Adesso

Ambiente para Desenvolvimento de Software Científico

Rubens Campos Machado

Junho de 2002

UNIVERSIDADE ESTADUAL DE CAMPINAS - UNICAMP
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

ADESSO
AMBIENTE PARA DESENVOLVIMENTO DE SOFTWARE CIENTÍFICO

Autor: **Rubens Campos Machado**

Orientador: **Prof. Dr. Roberto de Alencar Lotufo**

Comissão Julgadora: **Prof. Dr. Eleri Cardoso**
Prof. Dr. Gerald Jean Francis Banon
Prof. Dr. Maurício Ferreira Magalhães

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: **Engenharia de Computação**.

Junho de 2002

Campinas, SP - Brasil

Resumo

Este documento descreve a concepção e implementação do Adesso, um ambiente computacional de suporte ao desenvolvimento de aplicações científicas. O Adesso explora o modelo de programação baseado em componentes reutilizáveis para fornecer suporte ao desenvolvimento de componentes e sua integração a diversas plataformas de programação científica.

O Adesso baseia-se em uma base de dados de componentes representada em XML (*toolboxes*) e em um conjunto de ferramentas de transformação para geração automática de código, documentação e empacotamento dos componentes.

O processo de transformação é controlado por *folhas de estilos* definidas em uma linguagem que mescla elementos estruturais da linguagem XSLT com procedimentos e comandos da linguagem Tcl. Este trabalho apresenta a linguagem de transformação de documentos XML implementado para o Adesso.

Foram implementadas, na linguagem do processador de estilos do Adesso, diversas *folhas de estilo* que possibitam a transformação de uma *toolbox* em pacotes de componentes para utilização nas plataformas MATLAB e Tcl/Tk. São gerados automaticamente, para cada *toolbox*, a documentação em HTML e PDF, código fonte das interfaces e regras de construção (*Makefiles*).

Abstract

This document presents the conception and implementation of Adesso, a computacional environment for the development of scientific software. The Adesso environment leverages the reusable software component programming model to support the development and integration of the components to several scientific programming platforms.

The Adesso system is based on an XML component database (*toolboxes*) and a set of XML document transformation tools for the automatic generation of component code, documentation and packaging.

The transformation process is controlled by *stylesheets* written in a language that mix structural XSLT elements with commands and procedures of the Tcl language. This work describes the XML document transformation language, implemented for Adesso.

Using the Adesso style processor language, several stylesheets have been implemented. These stylesheets enable the tranformation of the toolbox data into component packages suitable for use in MATLAB and Tcl/Tk platforms. From each toolbox, the stylesheets generate HTML and PDF documentation, interface source code and package building rules (*Makefiles*).

A meu pai, Celso Machado, ainda que tarde.

Agradecimentos

Ao Lotufo, meu colega e orientador, por esses anos de parceria produtiva e estimulante que resultou neste trabalho.

Aos meus colegas de ITI, especialmente ao Paulo, testemunha deste longo processo desde o início.

À minha esposa, Efigênia, e aos meus meninos, Daniel e Júlia, pelo amor e alegria sempre presentes em nossa casa.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Os principais resultados de projetos de pesquisa em áreas como processamento de sinais, visão computacional, reconhecimento de padrões, controle de processos, bioinformática e outras afins, são métodos e algoritmos inovadores. Ao longo destes projetos, programas de demonstração são desenvolvidos em torno desses algoritmos para a comprovação de sua viabilidade. Estes programas, embora não constituam o principal resultado de um projeto de pesquisa, são, em muitos casos, o embrião de projetos de desenvolvimento que podem levar à criação de produtos inovadores.

A criação de aplicações científicas de qualidade envolve, porém, mais que a implementação de algoritmos e métodos. Uma aplicação de sucesso deve ter, além de algoritmos de alta qualidade científica, uma implementação robusta e atualizada tecnologicamente, em conformidade com padrões de qualidade ditados pelas práticas correntes. No atual estágio da tecnologia da informação, estes padrões são cada vez mais exigentes, principalmente em termos de interfaces gráficas sofisticadas, integração com outras aplicações, qualidade da documentação de usuário e segurança de informações.

