por Estadogramas (*Statecharts*), e fornecesse um único resultado: um simulador do sistema especificado. É importante notar que, neste caso, não basta que a estrutura de dados intermediária seja uma simples tradução da especificação para uma forma interna, mas que a especificação seja mapeada para um modelo padronizado de referência para o gerador de produtos.

No nível mais amplo de generalidade o usuário tem liberdade para definir o domínio de aplicação e facilidade para definir diferentes aplicações dentro de cada domínio. No modelo mostrado na figura 3.2(b), mostra-se que o gerador de aplicação deve oferecer ferramentas para definição de novos domínios, e essas ferramentas podem ser, e normalmente são, um gerador do tipo UDUa desenvolvido para o Domínio de Linguagens de Especificação, formalizado através de gramáticas de vários tipos.

Assim, após formalizado o domínio de aplicação, o usuário alimenta o sistema com a especificação desse domínio e obtém um analisador para a linguagem, que reconhece especificações nesse domínio alvo e produz como resultado a estrutura de dados interna utilizada para geração de produtos pelo módulo subsequente. Os utilitários LEX e YACC, do sistema UNIX, são, com bastante frequência, utilizados na construção desses geradores de analisadores para geradores do tipo MDUA e MDMA.

Os geradores MDMA são considerados meta-geradores, já que a instanciação de um determinado domínio e de uma aplicação dentro desse domínio torna-o um gerador do tipo UDUa. Deve-se notar que a instanciação de geradores dos tipos UDMA e MDUA normalmente também resultam em geradores UDUa. Os geradores UDUa são destinados aos usuários que possuem conhecimento no domínio de aplicação, enquanto os outros tipos de geradores são mais apropriados para os analistas e projetistas de domínios, já que eles também requerem o conhecimento de técnicas de computação.

Considerando os geradores de aplicação apresentados no capítulo 2, será feita a seguir uma análise e classificação desses geradores, segundo a proposta apresentada acima. O resumo dessa classificação é dado na tabela 3.2.

Os utilitários LEX e YACC para geração, respectivamente, de analisadores léxicos e analisadores sintáticos são talvez os exemplos mais conhecidos de geradores do tipo UDUa. Dotados de rotinas semânticas adequadas, LEX e YACC podem ser usados para gerar outras aplicações (Múltiplas Aplicações), mas aqui eles serão classificados considerando a “vocação natural” original, ou seja, LEX para gerar analisadores léxicos e YACC para gerar analisadores sintáticos.

Com relação aos geradores de editores orientados pela sintaxe, o gerador ALOE parece ser específico nessa função. Já o *Synthesizer Generator* é também utilizado para gerar outras aplicações a partir de especificações de ambientes interativos através de gramáticas de atributos. Sendo assim, o gerador ALOE é classificado como sendo do tipo UDUa e o *Synthesizer Generator* como sendo do tipo UDMA.

Os geradores IMX e TTU, de programas para a extração de dados, são do tipo UDUa. IMX é específico para bases de dados IMS, e TTU é específico para fitas geradas por computadores IBM. O meta-gerador PG2, a partir do qual IMX e TTU são instanci-

	UA	MA
UD	LEX, YACC ALOE, IMX, TTU Estelle-C, EDG, TAGS Microstep, G.G.S.R., I-M-E	<i>Synthesizer Generator</i> PG2
MD	—	STAGE, DRACO GEDAI

Tabela 3.2: Classificação de Geradores Existentes

ados, é do tipo UDMA, pois ele é específico para o domínio de bases de dados, utilizando uma mesma linguagem para a descrição de diferentes tipos de organização de dados.

Outros geradores do tipo UDUUA, dentre os apresentados no capítulo anterior, são: o Compilador Estelle-C, onde Estelle é uma linguagem apropriada para especificar protocolos de comunicação e o compilador gera uma implementação do protocolo especificado; o Editor Diagramático Generalizado (EDG), do projeto AMADEUS, que gera editores diagramáticos a partir de especificações dos diagramas em forma de gramáticas de atributos; os geradores baseados em linguagens de especificação gráficas: TAGS, Microstep e Gerador de Gerenciadores de Sistemas Reativos (abreviado na tabela por G.G.S.R.); e o sistema I-M-E, que gera aplicações comerciais a partir de especificações criadas com tabelas de decisão.

Os meta-geradores STAGE e DRACO são do tipo mais geral MDMA, pois permitem a definição de vários domínios e a definição de várias aplicações dentro de cada domínio. STAGE permite essa flexibilidade através da geração dos vários componentes de um gerador específico, enquanto DRACO permite a incorporação de novos domínios e aplicações em termos de outros domínios e aplicações já definidos.

Na próxima seção será apresentado um outro gerador de aplicação do tipo MDMA, alvo deste trabalho, chamado GEDAI (GErador com Domínio e Aplicação Instanciáveis). Sendo assim, GEDAI também foi incluído na tabela 3.2 como sendo do tipo mais geral.

3.3 GEDAI: Gerador com Domínio e Aplicação Instanciáveis

Nesta seção será apresentado o gerador do tipo mais geral (MDMA) desenvolvido neste trabalho, que se chama **GEDAI** (**GE**rador com **Domínio** e **Aplicação** **I**nstanciáveis). GEDAI facilita a definição de novos domínios através da geração de analisadores de especificações para as linguagens de especificação desses domínios, e possibilita a definição de várias aplicações dentro de cada domínio através de uma meta-linguagem apropriada para a construção de abstrações de produtos de aplicação. Serão apresentados os três módulos do gerador, segundo o modelo apresentado no capítulo 2 (figura 2.2).

3.3.1 O Módulo de Interface

Para o módulo de interface, na versão atual do gerador, não foi desenvolvida nenhuma ferramenta de apoio e, sendo assim, as especificações devem ser fornecidas na forma textual. Portanto, os geradores de aplicação que forem construídos com o apoio do meta-gerador GEDAI receberão as especificações na forma textual. Mesmo assim, existem meios de se tentar melhorar a interface entre o gerador e o usuário.

Para as linguagens originalmente textuais, pode-se pensar em editores orientados pela sintaxe. Como foi visto na seção 2.3.1, eles tornam mais amigável a interface do editor, permitindo que o usuário não se preocupe com os detalhes da linguagem e, conseqüentemente, construa especificações sem erros de sintaxe. Foi vista também, na seção 2.3.1, a possibilidade de geração de tais editores e, com respeito a isso, o próprio gerador GEDAI pode servir de apoio. Teria-se, através da instanciação do meta-gerador, um gerador UDUa que, a partir da própria definição das linguagens de especificação, construísse os editores de texto orientados pela sintaxe dessas linguagens.

Outra possibilidade é ter-se módulos de interface que criem as especificações textuais através de diálogos com os usuários. O usuário, através de perguntas e respostas, ou qualquer outra forma de diálogo, especifica a aplicação e o módulo de interface produz uma representação textual, que é a entrada do gerador.

Para as linguagens diagramáticas, uma solução é ter-se um módulo de interface independente, que manipule os símbolos visuais e proporcione as características originais da linguagem, e que, depois, transforme a representação diagramática final numa representação textual equivalente, que vai servir de entrada para o gerador. Essa linguagem textual equivalente à linguagem diagramática é, na verdade, a linguagem de especificação do gerador. Neste caso, o incômodo é ter-se que definir uma linguagem textual equivalente a uma linguagem diagramática já definida e formalizada.

Os módulos de interface para linguagens diagramáticas também podem ser gerados, utilizando, por exemplo, o formalismo de gramáticas de atributos, como é o caso do ambiente AMADEUS [Mra90], apresentado na seção 2.3.1. No caso do gerador GEDAI, seria interessante que o próprio mecanismo de geração de código do meta-gerador pudesse ser utilizado na construção desses módulos de interface.

3.3.2 O Analisador de Especificações

No gerador GEDAI, a análise de uma especificação é vista como uma atividade independente, ao contrário de alguns geradores que a misturam com outras atividades como, por exemplo, a geração de código. Portanto, o resultado de uma análise de especificação deve ser uma representação intermediária fixa, que vai servir para a fase seguinte.

A representação intermediária escolhida para ser o resultado do analisador de especificações é a árvore sintática de derivação de uma especificação, segundo as regras gramaticais da linguagem¹. Essa árvore é construída durante a análise e possui como nó raiz o símbolo inicial da gramática, como nós intermediários os símbolos não terminais e como nós folhas os símbolos terminais.

Para exemplificar a estrutura da árvore e os dados armazenados em cada nó, considere a gramática abaixo, em notação BNF (Backus-Naur Form) estendida, para a linguagem de especificação de conjuntos, apresentada por Cleaveland, em [Cle88], e também apresentada na seção 2.4.2, cujas produções são:

- (1) `<especificacao> ::= <conjunto> { <conjunto> }`
- (2) `<conjunto> ::= <nome_conjunto> '{' <elemento> { ',' <elemento> } '}'`
- (3) `<nome_conjunto> ::= [a-zA-Z][a-zA-Z0-9_]*`
- (4) `<elemento> ::= [a-zA-Z][a-zA-Z0-9_]*`

onde elas indicam que uma especificação é uma sequência de um ou mais conjuntos ($\{\alpha\}$ indica zero ou mais vezes a repetição de α); um conjunto é descrito através de seu nome e seus elementos; e, o nome dos conjuntos e os identificadores dos elementos são os símbolos terminais da gramática. As produções (3) e (4) usam a notação do utilitário LEX do sistema UNIX: um identificador deve começar com uma letra, podendo ser seguida de outras letras, dígitos ou do caracter de sublinha.

Uma possível definição de conjunto, seguindo as regras gramaticais acima é:

```
cursos{matematica, estatistica, computacao}
```

e a árvore sintática resultante da análise dessa definição é:

¹No restante do texto será considerado que as linguagens de especificação são textuais.

Como se pode notar pela organização acima, a árvore sintática é uma árvore binária, representando uma árvore n-ária, onde cada nó possui um ponteiro para o primeiro filho e outro ponteiro para o próximo irmão (irmão à direita). Um nó pai representa o símbolo não terminal do lado esquerdo de uma produção e o seu primeiro filho representa o primeiro componente do lado direito da mesma produção, partindo da esquerda para a direita.

A árvore sintática acima corresponde às produções (1), (2), (3) e (4), com cada produção associada a uma subárvore desta. Os símbolos especiais da linguagem ('{', '}', '(', ')') não foram incluídos na árvore sintática pois, neste caso, eles não são significativos, mas existem casos, em outras linguagens, em que algumas palavras reservadas e símbolos especiais devem ser considerados. Nesses casos, deve-se criar um nó folha para cada palavra reservada ou símbolo especial e, adicionalmente, pode-se criar um novo nó não terminal como pai desse nó folha. Esse novo não terminal pode facilitar a identificação da palavra reservada ou símbolo especial.

Na verdade, os nós da árvore sintática não armazenam os símbolos não terminais e terminais propriamente ditos. Eles armazenam índices de duas tabelas que indicam onde encontrar os símbolos associados: uma Tabela de Estruturas Sintáticas (TES), que armazena os símbolos não terminais, e uma Tabela de Símbolos Terminais (TST). A TES é criada de acordo com as produções da gramática que define a linguagem de especificação e a TST é criada durante a análise de uma especificação.

Depois de construída, a árvore sintática de derivação de uma especificação serve de entrada para o módulo gerador de produtos criar o produto desejado. Sendo assim, além das rotinas de criação da árvore, deve existir um número suficiente de rotinas de manipulação, para que o gerador de produtos consiga, de modo eficiente, as informações necessárias. Portanto, com a estrutura dos dados armazenados em cada nó e com as rotinas de criação e manipulação, a árvore sintática pode ser considerada um Tipo Abstrato de Dados (TAD) [Aho82,Lis86]. Na tabela 3.3 listam-se as principais operações sobre o TAD árvore sintática, com os seus parâmetros e uma descrição sucinta de cada uma (maiores detalhes e o código dessas operações podem ser encontrados em [Pag91]). No apêndice A é dado o tamanho em linhas de código fonte correspondente ao TAD árvore sintática.

Além dessas rotinas, existem outras, para maior apoio ao projetista de domínios na verificação e testes de projetos de geradores. Algumas delas são: armazenamento e recuperação da árvore sintática em disco e impressão da árvore sintática. Uma outra rotina, que realiza um percurso pela árvore sintática, não foi incluída como sendo do TAD porque ela possui código específico do módulo gerador de produtos, mas será introduzida mais adiante.

As rotinas de criação da árvore sintática dependem do tipo de análise sintática escolhido: `cria_filho` e `cria_irmao` para análises sintáticas descendentes; e, `cria_pai` e `liga_irmaos` para análises sintáticas ascendentes. Os índices das tabelas são obtidos

Operação	Parâmetros	Descrição
<code>inicia_arvore</code>	<code>int ind</code> : índice do símbolo inicial da gramática na TES.	Cria a árvore pelo nó raiz. Armazena o ponteiro na variável global <code>raiz</code> .
<code>cria_filho</code>	<code>int ind</code> : índice da TES ou TST do nó a ser criado. <code>PONT *corrente</code> : nó pai do nó a ser criado.	Cria um nó filho para um dado nó. Retorna o ponteiro do nó criado.
<code>cria_irmao</code>	<code>int ind</code> : índice do nó a ser criado. <code>PONT *corrente</code> : nó irmão do nó a ser criado.	Cria um nó irmão para um dado nó. Retorna o ponteiro do nó criado.
<code>cria_pai</code>	<code>int ind</code> : índice do nó a ser criado. <code>PONT *corrente</code> : nó filho do nó a ser criado.	Cria um nó pai para um dado nó. Retorna o ponteiro do nó criado.
<code>liga_irmaos</code>	<code>PONT *irmao1</code> , <code>*irmao2</code> : irmãos a serem ligados.	Liga dois nós como irmãos. Retorna o ponteiro do primeiro nó.
<code>comprimento</code>	<code>PONT *no</code> : qualquer nó da árvore sintática	Retorna o número de irmãos do nó, incluindo o próprio nó.
<code>simbolo</code>	<code>PONT *no</code> : qualquer nó da árvore sintática <code>char *simb</code> : símbolo terminal associado ao nó.	Retorna em <code>simb</code> o símbolo terminal associado ao símbolo não terminal apontado por <code>no</code> (filho de <code>no</code>). A função retorna 0 caso não exista um símbolo terminal associado.
<code>busca_estr_sint</code>	<code>char estrutura[80]</code> : estrutura sintática da linguagem (não terminal). <code>PONT *no_pai</code> : não terminal do lado esquerdo da produção que se deseja pesquisar.	Procura a estrutura sintática no lado direito da produção cujo lado esquerdo é apontado por <code>no_pai</code> . Retorna o ponteiro do nó encontrado ou, caso contrário, o ponteiro NULL.
<code>marca_estrutura</code>	<code>PONT *no</code> : qualquer nó da árvore sintática	Marca um nó da árvore sintática, sinalizando o campo <code>marca</code> com valor 1.
<code>desmarca_estrutura</code>	<code>PONT *no</code> : qualquer nó da árvore sintática	Sinaliza o campo <code>marca</code> com valor 0.
<code>estrutura_marcada</code>	<code>PONT *no</code> : qualquer nó da árvore sintática	Retorna verdadeiro (1) se a estrutura estiver marcada e, caso contrário, retorna falso (0).