Todos estes fatores conduzem a modelos de desenvolvimento de software em que o reaproveitamento de código exerce um papel fundamental. Um desses modelos, cada vez mais utilizado, estabelece que componentes preexistentes são interconectados através de plataformas integradoras para a criação de soluções para problemas específicos. A

funcionalidade total exigida de uma aplicação é fatorada entre sub-aplicações com interfaces bem definidas — componentes, visando sua reutilização em outros problemas.

1.1 O Modelo Componentes/Solução

A idéia de se dividir o desenvolvimento de aplicações em duas fases não é nova [1]. O surgimento de plataformas de programação como o Microsoft Visual Basic, em que componentes denominados *controles* são conectados através de uma linguagem visual para a confecção de aplicações, popularizou muito este modelo.

A aceitação desse modelo criou todo um mercado voltado para a oferta de componentes, pacotes de software que implementam, de forma eficiente, funcionalidades comuns a uma certa área de aplicações. O sucesso do modelo pode ser verificado, inclusive, na mudança de postura da Microsoft que, depois de um período de forte apoio aos sistemas baseados em reaproveitamento de código baseado em *frameworks* orientados a objeto (como, por exemplo, o MFC), apostou na evolução do modelo de componentes do *Visual Basic*, chegando ao que é, hoje, conhecido como COM, uma plataforma independente de linguagens para cooperação entre programas.

Um outro aspecto dessa evolução dos sistemas de software é o espectro de linguagens de programação utilizados na confecção de uma aplicação. De um lado temos as linguagens de programação de sistemas como C, C++, Java. De outro, linguagens de scripting e ambientes de programação visual como o *Visual Basic*, *Perl* e *Python*. As linguagens de programação de sistemas são adequadas à criação de sistemas onde a eficiência computacional é o fator mais relevante, sendo, pois, muito apropriada para a codificação de componentes de propósito específico, projetados para serem integrados em uma plataforma de scripting.

Um argumento importante para a utilização do modelo baseado em duas linguagens é a constatação de que as habilidades e características dos programadores e linguagens envolvidos em cada etapa são diferentes e a produtividade aumenta com a escolha correta das ferramentas.

Este paradigma se aplica especialmente bem ao desenvolvimento de software científico onde a implementação de algoritmos quase sempre exige a utilização de linguagens de programação de sistemas. A confecção de aplicações, entretanto, pode ser feita a partir de linguagens de scripting ou programação visual com grandes ganhos de produtividade e facilidades de depuração. O *MATLAB* é um exemplo clássico de plataforma de scripting para programação científica; o *Khoros* [2][3] é outro sistema, voltado principalmente ao processamento de imagens, que utiliza uma linguagem de programação visual. Linguagens de scripting de uso geral, como *Perl*, *Tcl* e *Python* [4], também são utilizadas nesta classe de problemas. Todas estas plataformas disponibilizam interfaces de programação bem definidas para a incorporação de extensões escritas, principalmente, em C/C++.

A criação destas extensões é um processo mecânico que se torna muito trabalhoso à medida em que se tem muitos componentes. Este tipo de situação incentiva o uso da prática "copia e cola" pois a maioria dos programas possuem um gabarito a ser seguido. É comum criar um novo componente a partir da cópia de um componente similar e fazer as substituições necessárias. Esta prática é uma das maiores dificuldades encontradas por equipes de desenvolvimento de sofware científico. A prática de "copia e cola" aumenta consideravelmente o número de linhas de programas, é uma fonte inerente de bugs e faz com que o sistema venha a apresentar uma manutenção muito cara para ser atualizado na tentativa de seguir as tendências tecnológicas. A existência de ferramentas para automatizar a criação destas extensões é vital para a qualidade dos produtos desenvolvidos com base neste modelo.

1.2 Documentação, Testes e Empacotamento

Um aspecto muitas vezes relegado a segundo plano é a importância da documentação especializada exigida pelo software científico. A documentação é muitas vezes a parte mais cara de todo o software pois deve ser feita por um profissional experiente, muitas vezes com as mesmas aptidões de um autor de livro científico. A documentação deve

tipicamente conter um manual de referência das ferramentas, um tutorial e um conjunto de demonstrações. O manual de referência deve conter a descrição sintática do comando ou função, exemplos ilustrativos, equações matemáticas, referências bibliográficas e descrição do algoritmo utilizado. Dois aspectos relacionados à documentação são ainda imprescindíveis: a importância da correspondência com as novas versões do software e a necessidade de apresentações diferentes, tipicamente na forma de documentos impressos, páginas para a Internet e ajuda on-line.