Tabela 3.3: Operações sobre o TAD Árvore Sintática

através da função:

```
pesquisa_tabela(cadeia, tabela)
```

que pesquisa, através de um valor de *hash* da cadeia de caracteres, o índice dela na tabela. Como exemplo, para se obter o índice da estrutura sintática CONJUNTO seria necessário o seguinte comando:

```
ind = pesquisa_tabela("CONJUNTO", TES);
```

Na tabela 3.3, ainda, foi mencionado o segundo campo de um nó da árvore sintática: o campo “marca”, que é utilizado para indicar se o nó a que ele pertence já foi considerado em algum momento.

Apesar dos resultados de diferentes analisadores de especificações terem uma estrutura semelhante, cada domínio possui uma linguagem de especificação própria e requer um analisador de especificações específico. O objetivo, então, é tentar facilitar ao máximo a construção desses analisadores, a partir da definição das linguagens de especificação.

No gerador GEDAI o intuito é ter-se um gerador de analisadores, do tipo UDU, que, a partir de uma gramática que defina uma linguagem de especificação, construa automaticamente um analisador de especificações para essa linguagem (figura 3.2). Esse gerador de analisadores funcionaria da seguinte maneira: a entrada seria uma gramática em notação BNF estendida de uma linguagem e a saída seria um analisador construído com o apoio dos utilitários LEX e YACC.

A diferença de se construir um analisador de especificações com o YACC ou com o gerador de analisadores é a seguinte: com o utilitário é preciso criar a gramática da linguagem de especificação na sua sintaxe e incluir nesta as chamadas das rotinas de criação da árvore sintática, nos locais apropriados; com o gerador de analisadores a idéia é gerar a mesma entrada para o YACC, já incluindo as chamadas das rotinas de criação da árvore. Portanto, o projetista de domínios não precisaria se preocupar com a construção da representação intermediária, preocupando-se somente com a definição da linguagem de especificação.

O gerador de analisadores está em fase de desenvolvimento e está sendo construído com o apoio do próprio gerador GEDAI, ou seja, ele será uma instanciação do meta-gerador. O desenvolvimento desse gerador de analisadores está em fase adiantada e a sua complementação está sendo efetivada por um projeto de Iniciação Científica.

A possibilidade de se ter esse gerador de analisadores surgiu da implementação de um analisador de especificações para um domínio específico (veja capítulo seguinte, seção 4.2.1), com o apoio dos utilitários LEX e YACC. Verificou-se na especificação de entrada para o YACC a existência de um padrão entre as regras que definem a gramática da linguagem de especificação e as chamadas das rotinas de criação da árvore sintática. Sendo assim, é possível estabelecer uma lei de formação tal que, dada uma regra gramatical, sejam obtidas automaticamente as chamadas das rotinas de criação da árvore sintática correspondentes a essa regra.

O gerador de analisadores recebe como especificação de entrada gramáticas de linguagens. Portanto, foi definida uma gramática para a própria notação BNF estendida, apresentada no apêndice D, e foi construído um analisador de gramáticas, que constrói as árvores sintáticas correspondentes às gramáticas de entrada.

O que falta ser feito, então, seguindo o modelo do gerador GEDAI, é definir a descrição de produto do gerador de analisadores e, para isso, será utilizada a meta-linguagem de descrição de produtos definida para o meta-gerador. É preciso estabelecer a tradução da notação BNF para a notação do YACC e estabelecer as leis de formação mencionadas acima. Uma questão ainda pendente é a inclusão de checagens semânticas da linguagem de especificação e uma possível solução é permitir, como no YACC, a chamada de rotinas semânticas dentro da especificação em BNF.

3.3.3 O Gerador de Produtos

Da mesma forma que a análise de uma especificação, a geração de código, no gerador GEDAI, é vista como uma atividade independente. Como já foi dito, a geração de código é realizada com o apoio de uma meta-linguagem para a descrição de produtos de aplicação. Os comandos dessa meta-linguagem estão relacionados com a gramática da Linguagem de Especificação (LE) considerada, manipulando os símbolos não terminais (estruturas sintáticas) e os símbolos terminais da gramática.

Uma Descrição de Produto é um tipo de gabarito, parecido com o produto final, onde trechos de código na linguagem alvo² estão misturados com comandos da meta-linguagem, que indicam ao gerador de produtos como buscar as informações complementares na árvore sintática. Os trechos de código na linguagem alvo são copiados para o produto final, e os comandos da meta-linguagem são interpretados e substituídos pelas informações computadas. Portanto, os comandos da meta-linguagem substituem as informações fornecidas pela especificação, e por isso a descrição de produto é construída uma única vez para cada aplicação, sendo depois utilizada na geração de todo produto semelhante.

A descrição do produto de uma aplicação deve ser construída pelo projetista de domínios depois que o domínio estiver definido pelo analista de domínios, juntamente com a gramática da linguagem de especificação. Além disso, o projetista de domínios deve conhecer a estrutura dos produtos a serem gerados, para poder identificar as partes dependentes e as partes independentes da especificação.

A meta-linguagem para definição de produtos projetada para o gerador GEDAI chama-se Linguagem de Descrição de Produtos (LDP). Os seus comandos são diferenciados dos trechos de código na linguagem alvo pois, como na meta-linguagem utilizada pelo gerador STAGE, eles são prefixados pelo caractere '%'. O interpretador da LDP é o responsável pela geração de código e corresponde, assim, ao módulo Gerador de Produtos.

Uma descrição de produto é dividida em duas partes: a parte de definição de funções e o módulo principal da descrição de produto. Todas as funções devem ser

²A linguagem alvo é a linguagem na qual o produto é implementado.

definidas antes do módulo principal, que possui a seguinte forma:

```
%gabarito identificador  
declaração_de_variáveis;  
comando_composto  
%fimgab
```

onde o identificador é usado apenas para nomear a descrição de produto e a parte de declaração de variáveis é opcional. No bloco de comandos estão mesclados os comandos da LDP e os trechos de código na linguagem alvo.

No restante desta seção será feita uma descrição da linguagem LDP, incluindo a declaração de variáveis, os seus comandos e a definição de funções. Uma descrição mais detalhada pode ser encontrada no Manual da Linguagem de Descrição de Produtos [Mei91A]. A sintaxe da LDP também é apresentada no apêndice B. Para exemplificar os comandos da LDP e verificar como eles estão relacionados com a gramática de uma LE, será considerado novamente o exemplo da linguagem de especificação de conjuntos, considerando a gramática apresentada na seção anterior.

Declaração de Variáveis

Os tipos de dados permitidos na LDP são três:

%INT: tipo inteiro.

%NO: tipo nó, que representa um nó da árvore sintática, ou seja, uma estrutura sintática da LE. Existem duas palavras chaves da LDP que são constantes do tipo nó: **%TOPO**, que representa a raiz da árvore sintática; e **%NULO**, que representa o ponteiro nulo.

%SIMBOLO: tipo símbolo, que representa um símbolo terminal ou uma constante do tipo símbolo. Uma constante do tipo símbolo é qualquer cadeia de caracteres entre aspas.

Os nomes das variáveis devem ser diferentes dos nomes das estruturas sintáticas da LE; toda variável deve ser declarada antes de ser usada; e, a sintaxe de uma declaração exige o tipo antes da lista de variáveis. Abaixo, é mostrado um exemplo da declaração de duas variáveis: uma do tipo inteiro e outra do tipo nó.

```
%INT contador; %NO estr_sintatica;
```

Busca de uma Estrutura Sintática

Existe um comando do tipo expressão que, dada uma produção da gramática e uma estrutura sintática da LE, busca a ocorrência da estrutura no lado direito da

produção. A sintaxe do comando é:

estrutura_sintática(expressão)

onde *estrutura_sintática* é qualquer não terminal da gramática e o resultado da *expressão* deve ser o não terminal do lado esquerdo da produção. Quando a produção tiver, no lado direito, mais de uma estrutura sintática do mesmo tipo, é retornada a primeira ocorrência da estrutura.

Por exemplo, a expressão `nome_conjunto(c)` retorna a estrutura sintática do tipo `NOME_CONJUNTO` da produção que tem como lado esquerdo o não terminal “c”, (produção (2) da gramática da seção 3.3.2), onde a variável “c” representa a estrutura sintática `CONJUNTO`. Convencionou-se usar o identificador de uma estrutura sintática da seguinte maneira: em maiúsculo, quando ele identificar o tipo da estrutura sintática, e em minúsculo, quando ele aparecer num comando do tipo descrito acima.

Cada domínio de aplicação possui uma LE própria e, conseqüentemente, as gramáticas das LEs possuem estruturas sintáticas distintas. Portanto, esses comandos de busca são diferentes para cada LE, dependendo das estruturas sintáticas que são armazenadas na TES.

Símbolo Terminal

Um dos comandos mais importantes da LDP é o comando, do tipo expressão, que retorna o símbolo terminal associado à produção que deriva esse mesmo símbolo. A sintaxe do comando é:

%simb(expressão)

onde o resultado da avaliação da *expressão* deve ser o não terminal do lado esquerdo da produção que deriva o símbolo terminal.

Por exemplo, o comando `%simb(nome_conjunto(c))` retorna o símbolo terminal correspondente ao nome do conjunto “c” (produção (3)).

Comando de Impressão

Durante a interpretação de uma descrição do produto é necessário que exista um comando que transfira o que foi computado no programa em LDP para o produto final. Esse comando é o comando de impressão:

%impr(expressão)

que transfere para o produto o resultado da *expressão*, que deve ser um número inteiro ou um símbolo (uma cadeia de caracteres).

Exemplos de comandos desse tipo são:

`%impr(10)`

```
%impr("Simbolo")
%impr(%simb(nome_conjunto(c)))
```

onde o último exemplo coloca na saída o nome do conjunto "c".

Número de Estruturas Sintáticas

Além de buscar uma estrutura sintática derivada de uma produção, às vezes, é útil se saber quantas estruturas sintáticas de um certo tipo são derivadas por uma produção. Na LDP existe o comando do tipo expressão:

```
%comp(expressão)
```

que calcula a quantidade de estruturas sintáticas do tipo retornado pela avaliação da *expressão*.

Esse comando é particularmente útil para as produções que derivam um número variável de estruturas sintáticas do mesmo tipo. Por exemplo, a produção (2) deriva um número variável de estruturas sintáticas do tipo ELEMENTO. Sendo assim, para se calcular o número de elementos de um conjunto tem-se o comando:

```
%comp(elemento(c))
```

Comando de Repetição

O comando de repetição da LDP tem a seguinte sintaxe:

```
%paratodo identificador:estrutura_sintática %repita
    comando_composto
%fimpt
```

Esse comando realiza um percurso pela árvore sintática visitando todos os nós do tipo *estrutura_sintática* (as estruturas sintáticas correspondem aos nós intermediários e os símbolos terminais aos nós folhas), e repete a execução do *comando_composto* para cada nó visitado. Por exemplo, o comando:

```
%paratodo c:CONJUNTO %repita
    %impr(%simb(nome_conjunto(c)))
%fimpt
```

percorre a árvore, visitando todos os nós do tipo CONJUNTO, e produz na saída os nomes dos conjuntos.

O percurso pode ser restrito a apenas uma subárvore da árvore sintática e, nesse caso, o comando tem uma sintaxe um pouco diferente:

```
%paratodo identificador:estrutura_sintática %em expressão %repita  
    comando_composto  
%fimpt
```

onde o percurso é realizado a partir da subárvore que tem como raiz o nó resultante da avaliação da *expressão*.

Comandos de Atribuição e Decisão

A atribuição de valor a uma variável pode ser feita de duas formas:

```
%seja variável = expressão;
```

ou simplesmente

```
%variável = expressão;
```

sendo que, na última, a primeira letra do identificador da variável deve seguir imediatamente o '%', isto é, não podem existir espaços em branco entre o '%' e a variável. A escolha de uma das duas opções depende da preferência do programador (projetista de domínios), valendo observar que a segunda é mais concisa.

O único comando da LDP para tomar decisões é o comando:

```
%se expressão %entao  
    comando_composto-1  
%senao  
    comando_composto-2  
%fimse
```

onde a parte *%senao* é opcional. A *expressão* é avaliada e se for “verdadeira” (se tiver valor diferente de zero) o *comando_composto-1* é executado. Se for “falsa” (valor igual a zero) e se houver a parte *%senao* o *comando_composto-2* é executado. Todo comando *%se-%entao-%senao* deve terminar com a palavra chave *%fimse*.

Marcar e Desmarcar uma Estrutura Sintática

Quando se realizam vários percursos simultâneos na árvore sintática (comandos de repetição aninhados), pode ser desejável que alguns nós que foram considerados num percurso anterior não sejam considerados num próximo percurso. Na LDP existem os comandos para marcar e desmarcar uma estrutura sintática, cujas sintaxes são, respectivamente:

```
%marca_estrutura(expressão)  
%desmarca_estrutura(expressão)
```

onde o resultado da *expressão* deve ser a estrutura que se deseja marcar ou desmarcar.

Como o próprio nome diz, marcar uma estrutura sintática significa sinalizar essa estrutura, indicando que ela já foi considerada em algum momento. Para se saber se uma dada estrutura está marcada ou não, existe um comando do tipo expressão que retorna verdadeiro se a estrutura estiver marcada e, em caso contrário, retorna falso. Esse comando é:

```
%estrutura_marcada(expressão)
```

onde o resultado da *expressão* deve ser a estrutura que se deseja testar. Para se poder considerar novamente uma estrutura sintática, basta desmarcá-la.