Uma atividade importante do desenvolvimento de software é a fase de testes do código desenvolvido. O modelo de componentes, por procurar diminuir as dependências entre módulos do sistema total, facilita a incorporação de *testsuites* ao processo de desenvolvimento. A existência de ferramentas de autoria que enfatizem a associação de rotinas de teste a cada componente desenvolvido facilita a deteção antecipada de problemas.

Uma vez criado o código e a documentação de um conjunto de componentes, é necessário empacotar o sistema de forma a facilitar o trabalho de construção, instalação e manutenção do software. Esta é uma etapa importante na criação de sistemas de produção e, juntamente com a fase de testes, caracteriza a passagem da fase de prototipagem para a fase de produção. Também aqui, a automação traz ganhos substanciais. Esta tarefa é desempenhada tipicamente por scripts conhecidos como *Makefiles*, conjunto de regras que especificam as dependências entre os componentes e os comandos para a construção de binários, seleção de subconjuntos e criação do pacote de componentes a ser distribuído.

1.3 O Modelo Adotado pelo Adesso

Qualquer informação traz consigo, inerentemente, três partes: conteúdo, estrutura e apresentação. Conteúdo é a informação em si, a sua estrutura define um conjunto de regras que permitem verificar a consistência da informação e a apresentação é a forma de comunicação com o usuário. Já há algum tempo é conhecido que a informação pode ser melhor gerenciada e reutilizada se estas três partes estiverem separadas.

Um exemplo proeminente desta separação são os bancos de dados. A informação é armazenada em tabelas cujos campos constituem a estrutura dos dados e implementam interrelacionamentos. Já a apresentação é composta dos diversos relatórios que podem ser gerados dependendo da visão desejada. A apresentação é moldada de acordo com a audiência. Um banco de dados acadêmicos mostra aparências diferentes quando visto por alunos, professores ou dirigentes administrativos. Todas estas aparências são geradas a partir de uma única base de dados.

A padronização SGML introduzida nos final dos anos 80 foi o primeiro esforço de padronização para estabelecer uma forma de guardar e processar a informação que passou a ser adotada por várias editoras. Recentemente com a explosão da informação na Internet, surgiu o padrão XML que é uma versão inspirada na SGML, porém mais moderna e versátil. XML é uma linguagem de marcação onde as etiquetas são definidas pelo usuário. A linguagem XML é projetada para ser manipulada pelo computador, permitindo uma análise léxica e sintática simplificada e eficiente.

A idéia básica que permeia o sistema de autoria de software científico aqui apresentado é a mesma que motivou o uso do XML na Internet. A informação líquida (algoritmos e documentação) é armazenada de forma estruturada e sem redundâncias. A apresentação desta informação é gerada de forma automática por geradores de código (a transformação é especificada em *stylesheets*) e podem apresentar a forma de programas em C, Java, arquivos de configuração, Makefiles, documentação HTML, PDF ou ajuda on-line, demonstrações e aplicações stand-alones.

A adoção da metodologia de estruturar a informação e usar geração de código e documentos de forma automatizada traz inúmeros benefícios, sendo os principais:

- diminuição do número de linhas de informação mestre, responsável tanto pelo programa como pela sua documentação, aumentando imunidade a bugs, diminuindo espaço de armazenamento de back-ups, e diminuindo o esforço de manutenção e gerenciamento da informação.

- consistência na interface com o usuário, visto que o código resultante é gerado automaticamente, sendo inerentemente sistemático por todo o sistema.

- informação facilmente adaptável para gerar código e documentação de acordo com os avanços tecnológicos exigidos pelo mercado.
- informação armazenada de forma estruturada e padronizada, facilmente convertida para novos padrões de informações.
- maior robustez a bugs tipicamente introduzidos em situações de "copia e cola".
- maior imunidade contra erros na criação de um empacotamento e distribuição do aplicativo, permitindo uma consistência na configuração desejada.

O objetivo do sistema descrito neste documento, Adesso, é automatizar os processos de criação de software científico através da geração do código, documentação, testsuites e informações de construção. O cerne do sistema é sua abordagem para a representação das informações, baseada na linguagem de marcação XML. Assim, o mesmo componente codificado em C, por exemplo, pode ser interfaceado com diferentes plataformas de scripting a um custo mínimo. A manutenção das informações do componente em formato estruturado e bem definido tem impactos, ainda, na geração de documentação e na criação de conjuntos bem definidos de componentes para distribuição.