Funções

Na LDP também é permitida a definição de funções. A forma geral de uma função é:

```
nome_da_função(lista_de_parâmetros)  
    corpo_da_função  
%fimfunc
```

As funções são importantes porque permitem modularizar uma descrição de produto tornando-a mais compreensível, e com elas não é preciso repetir um mesmo trecho de código em vários pontos. Elas devem ser definidas antes do módulo principal da descrição de produto. No corpo de uma função estão declaradas as suas variáveis locais e está definido o fluxo de controle interno dela.

A *lista_de_parâmetros* de uma função, se não for vazia, consiste de uma ou mais declarações de parâmetros, separadas por ponto e vírgula. Uma declaração de parâmetros deve ter o tipo dos parâmetros, seguido de um ou mais nomes de parâmetros, separados por vírgula.

O comando para chamar uma função tem duas formas:

```
%chame nome_da_função(lista_de_argumentos)
```

ou simplesmente

```
%nome_da_função(lista_de_argumentos)
```

valendo as mesmas considerações feitas para o comando de atribuição, mas quando uma função é chamada no meio de uma expressão não é preciso prefixá-la. A passagem dos argumentos na LDP é feita por “valor”, isto é, mudanças causadas nos parâmetros de uma função não afetam as variáveis usadas para chamá-la.

Existem dois modos para terminar a execução de uma função e retornar ao ponto de chamada: primeiro, quando o último comando da função for executado; e segundo,

quando for encontrado um comando de retorno. A sintaxe do comando de retorno é:

```
%retorna(expressão)
```

onde, se a *expressão* for vazia, causa apenas a saída imediata da função. O tipo do valor retornado pelo comando vai depender da avaliação da *expressão*. Uma função pode ter vários comandos de retorno, desde que eles retornem valores do mesmo tipo.

Rotinas de Escape para a Linguagem C

Em certas situações é possível que os comandos oferecidos pela LDP não sejam suficientes para computar algum resultado desejado. Nesses casos, a LDP oferece um recurso bastante flexível, que permite a passagem de controle para a linguagem C. Pode-se construir uma rotina em C que compute o resultado desejado, à parte da descrição do produto, e chamá-la através de um comando da LDP. O controle da execução é passado para a rotina e retomado ao final de sua execução.

O comando da LDP que passa o controle para a rotina de escape é do tipo *expressão* e tem como resultado o valor retornado pela rotina. A sintaxe do comando é:

```
%escape(tipo_da_função nome_da_função(lista_de_argumentos))
```

onde é necessário especificar o tipo do valor retornado pela função. Na lista de argumentos também é necessário especificar os tipos dos argumentos passados para a função. O tipo do argumento deve preceder a expressão que resulta no valor do argumento. Um exemplo hipotético de chamada de uma rotina de escape com dois argumentos é:

```
%var = %escape(%NO rot_exemplo(%NO arg1, %INT arg2));
```

onde a variável *var* do tipo nó recebe o valor retornado pela rotina *rot_exemplo*.

O programador da rotina de escape deve ficar atento para que o valor retornado seja um dos tipos aceitos pela LDP. Uma rotina, por exemplo, não pode voltar um número real. O programador da rotina deve, também, conhecer a estrutura dos nós da árvore sintática e suas rotinas de manipulação, pois é dela que ele vai conseguir as informações a respeito da especificação.

Aspectos da Implementação

O interpretador da LDP foi construído com base no interpretador C apresentado por Herbert Schildt, em [Sch89], e também foi implementado na linguagem C. Na figura 3.3 é apresentada a arquitetura do interpretador da LDP. No apêndice A são apresentados aspectos mais gerais da implementação, como o ambiente de desenvolvimento e o tamanho em linhas de código fonte.

O analisador léxico para o interpretador, responsável pela identificação dos símbolos lógicos (*tokens*) no código fonte, reconhece os tipos de *tokens* mostrados na tabela 3.4. Um *token* é reconhecido como uma estrutura sintática quando ele pertence à

Figura 3.3: Arquitetura do Interpretador da LDP

Tipo do <i>Token</i>	<i>Tokens</i>
delimitador	símbolos de pontuação e operadores
palavra_reservada	palavras reservadas da LDP
estrutura_sintática	estruturas sintáticas da LE (TES)
identificador	variáveis e nomes de funções
número	constantes numéricas inteiras

Tabela 3.4: Categoria dos Símbolos Lógicos da LDP

Tabela de Estruturas Sintáticas (TES) da Linguagem de Especificação (LE).

O *parser* de um interpretador, diferente do *parser* de um compilador, cuja função é gerar o código objeto correspondente ao código fonte, avalia o programa fonte em tempo de análise. O *parser* do interpretador da LDP utilizou o método de análise sintática descendente recursivo, que foi implementado por um conjunto de funções recursivas associadas a cada não terminal da gramática da LDP.

Uma informação importante para um interpretador é conhecer a localização de cada função definida num programa, para que a chamada de uma função seja realizada o mais rápido possível, sem a necessidade de se fazer uma busca sequencial para encontrar o ponto de entrada da função. Um meio eficiente de se conseguir essa informação é utilizar um pré-processador, chamado *prescan*, que percorre o código fonte, antes da execução, armazenando o ponto de entrada de cada função definida. O *prescan* para a LDP está embutido no próprio código do interpretador e percorre o código fonte apenas até o início do módulo principal, pois todas as funções devem ser definidas antes da palavra chave

%gabarito.

A execução de um programa LDP começa no ponto de parada do *prescan*, no início do módulo principal, e é baseada em blocos. A função principal do interpretador (`interp_bloco()`) – que decide qual ação a ser tomada, baseada no próximo *token* da entrada – é chamada cada vez que um novo bloco é encontrado, e estes incluem chamadas de funções e blocos de comandos, tais como os blocos `%se` e `%paratodo`. Dentro da função `interp_bloco()`, um comando repetitivo primeiro copia no produto final a parte invariante escrita na linguagem alvo, até que seja encontrado um comando da LDP, identificado pelo caractere '%', que então é interpretado. Esse comando repetitivo termina quando for encontrado o final do bloco de comandos para o qual foi chamada a função `interp_bloco()`.

As chamadas das rotinas de escape são realizadas por funções em C, que devem ser criadas antes da interpretação da descrição de produto. Essas funções são geradas por um outro pré-processador da LDP (veja capítulo seguinte, seção 4.4), a partir da própria descrição de produto, e devem ser incluídas no arquivo de rotinas de escape. O interpretador da LDP, então, quando identifica uma chamada de rotina de escape, passa o controle para a função em C correspondente à rotina e esta realiza a chamada.

3.4 Considerações Finais

No início do capítulo foi apresentada a classificação de geradores de aplicação proposta neste trabalho, que é baseada no suporte dado pelo gerador na definição de novos domínios e na especificação de novas aplicações dentro desses domínios. Além disso, foi ilustrado o modelo básico de cada nível de generalidade da classificação, segundo o modelo geral apresentado no capítulo 2. Ainda, com respeito à classificação, permitiu-se organizar e agrupar os geradores de aplicação também apresentados no capítulo 2.

No restante do capítulo foi apresentado o gerador de aplicação desenvolvido neste trabalho, que é classificado, segundo a classificação proposta, como sendo do nível mais alto de generalidade. Sendo assim, permite-se que o gerador seja instanciado para domínios e aplicações diferentes, tornando-se um gerador específico para um dado domínio e uma certa aplicação dentro desse domínio.

O gerador GEDAI foi desenvolvido considerando que é mais importante construir-se geradores rapidamente do que construir geradores eficientes. Portanto, GEDAI foi projetado para permitir aos projetistas de domínios construir geradores facilmente, embora esses geradores possam ser ineficientes em termos do uso de espaço ou tempo (eles geralmente utilizam grandes árvores sintáticas e realizam vários percursos nessas árvores).

Fazendo uma comparação entre o gerador GEDAI e os outros geradores do tipo mais geral MDMA apresentados, STAGE e DRACO, dois pontos podem ser destacados:

- STAGE possui um processo de automatização, na geração de geradores de aplicação, bem superior à instanciação de GEDAI, mas, com as extensões previstas, como, por exemplo, o gerador de analisadores de especificações, espera-se que GEDAI torne-se

uma ferramenta semelhante ao gerador STAGE.

- DRACO realiza a tradução da especificação para a implementação através de refinamentos sucessivos de domínios mais abstratos para domínios mais concretos, enquanto GEDAI realiza essa tradução em um único passo, transformando diretamente a especificação de entrada na implementação final. Além disso, DRACO é baseado na reutilização de componentes de software, enquanto GEDAI é baseado na exploração de técnicas de compilação a níveis mais abstratos.

Capítulo 4

Instanciação do Meta-gerador GEDAI

4.1 Considerações Iniciais

Neste capítulo são apresentados exemplos da instanciação do gerador GEDAI apresentado no capítulo anterior, ou seja, são estabelecidos, com a utilização dos mecanismos proporcionados pelo meta-gerador, domínios específicos e aplicações específicas dentro desses domínios, que tornam o gerador do tipo mais geral num gerador do tipo mais específico.

Instanciar o gerador GEDAI é, basicamente, definir a linguagem de especificação para o domínio, e para isso, dependendo do domínio, pode-se utilizar métodos e técnicas já existentes, e definir o produto a ser gerado, utilizando a meta-linguagem LDP, projetada para o meta-gerador e própria para a descrição de produtos de geradores de aplicação.

O capítulo está organizado da seguinte forma: na seção 4.2, é apresentado um gerador de aplicação para o domínio de Sistemas Reativos¹; na seção 4.3, é apresentado um gerador para o domínio da Teoria dos Conjuntos; na seção 4.4, são apresentadas ferramentas de apoio ao gerador GEDAI, que foram construídas com a instanciação do próprio meta-gerador; na seção 4.5, são apresentados os passos necessários na instanciação do GEDAI; e, finalmente, na seção 4.6, são apresentadas as considerações finais do capítulo.

4.2 Um Gerador de Aplicação para Sistemas Reativos

Como já foi dito, apesar da construção de geradores de aplicação ser uma tarefa complicada, o passo mais difícil do desenvolvimento de geradores é reconhecer onde poder usá-los. Adicionalmente, é importante reafirmar que a existência de métodos/técnicas

¹Este gerador de aplicação para Sistemas Reativos também é apresentado em [Mei91B].

formais para especificar aplicações dentro de um domínio alvo facilita o uso de geradores de aplicação, como é o caso do domínio apresentado nesta seção. Quando isso não ocorre, há a necessidade prévia de se analisar e formalizar o domínio, o que não é tarefa simples.

Um possível domínio de aplicação são os Sistemas Reativos. Um sistema reativo é caracterizado por ser orientado a eventos, ou seja, está continuamente respondendo a estímulos do meio externo e interno. Redes de Comunicação, Sistemas de Telefonia, Sistemas Operacionais, Sistemas para Aviação e Interfaces Homem-Máquina são alguns exemplos de sistemas reativos.

Uma das técnicas propostas para a especificação de sistemas reativos são os *Statecharts*² [Har87A]. A técnica Statechart (do inglês *State* = estado e *chart* = diagrama) é uma extensão dos diagramas de estados convencionais, incluindo noções de hierarquia, paralelismo e comunicação. Uma especificação baseada em Statecharts descreve o comportamento de um sistema em função dos estados que ele possui e dos eventos que causam as mudanças entre os estados, onde os estados são representados visualmente como bolhas conectadas por arcos, que representam as transições.

Para exemplificar a utilização dos Statecharts e introduzir informalmente os seus principais conceitos, é apresentada na figura 4.1 a especificação parcial do mostrador de um relógio de pulso digital. O mostrador é representado por onze estados em diferentes níveis hierárquicos, expressos pelo encapsulamento dos estados. Os estados *Normal* e *Volta* são duas formas de exibição do cronômetro e *Zero* é um estado no qual o cronômetro está desligado e o visor apresenta os dígitos com valor zero. Os eventos “a”, “b” e “c” correspondem a pressionar os pinos externos do relógio.

O cronômetro ou está no seu estado inicial (*Zero*) – representado visualmente por uma seta com um círculo na origem, apontando para o estado – ou está em *Operação*, não podendo estar nos dois ao mesmo tempo. Diz-se, assim, que os estados *Zero* e *Operação* são uma decomposição OU (*XOR*) do estado *Cronômetro*, da mesma forma que *Normal* e *Volta* o são de *Display*.

Num Statechart, dois subestados de um mesmo estado podem estar ativos ao mesmo tempo, caracterizando o paralelismo ou ortogonalidade. Uma decomposição E (*AND*) é aquela na qual estando num estado, o sistema deve estar em todos os seus componentes. No cronômetro, *Display* e *Contador* são uma decomposição ortogonal de *Operação*, significando que, após o cronômetro ser acionado, o tempo passa a ser mostrado no visor e contado simultaneamente. O paralelismo é representado visualmente por uma bolha na qual os componentes estão separados por uma linha tracejada.

Na figura 4.1 notam-se, também, as transições que ocorrem entre os estados, representadas pelos arcos direcionados. Se o cronômetro estiver acionado, no estado *Ligado*, e pressionar-se o botão “a”, vai-se para o estado *Desligado*, e vice-versa. Toda transição parte de um ou mais estados origens e chega a um ou mais estados destinos.

Uma outra maneira de se atingir um grupo de estados é através da “história” desse grupo, visualmente representada pela letra H inserida em um pequeno círculo. Ativar um determinado estado dentro de um grupo de estados que possui história significa

²Também chamados de Estadógramas. No restante do texto será utilizado o termo original em inglês.

Figura 4.1: Statechart do Mostrador de um Relógio Digital

considerar aquele estado que foi o mais recentemente visitado. A história se aplica ao nível no qual ela aparece ou, a partir dele, a todos os níveis mais internos do diagrama. Essa situação é representada visualmente pelo símbolo “*” associado ao símbolo de história.

Por exemplo, suponha-se que o *Cronômetro* tenha sido ativado e tenha entrado em *Operação* e, depois, tenha sido desativado, com os subestados *Normal* e *Desligado* sendo os estados atômicos mais recentemente visitados. Quando o *Cronômetro* for ativado novamente, através do evento “c”, esses mesmos estados serão considerados os estados atômicos ativos.

O conceito de história pode estar, ainda, associado ao conceito de estado inicial (*default*), como mostra a figura 4.1, ativando-se o estado *default* (*Zero*), se for a primeira visita ao estado (*Cronômetro*), ou ativando-se o estado mais recentemente visitado em caso contrário.

Uma outra característica dos Statecharts é que certas condições podem estar associadas aos eventos. Por exemplo, na figura 4.1, a condição *in(Desligado)* obriga que a transição que leva do estado *Normal* para o estado *Zero*, só dispare se o estado *Desligado* estiver ativo no instante em que ocorrer o evento “b”. Outros tipos de condições podem ser encontradas em [Har87B].

Nos Statecharts a reatividade do sistema não é expressa simplesmente por mudanças nos estados internos do sistema. É possível associar ações e atividades aos eventos. Uma ação é executada instantaneamente ao se entrar e/ou sair de um determinado estado e uma atividade é caracterizada pela sua execução enquanto o sistema permanecer no estado associado.

O gerador de aplicação para o domínio de Sistemas Reativos, que será apresentado a partir de agora, recebe como entrada uma especificação em Statechart e produz como saída um produto relacionado com o sistema especificado. No caso específico desse gerador, o produto a ser gerado é o programa que simula o sistema, realizando as mudanças de estados causadas pelos eventos, e executando as ações e atividades associadas a esses eventos. O gerador se chama **Gerador de Simuladores de Statecharts (GSS)**.

O gerador GSS é uma instanciação do meta-gerador GEDAI e, portanto, de acordo com o modelo MDMA da figura 3.2, deve possuir um Módulo de Interface, um Analisador de Especificações fornecidas em Statecharts, uma Descrição de Produto do simulador de Statecharts e, caso seja necessário, poderá incluir rotinas de escape para a linguagem C. A arquitetura do GSS pode ser vista na figura 4.2.

Figura 4.2: Arquitetura do Gerador de Simuladores de Statecharts (GSS)

Como, no gerador GEDAI, não existe nenhuma ferramenta de apoio à construção do módulo de interface, e para não perder o aspecto visual da técnica Statechart, aproveitou-se o Editor Gráfico de Statecharts (EGS) [Bat91], que está integrado a um simulador geral de Statecharts [For91]. Ao EGS adicionou-se uma função especial, que transforma uma representação gráfica de Statechart na representação textual equivalente, que serve de entrada para o GSS. Com isso proporciona-se uma interface mais amigável entre o gerador e o usuário, apesar da especificação ser fornecida ao gerador na forma textual. Nas próximas duas seções serão apresentados os outros componentes do gerador GSS.

4.2.1 A Linguagem de Especificação e o Analisador de Especificações

A linguagem textual definida para descrição de Statecharts chama-se Linguagem de Especificação de Statecharts (LES) e, da mesma forma que os diagramas, descreve o comportamento de um sistema em função dos estados e eventos. Ela é composta de três seções principais: a seção de declaração de variáveis, a seção de declaração de estados e a seção de declaração de transições. A primeira seção, opcional, permite a declaração de variáveis e seus tipos; a segunda descreve todos os estados, indicando, para cada estado, o seu nome, o tipo de decomposição e, dependendo da decomposição, os subestados e o

subestado inicial; e, por último, a terceira seção especifica as transições entre os estados, incluindo situações de história, condições, ações, etc. No apêndice C são apresentadas as produções da gramática que define a sintaxe completa da LES.

Considerando o Statechart do mostrador (figura 4.1), a sua descrição textual equivalente é a seguinte:

```
ESTADO Mostrador OU
    SUBESTADOS DEF Hora, Cronometro
ESTADO Hora ATOMO
ESTADO Cronometro OU
    SUBESTADOS DEF/H* Zero, Operacao
ESTADO Zero ATOMO
ESTADO Operacao E
    SUBESTADOS Display, Contador
ESTADO Display OU
    SUBESTADOS DEF Normal, Volta
ESTADO Contador OU
    SUBESTADOS DEF Ligado, Desligado
ESTADO Normal ATOMO
ESTADO Volta ATOMO
ESTADO Ligado ATOMO
ESTADO Desligado ATOMO;
EVENTO c
    ORIGEM Hora
    DESTINO Cronometro
EVENTO c
    ORIGEM Cronometro
    DESTINO Hora
EVENTO a
    ORIGEM Zero
    DESTINO Operacao
EVENTO b[in(Ligado)]
    ORIGEM Normal
    DESTINO Volta
EVENTO b[in(Desligado)]
    ORIGEM Normal
    DESTINO Zero
EVENTO b
    ORIGEM Volta
    DESTINO Normal
EVENTO a
    ORIGEM Ligado
    DESTINO Desligado
```

EVENTO a
ORIGEM Desligado
DESTINO Ligado;

O analisador de especificações do gerador GSS, então, a partir de uma entrada semelhante à apresentada acima, deve gerar a sua árvore sintática, de acordo com as regras da gramática da LES. Atualmente existem duas versões do analisador de especificações.

Numa primeira etapa, foram aproveitados os analisadores léxico e sintático de um analisador da própria LES, para um simulador geral de Statecharts [For91]. Esse analisador recebe especificações na linguagem LES e produz estruturas de dados que são utilizadas para a simulação. O analisador de especificações do GSS foi obtido retirando-se as rotinas de criação dessas estruturas de dados e substituindo-as pelas rotinas de criação da árvore sintática, nos locais apropriados.

Depois, numa segunda etapa, foi construído um novo analisador de especificações para o GSS, com o apoio dos utilitários LEX e YACC. Esse analisador foi construído da seguinte maneira: primeiro, a gramática da LES, em notação BNF, foi traduzida para a forma aceita pelo YACC; e, depois, foram incorporadas as rotinas de criação da árvore sintática.

Na primeira etapa, os objetivos foram a construção do TAD árvore sintática, identificando as rotinas de criação necessárias, e a identificação dos pontos de chamadas dessas rotinas, de acordo com as regras da gramática da LES. Na segunda etapa, o objetivo foi a generalização dos analisadores de especificações, tendo como ponto de partida uma especificação de entrada do LEX e uma especificação de entrada do YACC de um analisador de especificações específico, para servir como base na geração desses analisadores para outros domínios (veja capítulo anterior, seção 3.3.2).

Os dois analisadores, para uma dada especificação em LES, constroem a mesma árvore sintática de derivação, já que eles seguem as regras de uma mesma gramática. O primeiro realiza uma análise sintática descendente recursiva, construindo a árvore sintática a partir da raiz e em direção aos nós folhas. O segundo, construído com o YACC, realiza uma análise ascendente, para gramáticas do tipo LALR(1), criando a árvore a partir dos nós folhas.

Uma comparação entre os dois analisadores de especificações pode ser vista pelos seguintes pontos observados:

- o analisador gerado pelo YACC é mais eficiente, pelo método utilizado.
- o código executável do analisador descendente recursivo é menor.
- a especificação de entrada do YACC é mais fácil de ser feita do que ter-se que implementar o método descendente recursivo. Apesar de não ter implementado o analisador descendente recursivo para a LES, o autor deste trabalho teve experiências anteriores na construção de compiladores, utilizando o método descendente recursivo.

- a especificação de entrada do YACC é bem mais legível e compreensível do que o código em C do analisador descendente recursivo.
- a especificação de entrada do YACC é bem mais fácil de ser modificada do que o código do analisador descendente recursivo.

Todas essas vantagens apontadas para o analisador de especificações construído com o apoio do utilitário YACC foram obtidas pelo fato do próprio YACC ser um gerador de aplicação. Apenas o código gerado pelo YACC é maior do que o código do analisador descendente recursivo, mas isto decorre do método utilizado e de funções adicionais que são introduzidas no código e não foram utilizadas. No apêndice A também é feita uma comparação entre o tamanho em linhas de código fonte da primeira versão e o da segunda versão (entradas para o LEX e YACC).

Além de criar a árvore sintática de derivação de uma especificação em LES, os dois analisadores realizam checagens semânticas, como por exemplo: verificar se uma variável foi declarada, verificar se um subestado não é declarado como subestado de um outro estado, verificar se todos os subestados também foram declarados como estados, verificar se os estados origens e destinos das transições são estados especificados, etc.

4.2.2 O Produto e a Descrição de Produto

Os produtos a serem gerados pelo GSS são programas simuladores de Statecharts que especificam o aspecto comportamental de certos sistemas. A estrutura desses simuladores, necessária para se identificar as partes dependentes e independentes da especificação, pode ser obtida com a implementação do simulador de um Statechart específico. É importante notar que esse simulador deve possuir todas as características dos Statecharts que se queira incorporar nos produtos.

A seguir, é parcialmente apresentada uma possível implementação do simulador do Statechart do mostrador, escrita na linguagem C, mostrando alguns aspectos da estrutura de dados e da estrutura de controle do programa. Os estados são representados por um tipo enumerado e a hierarquia de estados, para maior rapidez de acesso, é implementada como uma tabela de tamanho fixo, com a última linha contendo apenas valores nulos, implementando uma estrutura de árvore com diversos ponteiros, onde cada nó representa um estado. O arranjo mostrado abaixo representa a hierarquia do mostrador, indicando que o sistema ainda não foi acionado (não existem estados ativos).

```
enum Estados{ MOSTRADOR, HORA, CRONOMETRO, ZERO, OPERACAO, DISPLAY,
               CONTADOR, NORMAL, VOLTA, LIGADO, DESLIGADO, NULO };
```

```
typedef struct Componentes{
    enum Estados nome_est;
    int prim_filho;
    int pai;
    int prox_irmao;
```

```

        int ant_irmao;
        int ligado;
        int ult_lig;
        int tipo;
        int defo;
        int nivel;
    }Estado;

Estado arvore_estado[] = {
    {MOSTRADOR,HORA,NULO,NULO,NULO,n_lig,n_lig,xor,n_def,0},
    {HORA,NULO,MOSTRADOR,CRONOMETRO,NULO,n_lig,n_lig,atomo,def,1},
    {CRONOMETRO,ZERO,MOSTRADOR,NULO,HORA,n_lig,n_lig,xor,n_def,1},
    {ZERO,NULO,CRONOMETRO,OPERACAO,NULO,n_lig,n_lig,atomo,def_hh,2},
    {OPERACAO,DISPLAY,CRONOMETRO,NULO,ZERO,n_lig,n_lig,and,n_def,2},
    {DISPLAY,NORMAL,OPERACAO,CONTADOR,NULO,n_lig,n_lig,xor,n_def,3},
    {CONTADOR,LIGADO,OPERACAO,NULO,DISPLAY,n_lig,n_lig,xor,n_def,3},
    {NORMAL,NULO,DISPLAY,VOLTA,NULO,n_lig,n_lig,atomo,def,4},
    {VOLTA,NULO,DISPLAY,NULO,NORMAL,n_lig,n_lig,atomo,n_def,4},
    {LIGADO,NULO,CONTADOR,DESLIGADO,NULO,n_lig,n_lig,atomo,def,4},
    {DESLIGADO,NULO,CONTADOR,NULO,LIGADO,n_lig,n_lig,atomo,n_def,4},
    {NULO,NULO,NULO,NULO,NULO,n_lig,n_lig,atomo,n_def,0}
};

```

Dentre os campos de dados armazenados em cada linha da tabela estão os seguintes:

ligado: indica se o estado faz parte da configuração, isto é, está ativo.

ult_lig: armazena a história de ativações do estado.

tipo: indica o tipo de decomposição: AND, OR ou ATOMO.

defo: indica se o estado é *default* ou não.

nivel: indica o nível de profundidade (altura) do estado.

Na estrutura de controle do programa são realizadas as mudanças entre os estados. O programa principal indica que, primeiramente, é constituída a configuração inicial descendo-se na hierarquia pelas entradas *defaults*. A seguir é executado um comando repetitivo onde, a cada iteração, é obtido o próximo evento, e, em função do mesmo, é analisada a configuração para ver se esta é afetada. A configuração dos estados ativos é salva em uma lista antes de cada passo da simulação, para que, durante a simulação, cada estado dessa configuração seja considerado, não confundindo com os estados ativos da nova configuração sendo formada.

```

main(){
    enum Eventos evento ;

    cabeca = (struct List *)malloc(sizeof(Lista));
    cabeca->proximo = NULL;
}

```

```

cabeca->nome_estado = NULO;
Ativa_Prole(MOSTRADOR); /* Configuracao inicial,por default */
while (evento != FIM){
    Exibe_Ativos(MOSTRADOR);
    evento = Ler_Eventos();
    Salvar_Conf_Ativos();
    Simular(evento);
}
}

```

No código da função *Simular*, apresentado abaixo, estão incorporadas as transições entre os estados em função dos eventos definidos na especificação. Um determinado estado ativo é selecionado e verifica-se se ele é afetado pelo evento corrente sendo analisado. Essa verificação é feita por uma estrutura de comando *switch* em dois níveis: no primeiro é selecionado o código correspondente ao estado e no segundo o código correspondente ao evento. No código também são incluídas as condições associadas aos eventos. A mudança para a nova configuração e o controle da execução de ações e atividades são efetuadas pela função *De_Para*, cujo código não é apresentado neste trabalho, mas pode ser encontrado em [Can91], onde é apresentado o código completo do simulador do Statechart do mostrador.

```

Simular(nome_evento)
enum Eventos nome_evento;
{
    Lista *p_lista, *p_lista2;
    int i;

    p_lista2 = p_lista = (struct List *)malloc(sizeof(Lista));
    p_lista = cabeca->proximo;
    while (p_lista != NULL){
        switch(p_lista->nome_estado){
            case HORA :
                switch(nome_evento) {
                    case C : De_Para(HORA,CRONOMETRO);
                        break;
                }
                break;
            case CRONOMETRO :
                switch(nome_evento) {
                    case C : De_Para(CRONOMETRO,HORA);
                        break;
                }
                break;
        }
    }
}

```

```

case ZERO :
    switch(nome_evento) {
        case A : De_Para(ZERO,OPERACAO);
            break;
    }
    break;
case NORMAL :
    switch(nome_evento) {
        case B : if (In(LIGADO))
            De_Para(NORMAL,VOLTA);
            if (In(DESLIGADO))
                De_Para(NORMAL,ZERO);
            break;
    }
    break;
case VOLTA :
    switch(nome_evento) {
        case B : De_Para(VOLTA,NORMAL);
            break;
    }
    break;
case LIGADO :
    switch(nome_evento) {
        case A : De_Para(LIGADO,DESLIGADO);
            break;
    }
    break;
case DESLIGADO :
    switch(nome_evento) {
        case A : De_Para(DESLIGADO,LIGADO);
            break;
    }
    break;
}
p_lista2 = cabeca->proximo;
cabeca->proximo = p_lista2->proximo;
p_lista = cabeca->proximo;
free(p_lista2);
}
}

```

Para não tornar o exemplo muito complexo, o simulador não inclui toda a generalidade dos Statecharts, como a entrada de mais de uma transição simultaneamente, transições com várias origens ou destinos e outras condições. Um simulador completo pode ser encontrado em [For91], onde os estados e as transições, ao contrário do simulador

apresentado acima, são representadas por estruturas de dados dinâmicas.