O capítulo 2 deste documento apresenta uma visão geral da arquitetura do Adesso e os principais cenários para sua utilização. Algumas da principais tecnologias relacionadas com o XML são apresentadas no capítulo 3. As informações que constituem uma *toolbox* são modeladas através de um *esquema XML*, discutido no capítulo 5.

A principal contribuição do sistema desenvolvido ao processo de desenvolvimento de software científico é o seu sistema de transformação de documentos XML e o conjunto de folhas de estilo para geração de código e documentação de *toolboxes*. Este sistema tem como núcleo o *Processador de Estilos* e sua linguagem para criação de folhas de estilo e é descrito em detalhes no capítulo 4. As folhas de estilo existentes são apresentadas no capítulo 6.

O capítulo 7 discute a implementação de sistemas de autoria para o Adesso. Alguns exemplos de utilização prática do processador de estilos são apresentados no capítulo

8. Finalmente, apresentamos algumas conclusões a respeito do trabalho realizado e sugerimos alguns projetos para o aperfeiçoamento do Adesso.

Capítulo 2

Arquitetura do Sistema

O Adesso é um sistema de autoria de programas científicos baseado no paradigma componente/solução onde conjuntos de componentes são desenvolvidos e mantidos em *toolboxes* com interfaces programáticas bem definidas. Em uma outra fase, estes componentes são conectados através de uma *plataforma de integração*, normalmente linguagens de scripting, para a criação de uma aplicação.

Uma *toolbox* é constituída por um conjunto ortogonal de informações, suficientes para que um *produto* seja gerado. Um *produto* é formado pelo código e documentação necessários para a utilização da *toolbox* em determinada *plataforma de integração* sob determinada *plataforma computacional*. Estes *produtos* são criados através de ferramentas de transformação, aqui denominadas *geradores de código*.

Assim, por exemplo, uma "Toolbox de Processamento de Imagens" pode ser a base de um produto que permite sua utilização a partir do MATLAB em sistemas Microsoft Windows. A mesma toolbox poderia originar outro produto para Python/Linux.