Não é difícil observar que o tipo enumerado de estados, a hierarquia de estados e a função *Simular* apresentados acima são particulares ao exemplo específico do mostrador. Sendo assim, essas porções de código são partes variantes do simulador proposto, ou seja, elas são reconfiguradas para cada Statechart apresentado.

Tendo, então, a linguagem de especificação e um simulador a ser gerado pelo GSS, resta definir, seguindo o modelo do gerador GEDAI, a descrição de produto em LDP do simulador de Statecharts. Essa descrição de produto será dividida em três partes: a descrição do tipo enumerado de estados, a descrição da hierarquia de estados e a descrição da função *Simular*. Com essas descrições em LDP, para cada Statechart de entrada, o GSS produzirá o tipo enumerado de estados, a hierarquia de estados e a função *Simular* correspondentes ao Statechart.

Primeiro, será considerada a descrição de produto do tipo enumerado *Estados*, mostrada no quadro abaixo, onde os trechos em *itálico* correspondem aos comandos da LDP. Esse simples programa em LDP, para cada descrição de estado, transfere para a saída o nome do estado.

```
%gabarito Tipo_enumerado
enum Estados{
    %paratodo e:DESCRICAO_ESTADO %repita
        %impr(%simb(estado(e))),
    %fimmt
        NULO };
%fimgab
```

No quadro seguinte é apresentada a descrição em LDP da hierarquia de estados. Não foram incluídas as definições das funções em LDP utilizadas, mas elas possuem nomes significativos que dispensam a apresentação do código, como por exemplo: *prim_filho(e)*, que retorna o primeiro filho do estado “e”.

Na descrição da hierarquia, para cada estado do Statechart, são transferidas as seguintes informações para o produto final: o nome do estado; o nome do pai do estado, se ele existir, senão retorna NULO; o nome do próximo irmão ou NULO; o nome do irmão anterior ou NULO; *n_lig* indicando que o estado não está ativo, pois o sistema ainda não foi acionado; *n_lig* para a informação sobre história; o tipo de decomposição do estado; se o estado é inicial ou não e, se for, o tipo (*def_h* ou *def_hh*); e, por último, o nível do estado na hierarquia. Vale observar que a descrição em LDP para a hierarquia de estados foi construída de forma que todas as informações referentes a um estado apareçam numa mesma linha no produto final.

Nota-se também, na descrição da hierarquia de estados, a chamada para duas rotinas de escape: a primeira, *prox_irmao*, retorna o próximo irmão de um dado es-

tado e a segunda, *ant-irmao*, retorna o irmão anterior. A implementação em C dessas rotinas foi necessária porque com os comandos da LDP não é possível obter as mesmas informações. Na verdade, a incorporação dessas rotinas no interpretador da LDP aparenta ser a definição de dois novos comandos da linguagem. Esses comandos não pertencem à própria definição da LDP porque achou-se que eles não são genéricos no sentido de manipular estruturas sintáticas de uma linguagem, ou seja, não se considerou significativo ter-se um comando que retorne o irmão à esquerda de uma estrutura sintática e outro que retorne o irmão à direita.

%gabarito Hierarquia

%NO no-pai, no-prox-irmao, no-ant-irmao;

Estado arvore_estado[] = {

%paratodo e:DESCRICAO_ESTADO %repita

{ %impr(%simb(estado(e))), %impr(prim_filho(e)),

%no-pai = pai(e);

%se no-pai != %NULO

%entao %impr(%simb(estado(no-pai))),

%no-prox-irmao = %escape(prox-irmao(%NO estado(e), %NO no-pai));

%se no-prox-irmao != %NULO

%entao %impr(%simb(no-prox-irmao)),

%senao %impr("NULO"),

%fimse

%no-ant-irmao = %escape(ant-irmao(%NO estado(e), %NO no-pai));

%se no-ant-irmao != %NULO

%entao %impr(%simb(no-ant-irmao)),

%senao %impr("NULO"),

%fimse %impr("n_lig"), %impr("n_lig"), %impr(tipo_estado(e)

), %impr(tipo_default(e, no-pai)), %impr(nivel(e))},

%senao %impr("NULO"), %impr("NULO"), %impr("NULO"), %impr("n_lig"

), %impr("n_lig"), %impr(tipo_estado(e)), %impr("n_def"), %impr(0)},

%fimse

%fimpt

{NULO, NULO, NULO, NULO, NULO, n_lig, n_lig, atomo, n_def, 0}

};

%fimgab

No terceiro e último quadro é apresentada a descrição em LDP da função *Simular*, onde também não foram incluídos os códigos das funções em LDP utilizadas.

```

%gabarito Simular
%NO ep, ep1, ev1;

Simular(nome_evento)
enum Eventos nome_evento;
{
    Lista *p_lista, *p_lista2;
    int i;

    p_lista2 = p_lista = (struct List *)malloc(sizeof(Lista));
    p_lista = cabeca->proximo;
    while (p_lista != NULL){
        switch(p_lista->nome_estado){
%paratodo e:DESCRICAO_ESTADO %repita
            %se tem_transicao(e) %entao
                case %impr(%simb(estado(e))):
                    switch(nome_evento){
%paratodo t:TRANSICAO %repita
                        %se (! %estrutura_marcada(t)) %entao
                            %se origem(e,t) %entao
                                %ep = evento_primitivo(evento(t));
                                case %impr(%simb(ep)):
%paratodo t1:TRANSICAO %repita
                                    %ev1 = evento(t1);
                                    %ep1 = evento_primitivo(ev1);
                                    %se ((%simb(ep1)==%simb(ep)) && origem(e,t1)) %entao
                                        %marca_estrutura(t1)
                                        %paratodo dest:ESTADO %em estados_destino(t1) %repita
                                            %se tem_condicao_in(ev1) %entao
                                                if (In(%impr(estado_condicao(ev1))))
%fimse
                                                    De_Para(%impr(%simb(estado(e))), %impr(%simb(dest)));
%fimpt
                                                %fimse
                                            %fimse
                                        %fimpt
                                    break;
%fimse
                                %fimse
                            %fimse
                        %fimpt
                    }
                    break;
%fimse
                %fimse
            %fimpt
        }
        p_lista2 = cabeca->proximo;
        cabeca->proximo = p_lista2->proximo;
        p_lista = cabeca->proximo;
        free(p_lista2);
    }
}
%fimgab

```

A interpretação dessa descrição é feita da seguinte maneira: para cada estado do Statechart são consideradas todas as transições que partem desse estado, caso exista alguma transição; para cada transição considerada, são consideradas também, caso existam, outras transições que partem do mesmo estado origem e são disparadas pelo mesmo evento primitivo que dispara a primeira; finalmente, para cada uma dessas transições, verifica-se a existência da “condição in” e, para cada estado destino, a função `De_Para` é transferida para o produto final, com seus dois argumentos: o nome do estado origem e o nome do estado destino. Cada transição considerada no segundo comando repetitivo referente às transições, que busca as transições com mesmo estado origem e mesmo evento primitivo, é marcada, para não ser novamente considerada no primeiro comando repetitivo.

4.3 Um Gerador de Aplicação para a Teoria dos Conjuntos

Nesta seção será considerado novamente o gerador de aplicação para a Teoria dos Conjuntos, proposto originalmente por Cleaveland, em [Cle88], e já tratado na seção 2.4.2. O objetivo é mostrar a possibilidade e a facilidade na instanciação do meta-gerador GE-DAI para outros domínios e aplicações. A apresentação será feita de forma que não se torne repetitiva.

4.3.1 A Linguagem de Especificação e o Analisador de Especificações

A gramática da linguagem de especificação de conjuntos é a mesma já apresentada nas seções 2.4.2 e 3.3.2. Para não repeti-la novamente, serão apresentadas a especificação de entrada para o LEX e parte da especificação de entrada para o YACC do analisador de especificações do gerador em questão. Nessas especificações, além das regras gramaticais da linguagem de especificação de conjuntos, estão incluídas as chamadas das rotinas de criação da árvore sintática. Com isso, dá-se a idéia de como pode ser construído, com o apoio do LEX e do YACC, um analisador de especificações para qualquer gerador de aplicação. Abaixo, apresenta-se a entrada do LEX para geração do analisador léxico do analisador de especificações.

```
identificador [a-zA-Z][a-zA-Z0-9_]*
```

```
%%
```

```
"{"          |
"}"          |
", "         return(yytext[0]);
```



```

{identificador}      {
                      ind = instala(yytext,TST);
                      yylval.no = cria_pai(ind,NULL);
                      return(IDENT);
                      }
" "                  |
\n                    |
\t                    |
.                     ;

```

Uma especificação de entrada para o LEX possui a seção de definições, a seção de regras e a seção de subrotinas do usuário, separadas pelo duplo percentual ‘%%’ [Sun90A]. A última seção, de subrotinas do usuário, freqüentemente é omitida, e é o caso da especificação do exemplo. A seção de definições da especificação acima define os identificadores como sendo a única categoria de símbolos reconhecida pelo analisador, que é representada pelo *token* IDENT. A seção de regras dessa mesma especificação, abaixo da marca ‘%%’, pode ser vista como uma tabela de duas colunas, onde a coluna da esquerda contém as “expressões regulares” e a coluna da direita contém as “ações”.

Uma expressão regular especifica um conjunto de cadeias de caracteres a serem casadas e, no caso do exemplo, tem-se expressões regulares para os símbolos especiais (‘{’, ‘}’ e ‘,’), para os identificadores e para qualquer outro caractere encontrado na entrada. As ações, que serão tratadas com maiores detalhes na apresentação da especificação de entrada para o YACC, podem ser entendidas da seguinte forma: para os símbolos especiais, a ação resume-se em enviar o próprio símbolo ao analisador sintático; para os identificadores, a ação consiste em instalar o identificador na TST, criar o nó folha da árvore sintática correspondente ao identificador e enviar o *token* IDENT ao analisador sintático, indicando que foi reconhecido um identificador; e, para os outros caracteres, a ação associada é nula, ou seja, os caracteres são ignorados.

A seguir é apresentada parte da especificação de entrada para o YACC do analisador de especificações. Toda especificação de entrada do YACC é composta de três seções: as declarações, as regras gramaticais e os programas criados pelo usuário [Sun90B]. Na especificação abaixo são apresentadas apenas uma parte da seção de declarações e a seção de regras, que também são separadas pelo duplo percentual ‘%%’.

```

%start inic
%token <no> IDENT
%type <no> espec conjunto nom_conj elemento
%%
inic      :      espec
          {
            ind = pesquisa_tabela("ESPECIFICACAO",TES);
            raiz = cria_pai(ind,$1);
          }
;

```

```

espec      :      conjunto
            |      conjunto espec
            {
                $$ = liga_irmaos($1,$2);
            }
            ;

conjunto:      nom_conj '{' elemento '}'
            {
                $$ = liga_irmaos($1,$3);
                ind = pesquisa_tabela("CONJUNTO",TES);
                $$ = cria_pai(ind,$$);
            }
            ;

nom_conj:      IDENT
            {
                ind = pesquisa_tabela("NOME_CONJUNTO",TES);
                $$ = cria_pai(ind,$1);
            }
            ;

elemento:      IDENT
            {
                ind = pesquisa_tabela("ELEMENTO",TES);
                $$ = cria_pai(ind,$1);
            }
            |      IDENT ',' elemento
            {
                ind = pesquisa_tabela("ELEMENTO",TES);
                $1 = cria_pai(ind,$1);
                $$ = liga_irmaos($1,$3);
            }
            ;

```

A seção de declarações acima indica que o símbolo inicial da gramática é o não terminal `inic` e a única categoria de símbolos que deve ser enviada pelo analisador léxico é `IDENT`. As duas últimas linhas indicam que tanto os identificadores quanto os não terminais da gramática representam ponteiros para nós da árvore sintática (<no>).

Na seção de regras, abaixo da marca '%%', são especificadas as produções da gramática da linguagem. A cada regra pode-se associar ações, que são executadas cada vez que a regra é reconhecida durante o processo de análise. Uma ação é especificada por um ou mais comandos em C entre chaves. No exemplo, as ações correspondem à criação da árvore sintática, utilizando as rotinas de criação apresentadas na seção 3.3.2.

As ações podem retornar valores e podem obter valores retornados por ações já executadas. Uma ação retorna um valor através da sua pseudo-variável '\$\$' e, para obter valores retornados por outras ações, utiliza as pseudo-variáveis '\$1', '\$2', ..., que se referem aos valores retornados pelos componentes à direita da regra correspondente, contando da esquerda para a direita. No exemplo, os valores retornados pelas ações são os ponteiros dos nós que vão sendo criados.

O valor de uma regra é o valor retornado pela sua ação correspondente. Regras que não possuem ação associada obtêm o valor do primeiro componente ('\$1'), como é o caso da regra:

espec : **conjunto**

Como, no YACC, a análise sintática é ascendente, a árvore sintática é construída a partir dos nós folhas (símbolos terminais), que são criados e retornados pelo analisador léxico. Esses ponteiros são passados para as regras no nível superior, que criam os nós intermediários correspondentes a elas, e assim sucessivamente, até que o nó raiz seja criado. O nó raiz do exemplo é o não terminal **ESPECIFICACAO**.

É importante observar que a especificação do YACC apresentada não inclui nenhuma checagem semântica da Teoria dos Conjuntos, como, por exemplo, que dois elementos de um mesmo conjunto não podem ter o mesmo nome. Checagens desse tipo também podem ser incluídas nas ações correspondentes a cada regra gramatical da especificação.

4.3.2 O Produto e a Descrição de Produto

Dada uma especificação de conjuntos, seguindo as regras gramaticais acima, o gerador de aplicação deve produzir estruturas de dados em C, que implementam os conjuntos, semelhantes às apresentadas na seção 2.4.2.

A descrição de produto em LDP do gerador, correspondente à descrição criada com a meta-linguagem do gerador STAGE, é mostrada no quadro abaixo. Deve-se observar que as duas descrições, a descrição em LDP e a descrição do STAGE, são praticamente idênticas. Isto se deve ao fato de que a LDP foi inspirada na meta-linguagem utilizada pelo STAGE.

%gabarito Conjuntos

```
int nro_conjuntos = %impr(%comp(conjunto(%topo)));
```

```
char *nome_conjunto[] = {  
    %paratodo n:NOME_CONJUNTO %repita  
        "%impr(%simb(n))",  
    %fimpt};
```

```
int tamanho_conjunto[] = {  
    %paratodo c:CONJUNTO  
        %repita %impr(%comp(elemento(c))),  
    %fimpt};
```

%paratodo c:CONJUNTO %repita

```
char *conjunto_%impr(%simb(nome_conjunto(c)))[] = {  
    %paratodo e:ELEMENTO %em c %repita  
        "%impr(%simb(e))",  
    %fimpt};  
%fimpt
```

```
char **valores_de_conjunto[] = {  
    %paratodo n:NOME_CONJUNTO %repita  
        conjunto_%impr(%simb(n)),  
    %fimpt};
```

%fimgab

Para exemplificar um produto de saída do gerador de aplicação, obtido pela interpretação da descrição em LDP acima e com as informações de uma especificação de conjuntos, será considerada a seguinte especificação:

```
carros{monza, santana}  
animais{cachorro, gato, cavalo}
```

Para essa especificação, o produto gerado é o seguinte:

```
int nro_de_conjuntos = 2;  
  
char *nome_do_conjunto[] = {
```

```

    "carros",
    "animais",});

int tamanho_do_conjunto[] = { 2, 3,};

char *conjunto_de_carros[] = {
    "monza",
    "santana",});

char *conjunto_de_animais[] = {
    "cachorro",
    "gato",
    "cavalo",});

char **valores_de_conjunto[] = {
    conjunto_de_carros,
    conjunto_de_animais,});

```

4.4 Ferramentas de Suporte Obtidas com a Própria Técnica de Geração

Uma prática comum durante a construção de geradores de aplicação é o aproveitamento da própria técnica de geração de código na construção de ferramentas de suporte ao próprio gerador. É o caso, por exemplo, do editor ALOEGEN, que foi construído com o gerador ALOE, e que apoia o próprio gerador na criação de especificações de outros editores orientados pela sintaxe [Hab86], e do meta-gerador STAGE, que foi construído com o seu próprio apoio [Cle88].

No gerador GEDAI, atualmente, duas ferramentas de suporte utilizam o mecanismo de geração de código definido para o próprio meta-gerador. Uma delas é o gerador de analisadores de especificações, apresentado na seção 3.3.2, que utiliza a linguagem LDP na descrição da entrada para o YACC desse tipo de analisador. A outra é um pré-processador da LDP, já mencionado na seção 3.3.3, sem maiores detalhes, que apoia o interpretador da LDP no caso de descrições de produtos com rotinas de escape.

Quando o interpretador da LDP identifica uma chamada de rotina de escape, é preciso fazer a chamada dessa rotina na sintaxe da linguagem C, com o nome dela e os seus argumentos. A “montagem” dessa chamada na sintaxe da linguagem C é realizada antes da interpretação da descrição de produto, pelo pré-processador.

O pré-processador recebe como entrada a própria descrição de produto em LDP e, para cada rotina de escape, gera uma função em C que realiza a chamada da rotina. Além disso, é gerado um vetor que associa o nome de cada rotina de escape ao ponteiro da função³ correspondente em C. O resultado final do pré-processador é o arquivo “escape.h”,

³O ponteiro de uma função é o endereço de memória do ponto de entrada da função.

que deve ser incluído (*header file*) no arquivo com as rotinas de escape e este deve ser compilado e ligado ao código objeto do interpretador da LDP⁴. Portanto, com o pré-processador, o programador do programa em LDP fica livre de ter que construir o arquivo "escape.h" manualmente. É importante observar que apenas as descrições de produtos com chamadas de rotinas de escape necessitam desse pré-processamento.

Durante a interpretação, então, o interpretador da LDP, com o *token* que representa o nome da rotina de escape, obtém no vetor gerado pelo pré-processador o ponteiro da função que chama a rotina e transfere a ela a execução. Os argumentos da rotina são computados pelo interpretador e são armazenados num vetor de argumentos, que é utilizado pela função que realiza a chamada da rotina de escape.

Como exemplo, será considerada a descrição de produto da hierarquia de estados do gerador GSS, apresentada na seção 4.2.2, que possui chamadas para duas rotinas de escape: *prox_irmao* e *ant_irmao*. O vetor e as duas funções em C gerados pelo pré-processador para essa descrição são os seguintes:

```
struct declaracao_rotinas_escape{
    char *rotina;          /* Nome da rotina de escape */
    void (* pont_funcao)(); /* Ponteiro para a funcao em C */
} rotinas_escape[] = {
    "prox_irmao", chama_prox_irmao,
    "ant_irmao", chama_ant_irmao,
    "", 0
};

void chama_prox_irmao()
{
    PONT *arg1;
    PONT *arg2;
    arg1 = escape_argumentos[0].valor.pont;
    arg2 = escape_argumentos[1].valor.pont;
    escape_retorno.tipo = NO;
    escape_retorno.valor.pont = prox_irmao( arg1, arg2 );
}

void chama_ant_irmao()
{
    PONT *arg1;
    PONT *arg2;
    arg1 = escape_argumentos[0].valor.pont;
    arg2 = escape_argumentos[1].valor.pont;
    escape_retorno.tipo = NO;
    escape_retorno.valor.pont = ant_irmao( arg1, arg2 );
}
```

⁴Maiores detalhes dessa operação podem ser encontrados em [Mei91A]

Na definição do vetor `rotinas_escape`, o primeiro campo de cada elemento é o nome da rotina de escape e o segundo campo é o ponteiro para a função que realiza a chamada da rotina. O último elemento do vetor funciona como uma sentinela.

As funções que chamam as rotinas de escape declaram os tipos de cada argumento da rotina, obtém esses argumentos do vetor de argumentos computado pelo interpretador e realizam a chamada da rotina. Além disso, elas declaram o tipo do valor retornado pela rotina e armazenam esse valor, utilizando a variável `escape_retorno`, que depois é utilizada pelo interpretador. No caso do exemplo acima, os argumentos das rotinas e os valores retornados por elas são do tipo `PONT`, que representa um ponteiro para um nó da árvore sintática.

O analisador de especificações do pré-processador só considera as chamadas de rotinas de escape – o restante das descrições de produtos é ignorado – criando a árvore sintática de acordo com a sintaxe dessas chamadas (veja apêndice B). Sendo assim, possíveis erros de sintaxe nas chamadas dessas rotinas são identificados pelo pré-processador. No apêndice A é dado o tamanho em linhas de código fonte desse analisador.

A descrição de produto do pré-processador inclui a descrição do vetor de rotinas de escape e a descrição das funções em C que realizam as chamadas das rotinas. Essas descrições são apresentadas nos quadros abaixo. A descrição de produto do pré-processador inclui ainda certas partes invariantes como, por exemplo, declarações de tipos e variáveis em C, que não interessam no momento e, sendo assim, não foram apresentadas.

%gabarito Vetor_de_rotinas

```
struct declaracao_rotinas_escape{
    char *rotina;          /* Nome da rotina de escape */
    void (* pont_funcao)(); /* Ponteiro para a funcao em C */
} rotinas_escape[] = {
    %paratodo esc:ROTINA_ESCAPE %repita
    "%impr(%simb(nome_rotina(esc)))", chama_%impr(%simb(nome_rotina(esc))),
    %fimpt
    "", 0
};
%fmgab
```

Na descrição do vetor (quadro acima) é feito um percurso na árvore sintática e, para toda rotina de escape, imprime-se duas vezes o nome dela na saída. O resto é parte invariante do produto. A descrição da função de chamada (quadro abaixo), igual para todas rotinas de escape, é interpretada da seguinte forma: para todo argumento da rotina, representado pela estrutura sintática TIPO – pela sintaxe, toda expressão que resulta num argumento deve ter um tipo associado – é criado o seu comando de declaração e o comando que obtém o seu valor no vetor de argumentos; depois, finalizando, são criados o comando que declara o tipo retornado pela rotina de escape e o comando que realiza a chamada dessa rotina.

```

%gabarito Funcao_de_chamada
%INT ind;

%paratodo esc:ROTINA_ESCAPE %repita
%ind = 1;

void chama_%impr(%simb(nome_rotina(esc))){
{
%paratodo tip:TIPO %em argumentos_escape(esc) %repita
%se %simb(tip)=="%INT" %entao
%impr("int") arg %impr(ind);
%senao %se %simb(tip)=="%NO" %entao
%impr("PONT") *arg %impr(ind);
%senao
%impr("char") arg %impr(ind)[TAM_ID];
%fimse
%fimse
%ind = ind+1;
%fimpt
%ind = 1;
%paratodo tip:TIPO %em argumentos_escape(esc) %repita
%se %simb(tip)=="%INT" %entao
arg %impr(ind) = escape_argumentos[%impr(ind-1)].valor.i;
%senao %se %simb(tip)=="%NO" %entao
arg %impr(ind) = escape_argumentos[%impr(ind-1)].valor.pont;
%senao
strcpy(arg%impr(ind), escape_argumentos[%impr(ind-1)].valor.simb);
%fimse
%fimse
%ind = ind+1;
%fimpt
%se %simb(tipo(esc))=="%INT" %entao
escape_retorno.tipo = INTEIRO;
%ind = 0;
escape_retorno.valor.i = %impr(%simb(nome_rotina(esc))){
%paratodo tip:TIPO %em argumentos_escape(esc) %repita
%se (ind>0) %entao arg %impr(ind),
%fimse
%ind = ind+1;
%fimpt arg %impr(ind) );
%senao %se %simb(tipo(esc))=="%NO" %entao
escape_retorno.tipo = N0;
%ind = 0;
escape_retorno.valor.pont = %impr(%simb(nome_rotina(esc))){
%paratodo tip:TIPO %em argumentos_escape(esc) %repita
%se (ind>0) %entao arg %impr(ind),
%fimse
%ind = ind+1;
%fimpt arg %impr(ind) );
%senao
escape_retorno.tipo = SIMBOLO;
%ind = 0;
strcpy(escape_retorno.valor.simb, %impr(%simb(nome_rotina(esc))){
%paratodo tip:TIPO %em argumentos_escape(esc) %repita
%se (ind>0) %entao arg %impr(ind),
%fimse
%ind = ind+1;
%fimpt arg %impr(ind) );
%fimse
%fimse
}
%fimpt
%fimgab

```


Resumindo, uma especificação fonte para o pré-processador é um programa em LDP que utiliza chamadas de rotinas de escape. O pré-processador, a partir desse programa, traduz as chamadas de rotinas de escape, na sintaxe da linguagem LDP, para um produto, o arquivo "escape.h", que inclui as chamadas dessas mesmas rotinas de escape, agora na sintaxe da linguagem C. Essa tradução é realizada interpretando-se a descrição de produto em LDP.

4.5 Passos a serem Seguidos na Instanciação de GEDAI

Nesta seção serão apresentados os passos a serem seguidos para se instanciar GEDAI, ou seja, os passos necessários para se construir um gerador de aplicação específico, utilizando os mecanismos proporcionados pelo meta-gerador.

A instanciação do gerador GEDAI é dividida em quatro passos:

Definição da Linguagem de Especificação: neste passo define-se a linguagem textual de especificação para o domínio de aplicação. O resultado é uma gramática dessa linguagem, na qual as estruturas sintáticas (símbolos não terminais da gramática) devem estar bem definidas.

Construção do Analisador de Especificações: neste passo deve-se construir o analisador que verifica a correção sintática de uma especificação de entrada e constrói a árvore sintática de derivação dessa especificação, de acordo com as regras gramaticais definidas no passo anterior.

Definição da Estrutura dos Produtos: este passo consiste em identificar, de alguma forma, a estrutura dos produtos a serem gerados. Uma prática comum é identificar a estrutura de um produto específico a ser gerado, correspondente a uma certa especificação. É importante que esse produto inclua todos os recursos que se queira incorporar nos produtos. Outra maneira é observar as estruturas de produtos já existentes (por exemplo, sistemas de um certo domínio), que sejam semelhantes aos produtos que se deseja criar com o gerador, e obter uma estrutura mais geral com características selecionadas desses produtos.

Sendo assim, neste passo é possível identificar as partes variantes e invariantes dos produtos, isto é, as partes que dependem e as que não dependem da especificação de entrada.

Construção da Descrição de Produto: neste passo, o último, constrói-se a descrição em LDP dos produtos a serem gerados. Ela é construída com base na estrutura definida no passo anterior, substituindo as partes variantes por comandos da LDP, onde esses comandos devem considerar a gramática da linguagem de especificação definida no primeiro passo. Também neste passo, devem ser implementadas as rotinas de escape, caso elas sejam necessárias.

O passo de Definição da Estrutura dos Produtos pode ser realizado independentemente dos outros três passos, desde que se saiba, previamente, quais características do domínio devem possuir os produtos a serem gerados. Os passos de Construção do Analisador de Especificações e de Construção da Descrição de Produto dependem do passo de Definição da Linguagem de Especificação. O passo de Construção da Descrição de Produto depende também do passo de Definição da Estrutura dos Produtos.

Além dos passos acima, certos comandos operacionais devem ser executados, para que se tenha a versão executável final do gerador de aplicação que se está construindo. Esses comandos incluem a geração do código objeto do analisador de especificações, a geração do código objeto das rotinas de escape, caso elas existam, e a ligação desses códigos com o código objeto do interpretador da LDP. Maiores detalhes desses comandos podem ser encontrados em [Mei91A].

Com o término da implementação do gerador de analisadores de especificações, o passo de construção desse tipo de analisador ficará despercebido para quem estiver construindo um gerador de aplicação com o apoio de GEDAI. O projetista de domínios não precisará mais implementar manualmente o analisador, bastando a ele fornecer a gramática da linguagem de especificação ao gerador de analisadores, que, automaticamente, produzirá o analisador de especificações correspondente.

Abaixo, será ilustrado como os passos de instanciação do gerador GEDAI se inserem nos sete passos da estratégia de Cleaveland, apresentada na seção 2.2. Como exemplo, será considerado o gerador de aplicação GSS, para o domínio de Sistemas Reativos, apresentado na seção 4.2.