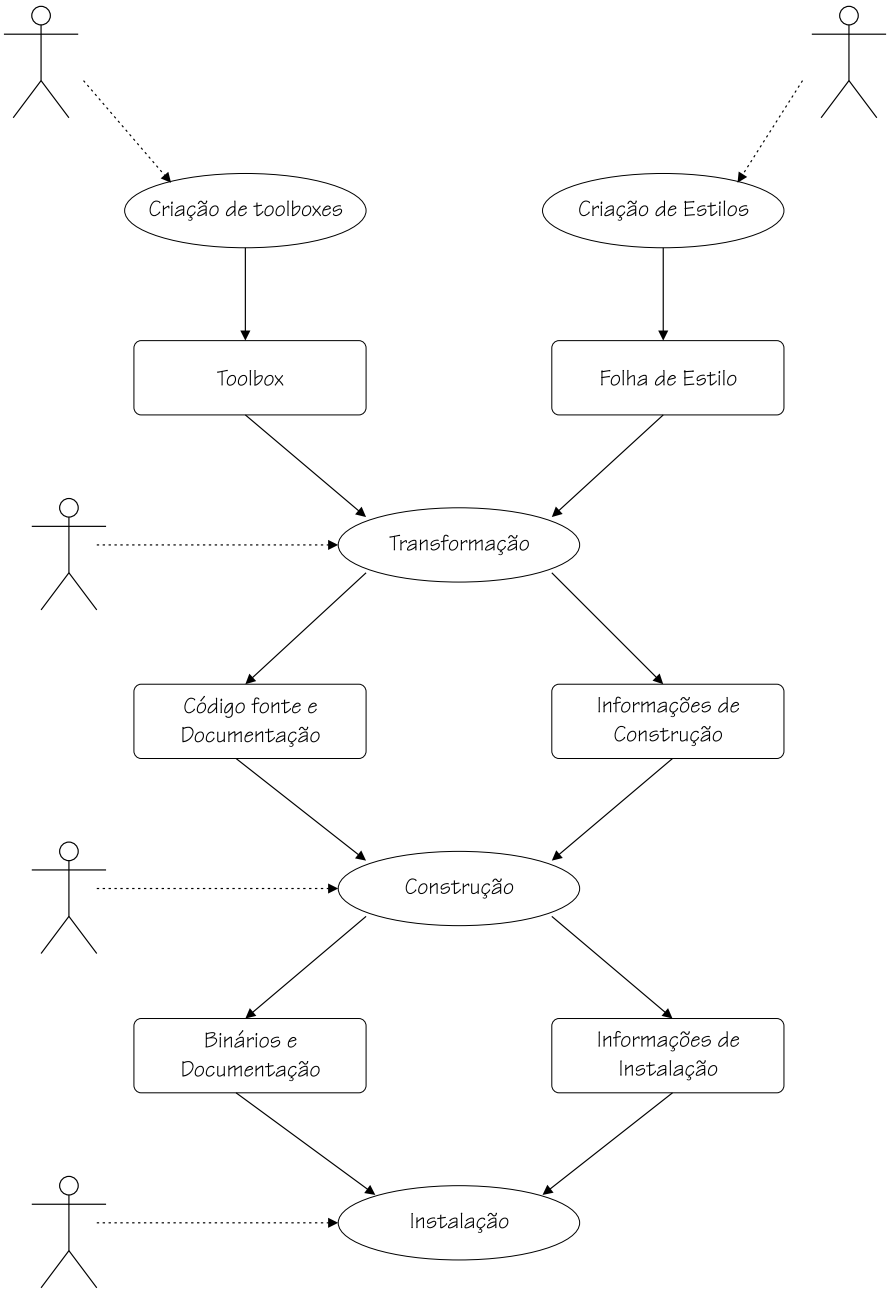

Figura 2.1: Cenários de Utilização do Adesso

2.1 Cenário de Utilização

O Adesso provê um conjunto de ferramentas para a criação de *toolboxes*, estilos de transformação e produtos derivados. A figura 2.1 mostra as etapas do processo de desenvolvimento apoiadas por ferramentas do Adesso.

- **Criação de toolboxes**

  Com o auxílio de ferramentas de autoria, o autor especifica o código e a documentação da toolbox, de acordo com um determinado modelo de dados, descrito no capítulo 5. As informações que constituem a toolbox são independentes tanto da plataforma de integração (*MATLAB*, *Tcl* etc) quanto da plataforma computacional (MSWindows, Linux etc).

  O produto desta etapa é um conjunto de arquivos XML contendo todas as informações pertinentes à toolbox.

- **Geração de código e documentação (Transformação)**

  A etapa de transformação é responsável pela criação de uma *distribuição fonte* da toolbox para uma determinada *plataforma de integração*. Esta distribuição contém toda a documentação e arquivos fonte das funções que compõem a toolbox devidamente adaptados às interfaces providas pela plataforma de integração.

  O processo de tranformação é especificado por *folhas de estilo* e implementado por um *processador de estilos*. Para cada plataforma de integração é usado um conjunto de folhas de estilo.

  Uma distribuição fonte é independente da plataforma computacional, ficando para a fase seguinte a criação de distribuições específicas para cada sistema operacional. Para orientar este processo, informações de construção, na forma de um conjunto de *Makefiles*, também são criadas nesta fase.

- **Construção de uma distribuição binária**

A construção de uma *distribuição binária* específica para um sistema operacional é realizada com o auxílio de uma ferramenta de construção. A distribuição fonte criada na etapa anterior é transportada para a plataforma computacional alvo onde, com o auxílio da ferramenta de construção e de compiladores e ferramentas específicas da plataforma, a *distribuição binária* é criada.

Ainda nesta fase, diferentes tipos de distribuição podem ser construídos baseados em restrições impostas quanto ao conjunto de componentes ou a limitações propositais. Versões estudantis ou de avaliação são exemplos deste tipo de restrição.

- **Instalação da toolbox**

  A última etapa do processo é realizada, usualmente, pelo usuário da toolbox com a ajuda de uma ferramenta de instalação. Nesta etapa, são especificadas informações relativas a diretórios e opções de instalação.

- **Criação de folhas de estilo**

  O porte da toolbox para novas plataformas de integração ou a incorporação de novos formatos de documentos é feita através da criação de novas folhas de estilos.

2.2 Organização do Adesso

O Adesso, veja a figura 2.2, é organizado em torno de uma base de dados (*Toolboxes*) que contém informações que definem conjuntos de componentes de software. A base de dados do Adesso utiliza uma representação baseada na linguagem de marcação XML [5].

XML, *Extensible Markup Language*, é uma linguagem de marcação de documentos baseada em um padrão mais antigo, o SGML, *Standardized General Markup Language*. O SGML foi desenvolvido na IBM, em 1969, como uma forma padronizada de estruturar documentos. A complexidade da linguagem, porém, limitou sua utilização a grandes corporações que manipulam vastas quantidades de documentos. O XML surgiu com a

meta de ampliar o uso do SGML, visando principalmente a Web, através de sua simplificação. Suas principais características são:

- em primeiro lugar, conforme estabelecido como prioridade na recomendação do W3C [5] que padroniza a linguagem: XML é simples e fácil de usar;
- XML suporta uma variedade de aplicações, permitindo a criação de marcações específicas de cada domínio;
- documentos XML seguem regras e princípios formais e bem definidos;
- os documentos são legíveis e razoavelmente claros para o entendimento de pessoas.

Em nosso sistema, o XML foi usado para criar uma marcação para a especificação de bases de dados de componentes de software.

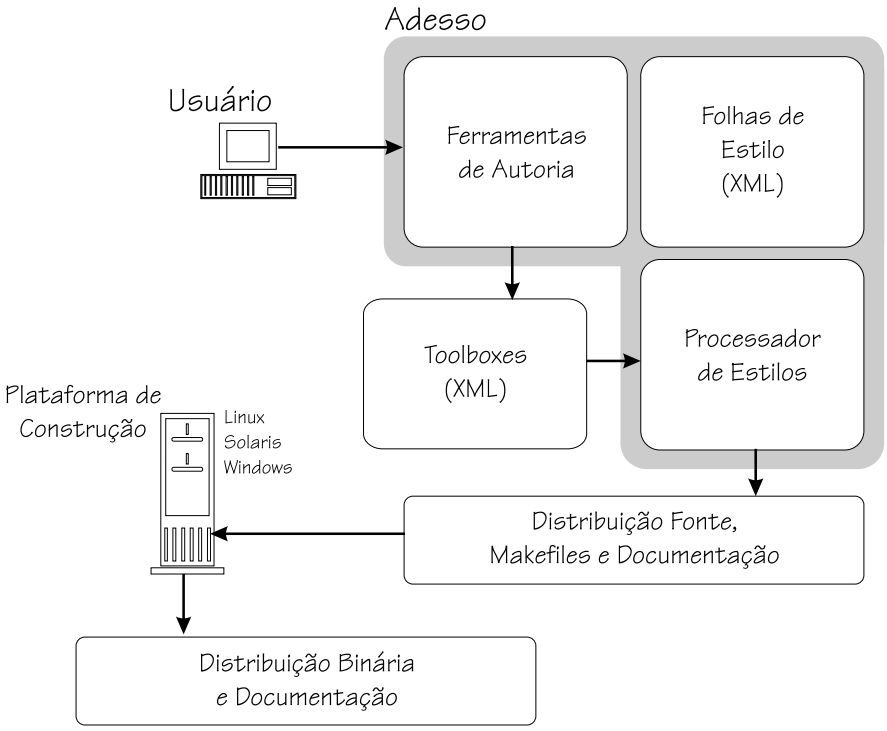

Figura 2.2: Organização do Adesso

A base de dados do Adesso foi modelada usando a linguagem de definição de esquemas do XML (XML Schema Definition Language, [6], [7], [8]). A norma *XML Schema* especifica uma linguagem, baseada em XML, para a definição de esquemas de representação de dados. O propósito de um esquema XML é definir e descrever uma classe de documentos XML através da utilização dos componentes da linguagem para restringir e documentar o significado, a utilização e as relações entre as partes constituintes do documento: tipos de dados, elementos e seus conteúdos e atributos e seus valores. Em resumo, um esquema XML define um vocabulário para uma classe de documentos XML.

A partir das informações de sua base de dados e utilizando ferramentas de transformação inspiradas no modelo do XML/XSLT [9], o Adesso é capaz de gerar interfaces para diversas plataformas de scripting e programação visual além de uma variedade de formatos de documentação.