Os três primeiros passos da estratégia, **Reconhecimento de Domínios**, **Definição dos Limites do Domínio** e **Definição do Modelo Básico**, se referem à definição do domínio de aplicação e, portanto, não há o que se fazer com o gerador GEDAI.

No caso do GSS, havia um grupo trabalhando com a técnica Statechart e com a simulação de sistemas especificados com essa técnica, e percebeu-se a possibilidade da utilização de um gerador de aplicação. Com isso, o modelo básico foi definido como sendo os próprios Statecharts e o domínio ficou limitado aos tipos de sistemas que se pode especificar com a técnica.

Definição das Partes Variante e Invariante: este passo corresponde ao passo de **Definição da Estrutura dos Produtos**.

Num programa simulador de Statechart, a parte variante são os nomes dos estados, a quantidade de estados, as transições, os eventos, etc, toda ela obtida do próprio Statechart. A parte invariante corresponde à lógica do simulador e é genérica para qualquer Statechart. Essa lógica foi obtida com a codificação em C do simulador do Statechart do mostrador de um relógio digital [Can91].

Definição da Entrada da Especificação: este passo corresponde ao passo de **Definição da Linguagem de Especificação**.

Como a técnica Statechart é visual, mas, no GEDAI, as especificações devem ser fornecidas na forma textual, definiu-se a Linguagem de Especificação de Statecharts

(LES) e criou-se uma função adicional no editor gráfico EGS, que transforma uma representação diagramática na representação textual em LES correspondente.

Definição do Produto: este passo corresponde ao passo de **Construção da Descrição de Produto**.

A descrição de produto do GSS foi criada a partir do programa simulador do Statechart do mostrador do relógio digital. Os pontos desse programa que dependiam do Statechart foram substituídos por comandos da linguagem LDP, que buscam as mesmas informações, para qualquer outro Statechart, na árvore sintática. Também, foi necessária a implementação de duas rotinas de escape.

Implementação do Gerador: neste passo inclui-se o passo de **Construção do Analisador de Especificações**.

No caso do exemplo, a partir da gramática da LES, construiu-se o analisador de especificações do GSS. De resto, foi preciso apenas juntar as partes definidas anteriormente e alimentar o meta-gerador.

4.6 Considerações Finais

Os requisitos necessários para alguém que queira instanciar o meta-gerador GE-DAI são: habilidade na definição de linguagens textuais, através de gramáticas; conhecimento e prática na linguagem LDP; experiência na construção de analisadores sintáticos; e, conhecimento das rotinas de criação da árvore sintática e dos locais apropriados para as chamadas dessas rotinas. Os dois últimos requisitos não serão mais necessários quando o gerador de analisadores de especificações estiver pronto.

Supondo a existência do gerador de analisadores de especificações e com a prática adquirida na construção dos três geradores de aplicação apresentados neste capítulo, pode-se afirmar o seguinte: tendo-se definido o domínio de aplicação e sabendo-se a estrutura de pelo menos um produto a ser gerado, que inclua todos os recursos desejados, fica extremamente simples a construção de um gerador de aplicação.

Capítulo 5

Conclusão

5.1 Considerações Gerais

Fazendo um resumo geral do trabalho, depois de se ter pesquisado os geradores de aplicação e apresentado vários exemplos do uso desses geradores, fica confirmado que eles, quando bem aplicados, têm obtido grandes resultados. As principais contribuições ocorrem na fase de desenvolvimento de software, com o aumento da produtividade e qualidade de código e com a redução nos custos, e na fase de manutenção, permitindo que as modificações sejam feitas apenas na especificação.

Porque, então, os geradores de aplicação não são largamente utilizados? Os motivos principais identificados foram: a dificuldade de se identificar domínios de aplicação apropriados e a incerteza do sucesso do gerador. Aliados a essas dificuldades, a complexidade na construção de geradores de aplicação e o longo prazo na obtenção de resultados, decorrente do tempo de desenvolvimento do gerador, contribuem para desestimular o desenvolvimento dos geradores.

Com o gerador de propósito geral desenvolvido neste trabalho tentou-se minimizar as dificuldades encontradas na construção de geradores de aplicação. Quanto ao problema de se identificar domínios apropriados para o uso de geradores, ainda aberto, fica a observação de que esta é uma questão mais subjetiva, dependendo principalmente da sensibilidade do analista de domínios para descobrir novos domínios. Acredita-se que a experiência acumulada com a criação de geradores para diferentes domínios, facilitada pelo uso do meta-gerador, possa levar a um maior conhecimento desse problema e a uma eventual solução.

O grande trunfo do meta-gerador apresentado é estabelecer um processo organizado na construção de geradores de aplicação, permitindo ao projetista de domínios, depois de uma certa prática com os mecanismos oferecidos e com as ferramentas de suporte proporcionadas, construir facilmente esse tipo de gerador. Essa facilidade foi demonstrada e comprovada com a construção de dois geradores para domínios específicos diferentes, além da utilização do próprio meta-gerador na construção de suas ferramentas de suporte.

Esse processo padronizado de construção de geradores de aplicação proposto

também é importante pois permite que novos esforços de pesquisa fiquem concentrados nos mecanismos utilizados pelo meta-gerador, e possíveis avanços tecnológicos, decorrentes desses esforços, sejam compartilhados por todos os geradores construídos.

5.2 Contribuições do Trabalho Realizado

Acredita-se que a classificação de geradores de aplicação proposta, aliada aos modelos básicos correspondentes, propostos para cada nível de generalidade da classificação, possa vir a ser de grande utilidade para o profissional que deseja construir ou selecionar um gerador, permitindo que ele escolha o modelo mais adequado às suas necessidades. Por exemplo, se um usuário decide selecionar um gerador de aplicação, dentre vários geradores existentes, ele pode escolher um gerador do tipo menos genérico, se o resultado desejado é específico, ou pode escolher um do tipo mais genérico, se ele deseja estender os benefícios do gerador a mais de um problema.

Além disso, a síntese do conhecimento na área, organizada na forma do modelo de classificação proposto, pode dar origem a novos “insights” na área. Como exemplo, métodos de desenvolvimento baseados em geradores de aplicação podem ser explorados.

O desenvolvimento do gerador para o domínio de Sistemas Reativos é considerado inovativo, no sentido de explorar um novo domínio de aplicação, apesar do simulador possível de ser gerado não incluir toda a generalidade da técnica Statechart. É importante ficar claro que a implementação completa de um simulador, incluindo todos os recursos dos Statecharts, é possível de ser feita, mas, neste trabalho, o interesse principal foi demonstrar a funcionalidade e a utilidade do meta-gerador.

Também, acredita-se que esse gerador para o domínio de Sistemas Reativos possa se tornar uma ferramenta de apoio a métodos de especificação de sistemas de tempo real que utilizam Diagramas de Transição de Estados para especificar o aspecto comportamental desses sistemas, como é o caso das extensões da Análise Estruturada propostas por Ward, em [War86], e Hatley, em [Hat84].

5.3 Sugestões de Futuras Pesquisas

Este primeiro trabalho sobre o tema geradores de aplicação no ICMSC abre caminho para várias novas pesquisas possíveis de serem feitas. De imediato, pode-se sugerir a exploração de novos domínios de aplicação e a implementação dos respectivos geradores, utilizando os mecanismos proporcionados pelo meta-gerador desenvolvido. Com isso, poder-se-ia identificar novos comandos da meta-linguagem de descrição de produtos, que não se sentiu ou percebeu a necessidade neste trabalho, ou descobrir novas e relevantes informações a serem armazenadas na estrutura interna criada a partir da especificação fonte. Isto seria de fundamental importância para aumentar o poder do meta-gerador.

Em continuidade ao desenvolvimento do meta-gerador tem-se duas opções: o término do desenvolvimento do gerador de analisadores de especificações, permitindo a

automação completa no processo de geração desses analisadores, inclusive com a incorporação de checagens semânticas das linguagens de especificação; e o desenvolvimento de ferramentas de suporte para a construção do módulo de interface.

Uma outra linha de pesquisa é tentar, como na abordagem DRACO, estabelecer uma organização de domínios de aplicação, onde domínios mais gerais e abstratos podem ser descritos em termos de domínios mais específicos e concretos. Esses refinamentos poderiam ser realizados através do mecanismo de tradução de uma representação abstrata para uma representação concreta utilizado pelo meta-gerador, ou seja, o resultado de um nível de refinamento de uma representação mais abstrata para uma representação mais concreta poderia ser descrito através da meta-linguagem de descrição de produtos.

Bibliografia

- [Aho82] AHO, A.; HOPCROFT, J.E.; ULLMANN, J.D. - *Data Structures and Algorithms*. Massachusetts, Addison-Wesley, 1982.
- [Bah85] BAHLKE, R.; SNELTING, G. - The PSG-Programming System Generator. In: Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, Seattle, Washington, 1985. *SIGPLAN Notices*, 20(7): 28-33, 1985.
- [Bah86] BAHLKE, R.; SNELTING, G. - The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, 8(4): 547-576, 1986.
- [Bat91] BATISTA, J.E.S. - *Um Editor Gráfico para Statecharts*. Dissertação de Mestrado, ICMSC-USP, 1991.
- [Bel85] BELANGER, D.G.; KINTALA, C.M.R. - Data-Extraction Tools. *AT&T Technical Journal*, 64(9): 2025-2035, 1985.
- [Ben86] BENTLEY, J. - Programming Pearls: Little Languages. *Communications of the ACM*, 29(8): 711-721, 1986.
- [Boe83] BOEHM, B.W.; STANDISH, T.A. - Software Technology in the 1990's: Using an Evolutionary Paradigm. *Computer IEEE*, 16(11): 30-37, 1983.
- [Can91] CANGUSSÚ, J. W. L. - *Um Simulador para um Cronômetro Digital*. Relatório Técnico, ICMSC-USP, 1991.
- [Cle88] CLEAVELAND, J.C. - Building Application Generators. *IEEE Software*, 5(4): 25-33, 1988.
- [Dar87] DART, S.A. et al. - Software Development Environments. *Computer IEEE*, 20(11): 18-28, 1987.
- [Dem81] DEMERS, A.; REPS, T.; TEITELBAUM, T. - Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In: Conference Record of the 8th ACM Symposium on Principles of Programming Languages, Williamsburg, VA, 1981. p. 105-116.

- [Don80] DONZEAU-GOUGE, V. et al. - Programming Environments Based on Structure Editors: The Mentor Experience. *INRIA Technical Report 26*, France, 1980. Reprinted in: BARSTON, D.R.; SHROBE, H.E.; SANDEWALL, E. eds. - *Interactive Programming Environments*. New York, McGraw-Hill, 1984. p. 128-140.
- [Don84] DONZEAU-GOUGE, V. et al. - Documents Structure and Modularity in Mentor. In: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, 1984. *SIGPLAN Notices*, 19(5): 141-148, 1984.
- [Ell85] ELLISON, R.J.; STAUDT, B.J. - The Evolution of the Gandalf System. *The Journal of Systems and Software*, 5(2): 107-119, 1985.
- [Fai85] FAIRLEY, R.E. - *Software Engineering Concepts*. New York, McGraw-Hill, 1985.
- [Fig91] FIGUEIREDO, A.G.; LIESENBERG, H.K.E. - Geração de Gerenciadores de Sistemas Reativos. Anais do V Simpósio Brasileiro de Engenharia de Software, Ouro Preto, Minas Gerais, 23 a 25 de Outubro, 1991 (a aparecer).
- [Fis84] FISCHER, C.N. et al. - The Poe Language-Based Editor Project. In: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, 1984. *SIGPLAN Notices*, 19(5): 21-29, 1984.
- [For91] FORTES, R.P.M. - *Uma Ferramenta de Apoio à Utilização de Statecharts para Especificação do Comportamento de Sistemas de Tempo Real Complexos*. Dissertação de Mestrado, ICMSC-USP, 1991.
- [Fre87] FREEMAN, P. - A Conceptual Analysis of the Draco Approach to Constructing Software Systems. *IEEE Transactions on Software Engineering*, 13(7): 830-844, 1987.
- [Hab86] HABERMANN, A.N.; NOTKIN, D. - Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, 12(12): 1117-1127, 1986.
- [Har87A] HAREL, D. - Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3): 231-274, 1987.
- [Har87B] HAREL, D. et al. - On the Formal Semantics of Statecharts. In: Proceedings of the 2nd IEEE Symposium on Logic in Computer Science, Ithaca, N.Y., 1987. p. 54-64.
- [Hat84] HATLEY, D.J. - The use of Structured Methods in the Development of Large-software Avionics Systems. In: Proceedings of the AIAA/IEEE 6th Digital Avionics Systems Conference, Washington D.C., 1984.

- [Hor85] HOROWITZ, E.; KEMPER, A.; NARASIMHAN, B. - A Survey of Application Generators. *IEEE Software*, 2(1): 40-54, 1985.
- [Knu68] KNUTH, D.E. - Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2): 127-145, 1968 (Correction 5(1): 95-96, 1971).
- [Lei90] LEITE, J.C.S.P. ; FRANCO, A.M.M. - O Uso de Hipertexto na Elicitação de Linguagens da Aplicação. Anais do IV Simpósio Brasileiro de Engenharia de Software, Águas de São Pedro, SP, 1990. p. 134-149
- [Lev86] LEVY, L.S. - A Metaprogramming Method and Its Economic Justification. *IEEE Transactions on Software Engineering*, SE-12(2): 272-277, 1986.
- [Lew90] LEWIS, T. - Code Generators. *IEEE Software*, 7(3): 67-70, 1990.
- [Lis86] LISKOV, B. - *Abstraction and Specification in Program Development*. Cambridge, MIT Press, 1986.
- [Luk86] LUCKER, P.A. ; BURNS, A. - Program Generators and Generation Software. *The Computer Journal*, 29(4), 1986.
- [Mar85] MARTIN, J.M. - *Fourth Generation Languages*. New Jersey, Prentice-Hall, 1985.
- [Mei91A] MEIRA, C.A.A. - *Manual da Linguagem de Descrição de Produtos (LDP) de Geradores de Aplicação*. Relatório Técnico, ICMSC-USP, 1991.
- [Mei91B] MEIRA, C.A.A.; MASIERO, P.C. - Um Gerador de Aplicações para Sistemas Reativos. Anais do V Simpósio Brasileiro de Engenharia de Software, Ouro Preto, Minas Gerais, 23 a 25 de Outubro, 1991 (a aparecer).
- [Mra90] MRACK, F.R.; SILVA, M.S.; PRICE, R.T. - Um Dicionário de Dados para um Editor Diagramático Generalizado. Anais do IV Simpósio Brasileiro de Engenharia de Software, Águas de São Pedro, SP, 1990. p. 218-230.
- [Nei84] NEIGHBORS, J.M. - The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, 10(5): 564-574, 1984.
- [Not85] NOTKIN, D. - The GANDALF Project. *The Journal of Systems and Software*, 5(2): 91-105, 1985.
- [Pag91] PAGANINI, G.V. - *Um Gerador de Árvores Sintáticas para a Linguagem de Especificação de Statecharts*. Documento de Trabalho nº IC-91/03, ICMSC-USP, 1991.
- [Pre87] PRESSMAN, R.S. - *Software Engineering: A Practitioner's Approach*. 2 ed. New York, McGraw-Hill, 1987.