Várias transformações no estilo XSLT são aplicadas aos componentes de uma toolbox para a geração de uma distribuição-fonte. Uma distribuição-fonte é constituída pelo código fonte gerado para cada componente, pela sua documentação, scripts para orientar o processo de construção dos binários etc. Uma distribuição-binária pode então ser construída, a partir da distribuição-fonte, em diferentes plataformas computacionais (Unix, MSWindows) através de ferramentas usuais de compilação de programas.

Todos os processos de transformação efetuados pelo Adesso são realizados através de seu *Processador de Estilos*. Este componente aceita como entrada arquivos da base de dados de uma toolbox juntamente com uma *folha de estilo* — um arquivo XML que descreve uma transformação a ser efetuada nos dados de entrada. O Adesso tem um conjunto extensível de folhas de estilo para a geração de interfaces e documentação para algumas plataformas de scripting como o Tcl e o Matlab.

A linguagem de transformação utilizada pelo Adesso para a implementação de seus geradores de código, inspirada pela norma XSLT, é baseada em textos XML marcados com instruções da linguagem de scripting Tcl [4]. Esta configuração origina scripts de transformação que possibilitam o acesso à interface básica DOM [10], [11] e a uma im-

plementação das expressões XPath [12], facilitando a utilização de *queries* complexos. Esta decisão de projeto, porém, não exclui a possibilidade do uso da linguagem XSLT para a criação de geradores de código.

Para a criação e manutenção de seus componentes e a aplicação de transformações, o Adesso é dotado de um conjunto de ferramentas de autoria que constituem um ambiente integrado de desenvolvimento. O ambiente de desenvolvimento é dotado de um editor XML criado a partir do esquema da base de dados. Todas as ferramentas de geração de código e documentação são acessíveis através destas ferramentas.

Capítulo 3

Principais Tecnologias Envolvidas

3.1 XML

XML [5] é uma linguagem de marcação para a definição e comunicação de dados estruturados através de arquivos de texto. Dados estruturados permeiam os sistemas computacionais atuais. Planilhas, parâmetros de configuração, desenhos técnicos, bookmarks, transações financeiras são exemplos de dados estruturados produzidos e/ou consumidos por aplicações computacionais. XML especifica um conjunto de regras e convenções para o projeto de textos que representem tais tipos de dados de forma a facilitar a geração e o entendimento (por computadores), evitar ambiguidades e dependências de plataformas computacionais e facilitar a troca de informações entre computadores.

Apesar de representado em texto simples, XML não visa a sua leitura e entendimento por pessoas. É voltado, sim, para sua utilização por computadores. O fato de ser representado em arquivos de texto facilita a manutenção dos dados, possibilitando a correção de problemas, em situações anormais, com a utilização de editores de texto simples. Contudo, XML é verboso e extremamente sensível a erros de sintaxe — o que desaconselha sua edição direta.

Outra decorrência da representação na forma de texto é que arquivos XML tendem a se tornar volumosos. Contudo esta é uma decisão de projeto baseada, principalmente, nas

vantagens advindas desta representação e na disponibilidade de ferramentas de compressão de dados.

Um arquivo XML se parece muito com um arquivo HTML. XML, todavia, é muito diferente do HTML em seus objetivos. A marcação da linguagem HTML visa especificar a aparência de seu conteúdo. A marcação XML explicita as relações estruturais existentes entre as partes componentes do documento através de etiquetas de marcação (*tags*) cuja semântica é associada ao domínio do documento. HTML oferece um conjunto de etiquetas pré-definido. XML é um método para aplicar marcadores cujo significado depende do domínio da aplicação. Ambas as linguagens de marcação têm relação direta com o SGML. O HTML é um vocabulário específico, definido a partir do SGML, para a apresentação de documentos. O XML é um padrão equivalente ao SGML, com simplificações para facilitar seu uso.