- [Rep83] REPS, T.; TEITELBAUM, T.; Demers, A. - Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Transactions on Programming Languages and Systems*, 5(3): 449-477, 1983.
- [Rep84A] REPS, T.; TEITELBAUM, T. - The Synthesizer Generator. In: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, 1984. *SIGPLAN Notices*, 19(5): 42-48, 1984.
- [Rep84B] REPS, T.; ALPERN, B. - Interactive Proof Checking. In: Conference Record of the 11th ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, 1984. p. 36-45.
- [Sch89] SCHILDT, H. - Building Your Own C Interpreter. *Dr. Dobbs's Journal*, August, 1989. p. 38-49.
- [Set91] SETZER, V.W. - *Superstructured Programming and the I-M-E Application Generator*. Bericht Nr. 7, Philipps-Universität Marburg, 1991.
- [Sie85] SIEVERT, G.E.; MIZELL, T.A. - Specification-Based Software Engineering with TAGS. *Computer IEEE*, 18(4): 56-65, 1985.
- [Sun90A] SUN MICROSYSTEMS - LEX - Lexical Analyser Generator. *Programmer's Overview - Utilities & Libraries*, Chapter 9: 203-226, 1990.
- [Sun90B] SUN MICROSYSTEMS - YACC - Yet Another Compiler-Compiler. *Programmer's Overview - Utilities & Libraries*, Chapter 10: 227-263, 1990.
- [Tei81A] TEITELBAUM, T.; REPS, T. - The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM*, 24(9): 563-573, 1981.
- [Tei81B] TEITELBAUM, T.; REPS, T.; SUZAN, H. - The Why and Wherefore of the Cornell Program Synthesizer. In: Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland, Oregon, 1981. *SIGPLAN Notices*, 16(6): 8-16, 1981.
- [Vuo88] VUONG, S.T.; LAU, A.C.; CHAN, R.I. - Semiautomatic Implementation of Protocols Using an Estelle-C Compiler. *IEEE Transactions on Software Engineering*, 14(3): 384-393, 1988.
- [War86] WARD, P.T. - The Transformation Schema: an Extension of the Data Flow Diagram to Represent Control and Timing. *IEEE Transactions on Software Engineering*, SE-12(2): 198-210, 1986.
- [You78] YOURDON, E.; CONSTANTINE, L.L. - *Structured Design*. New York, Yourdon Press, 1978.

Apêndice A

Aspectos Gerais da Implementação

O ambiente de desenvolvimento deste trabalho foi uma estação de trabalho SUN SPARCstation 1, sob o controle do sistema operacional UNIX (SunOS). Todos os programas foram desenvolvidos na linguagem C, com a edição e depuração apoiadas pelo ambiente OpenWindowsTM versão 2.

Na tabela A.1 são apresentadas as partes do projeto desenvolvidas e discutidas no decorrer deste trabalho, juntamente com o tamanho em linhas de código fonte correspondente a cada parte, num total aproximado de 6000 linhas. O tempo de duração da fase de implementação foi de cerca de seis meses.

Um ponto a se comentar na tabela é que, apesar da diferença de tamanho entre o código fonte da primeira versão do Analisador de Especificações para a LES e o código fonte (entradas para o LEX e YACC) da segunda versão não ser muito grande (aproximadamente 600 linhas), a segunda versão foi mais simples de ser construída e o código fonte é bem mais legível e compreensível do que o código da primeira versão. Sendo assim, essa diferença torna-se muito significativa e a segunda versão é considerada bem superior à primeira.

Parte do Projeto	Tamanho em Linhas de Código Fonte
Interpretador da LDP	1735
Analizador de Especificações LES (descendente recursivo)	1885
Analizador de Especificações LES (LEX e YACC)	1309
TAD Árvore Sintática	714
Pré-processador da LDP	339

Tabela A.1: Tamanho em Linhas de Código Fonte das Partes do Projeto

Apêndice B

Gramática da Linguagem de Descrição de Produtos (LDP)

A seguir é definida a sintaxe da LDP, utilizando a notação BNF (Backus-Naur Form) estendida. Observações: $\alpha \mid \beta$ indica a opção por α ou β ; $[\alpha]$ indica que a cadeia α é opcional; e, $\{\alpha\}$ indica zero ou mais vezes a repetição da cadeia α . Os símbolos terminais estão descritos com a notação utilizada pelo utilitário LEX.

1. `<descricao do produto> ::=`
 `[ <declaracoes de funcoes> ]`
 `<modulo principal>`
2. `<modulo principal> ::=`
 `'%gabarito' <identificador>`
 `[ <declaracoes de variaveis> ';' ]`
 `<comando composto> '%fimgab'`
3. `<declaracoes de funcoes> ::=`
 `<funcao> { <funcao> }`
4. `<funcao> ::=`
 `<identificador> '(' [ <declaracoes de variaveis> ] ')'`
 `[ <declaracoes de variaveis> ';' ]`
 `<comando composto> '%fimfunc'`
5. `<declaracoes de variaveis> ::=`
 `<declaracao> { ';' <declaracao> }`
6. `<declaracao> ::=`
 `<tipo> <lista de identificadores>`
7. `<tipo> ::= ( '%NO' | '%INT' | '%SIMBOLO' )`

8. <comando composto> ::=
 <comando> { <comando> }

9. <comando> ::=
 <atribuicao>
 | <chamada de subrotina>
 | <retorna>
 | <imprime>
 | <comando de decisao>
 | <comando repetitivo>
 | <marca estrutura>
 | <desmarca estrutura>

10. <atribuicao> ::=
 '%seja' <identificador> '=' <expressao> ';' ;
 | <identificador> '=' <expressao> ';' ;

11. <chamada de subrotina> ::=
 '%chame' <chamada de funcao>
 | <chamada de funcao>

12. <retorna> ::=
 '%retorna' '(' <expressao> ')'

13. <imprime> ::=
 '%impr' '(' <expressao> ')'

14. <comando de decisao> ::=
 '%se' <expressao> '%entao' <comando composto>
 ['%senao' <comando composto>] '%fimse'

15. <comando repetitivo> ::=
 '%paratodo' <identificador> ':' <estrutura sintatica>
 ['%em' <expressao>] '%repita' <comando composto> '%fimpt'

16. <marca estrutura> ::=
 '%marca_estrutura' '(' <expressao> ')'

17. <desmarca estrutura> ::=
 '%desmarca_estrutura' '(' <expressao> ')'

18. <expressao> ::=
 <expressao simples> [<relacao> <expressao simples>]

19. <relacao> ::= ('=' | '!=' | '<' | '<=' | '>=' | '>')
20. <expressao simples> ::=
[('+' | '-')] <termo> { ('+' | '-' | '||') <termo> }
21. <termo> ::=
<fator> { ('*' | '/' | '&&') <fator> }
22. <fator> ::=
<identificador>
| <estrutura sintatica> '(' <expressao> ')'
| '%comp' '(' <expressao> ')'
| '%simb' '(' <expressao> ')'
| '%estrutura_marcada' '(' <expressao> ')'
| <chamada de funcao>
| <rotina escape>
| <numero>
| '!' <expressao>
| '(' <expressao> ')'
23. <estrutura sintatica> ::= <identificador>
24. <chamada de funcao> ::=
<identificador> '(' [<lista de expressoes>] ')'
25. <rotina escape> ::=
'%escape' '(' <tipo> <nome rotina>
'(' [<argumentos escape>] ')')'
26. <nome rotina> ::= <identificador>
27. <argumentos escape> ::=
<tipo> <expressao> { ',' <tipo> <expressao> }
28. <lista de expressoes> ::=
<expressao> { ',' <expressao> }
29. <lista de identificadores> ::=
<identificador> { ',' <identificador> }
30. <identificador> ::= [a-zA-Z][a-zA-Z0-9_]*
31. <numero> ::= [0-9]+

Apêndice C

Gramática da Linguagem de Especificação de Statecharts (LES)

Abaixo são apresentadas as produções da gramática que define a linguagem textual para descrição de Statecharts, descritas em notação BNF estendida. Os símbolos terminais estão descritos com a notação utilizada pelo utilitário LEX.

1. `<estadograma> ::=`
 `[ <declaracoes> ';' ]`
 `<estados> ';'`
 `<transicoes> ';'`
2. `<declaracoes> ::= <declaracao> { <declaracao> }`
3. `<declaracao> ::= <tipo_id> <variavel> { ',' <variavel> }`
4. `<tipo_id> ::= ( 'INT' | 'BOOL' | 'REAL' | 'STRING' | 'COND' )`
5. `<variavel> ::= <identificador>`
6. `<estados> ::= <descricao_estado> { <descricao_estado> }`
7. `<descricao_estado> ::= 'ESTADO' <estado> <tipo>`
8. `<estado> ::=`
 `<identificador>`
 `| <estado rotulado>`
9. `<tipo> ::=`
 `'OU' <subestados_ou>`
 `| 'E' <subestados_e>`
 `| 'ATOMO'`

10. <subestados_ou> ::=
 'SUBESTADOS' [<default>] <estado> { ',' <estado> }
11. <default> ::= ('DEF' | 'DEF/H' | 'DEF/H*')
12. <subestados_e> ::=
 'SUBESTADOS' <estado> { ',' <estado> }
13. <transicoes> ::= <transicao> { <transicao> }
14. <transicao> ::=
 'EVENTO' <evento>
 'ORIGEM' <estados_origem>
 'DESTINO' <estados_destino>
 ['ACAO' '{' <acoes> '}']
15. <evento> ::= <ev_ou> { 'or' <ev_ou> }
16. <ev_ou> ::= <ev_e> { 'and' <ev_e> }
17. <ev_e> ::= <ev_f> ['[' <exp> ']']
18. <ev_f> ::=
 <ev_primitivo>
 | <bool> '(' <exp> ')'
 | <changed>
 | <exit>
 | <entered>
 | <timeout>
19. <ev_primitivo> ::= <identificador>
20. <bool> ::= (('true' | 'tr') | ('false' | 'fs'))
21. <changed> ::= ('changed' | 'ch') '(' <exp> ')'
22. <exit> ::= ('exit' | 'ex') '(' <estado> ')'
23. <entered> ::= ('entered' | 'en') '(' <estado> ')'
24. <timeout> ::= 't_out' '(' <numero> <tempo> ')'
25. <tempo> ::= ('seg' | 'min' | 'hr')

26. <exp> ::=
 <expS> [('=' | '>' | '<' | '<>' | '<=' | '>=') <expS>]

27. <expS> ::=
 [('+' | '-')] <termo> { ('+' | '-' | 'or') <termo> }

28. <termo> ::=
 <fator> { ('*' | '/' | 'and') <fator> }

29. <fator> ::=
 <bool>
 | <numero>
 | <variavel>
 | <cond_in>
 | <not_yet>
 | <current>
 | 'not' <exp>
 | '(' <exp> ')'

30. <cond_in> ::= 'in' '(' <estado> ')'

31. <not_yet> ::= ('not_yet' | 'ny') '(' <evento> ')'

32. <current> ::= ('current' | 'cr') '(' <exp> ')'

33. <estados_origem> ::= <estado> { ',' <estado> }

34. <estados_destino> ::= <destino> { ',' <destino> }

35. <destino> ::= <estado> [<historia>]

36. <historia> ::= ('/H' | '/H*')

37. <acoes> ::= <acao> { ';' <acao> }

38. <acao> ::=
 <rotina>
 | <atribuicao>
 | <clear>

39. <rotina> ::= <nome_rot> '(' <lista_arg> ')'

40. <nome_rot> ::= <identificador>

- 41. <lista_arg> ::= <argumento> { ',', <argumento> }
- 42. <argumento> ::= <exp>
- 43. <atribuicao> ::= <variavel> ':' <exp>
- 44. <clear> ::= 'clear' '(' <estado> ['*'] ')'
- 45. <identificador> ::= [a-zA-Z][a-zA-Z0-9_/*]*
- 46. <estado rotulado> ::=
[a-zA-Z][a-zA-Z0-9_/*]*"."[a-zA-Z][a-zA-Z0-9_/*]*
- 47. <numero> ::= [0-9] +

Apêndice D

Gramática da Notação BNF Estendida

A seguir são apresentadas as produções da gramática que define a notação BNF estendida, utilizando a própria notação. Os símbolos terminais estão descritos com a notação utilizada pelo utilitário LEX.

1. `<bnf> ::=`
 `<conjunto_regras>`
2. `<conjunto_regras> ::=`
 `<regra>`
 | `<regra> <conjunto_regras>`
3. `<regra> ::=`
 `<nao_terminal> '::=' <composto>`
4. `<composto> ::=`
 `<concatenacao>`
 | `<concatenacao> '|' <composto>`
5. `<concatenacao> ::=`
 `<elementar>`
 | `<elementar> <concatenacao>`
 | `'[' <opcional> ']' <concatenacao>`
 | `'{' <iteracao> '}' <concatenacao>`
 | `<cadeia_vazia>`
6. `<elementar> ::=`
 `<nao_terminal>`
 | `<terminal>`

7. <opcional> ::= <concatenacao>
8. <iteracao> ::= <concatenacao>
9. <cadeia_vazia> ::=
10. <nao_terminal> ::= '<' <identificador> '>'
11. <terminal> ::= <cadeia de caracteres>
12. <identificador> ::= [a-zA-Z][a-zA-Z0-9_]*
13. <cadeia de caracteres> ::= '[^'\n\t\f\r\b]*'