Para ilustrar melhor, consideremos o seguinte documento XML, que define o *protótipo* de uma função C. O *protótipo* de uma função define a sintaxe de utilização da função e os tipos de cada argumento e do valor de retorno.

```
<?xml version="1.0"?>
<AdCFunction name="processImage">
    <!-- função criada por RXF -->
    <Return type="adImage" name="out" output="no">
        Soma das imagens (out = i0 + i1)
    </Return>
    <Args>
        <Arg name="i0" dir="in"  type="adImage" constraint="GRAY"/>
        <Arg name="i1" dir="in"  type="adImage" constraint="GRAY"/>
        <Arg name="i2" dir="out" type="adImage">
            i2 = out = i0 + i1
        </Arg>
        <Arg name="i3" dir="in"  type="adInt" optional="yes"
                                              default="99"/>
    </Args>
    <Source file="srcs/processImage.c"/>
    Esta função efetua ...
</AdCFunction>
```

A definição da função C `processImage` é formada por três conjuntos de definições correspondentes aos elementos `Return`, `Args` e `Source`.

O elemento `Return` especifica que a função retorna um valor do tipo `adImage`, posto que `'type=adImage'`, contudo, não deve ser retornado à plataforma com a qual a função é interfaceada (`output=no`). Estas definições são feitas através de símbolos que são associados a valores e são denominados atributos.

A definição dos argumentos da função é feita através do elemento `Args` que, por sua vez, é formado por quatro elementos de tag `Arg`. Elementos como `Args`, que podem conter outros elementos, possuem *tipos complexos* enquanto que os tipos de elementos que não podem conter outros elementos nem atributos são denominados elementos de *tipos simples* (estas definições estão associadas ao esquema do documento, discutido na seção seguinte). Atributos sempre têm valores de tipos simples.

O elemento `Args` define que a função `processImage` aceita quatro argumentos, sendo três de entrada e um de saída. O argumento `i3` é opcional e, caso não seja especificado, assume o valor default `99`.

Finalmente, o elemento `Source` especifica um arquivo que contém a implementação da função.

A constituição dos tipos complexos e de alguns tipos simples (existe um conjunto de tipos simples pré-definidos) utilizados em um documento é especificada no esquema associado ao documento. Esta associação esquema-documento normalmente é definida pelo próprio documento através de um tag específico para este fim (em nosso exemplo, o esquema não está especificado no documento).

3.2 XML Schema

A norma *XML Schema* ([6], [7], [8]) especifica uma linguagem, baseada em XML, para a definição de esquemas de representação de dados. O propósito de um esquema XML é definir e descrever uma classe de documentos XML através da utilização dos

componentes da linguagem para restringir e documentar o significado, a utilização e as relações entre as partes constituintes do documento: tipos de dados, elementos e seu conteúdo e atributos e seus valores. Em resumo, um esquema XML define um vocabulário para uma classe de documentos XML.

O esquema seguinte define um vocabulário para a definição de funções C. O documento XML apresentado anteriormente está em conformidade com este fragmento de esquema.

```
<schema>
<element name='AdCFunction' type='AdCFunctionType'/>
 <complexType name='AdCFunctionType'>
     <attribute name='name' type='FunctionID'/>
     <element name='Return'>
         <complexType>
             <attribute name='name' type='string'/>
             <attribute name='type' type='typeRef'/>
             <attribute name='output' type='boolean'/>
         </complexType>
     </element>
     <element name='Args'>
         <complexType>
             <element name='Arg' minOcurrs='0' maxOcurrs='unbounded'>
                 <complexType>
                     <attribute name='name' type='string'/>
                     <attribute name='type' type='typeRef'/>
                     <attribute name='dir' type='ArgDir'/>
                     <attribute name='optional' type='boolean'/>
                     <attribute name='default' type='string'/>
                     <attribute name='constraint' type='string'/>
                 </complexType>
             </element>
         </complexType>
     </element>
     <element name='Source'>
         <complexType content='mixed'>
             <attribute name='file' type='FileID' minOcurrs='0'/>
         </complexType>
```

```
     </element>
</complexType>

<simpleType name="FunctionID" base="string">
    <maxLength value="32"/>
</simpleType>

<simpleType name="ArgDir" base="string">
    <enumeration value="in"/>
    <enumeration value="out"/>
    <enumeration value="inout"/>
</simpleType>
...
</schema>
```

O esquema é formado pelo elemento `schema` e seus sub-elementos, notadamente os elementos `complexType, simpleType` e `element`.

A declaração `<element name="AdCFunction" type="AdCFunctionType"/>` indica que a constituição do elemento `AdCFunction` (raiz de nosso documento) é ditada pela definição do tipo `AdCFunctionType`. A construção que se segue é a definição deste tipo.

`AdCFunctionType` é um tipo complexo, definido com ajuda do elemento `complexType`: tem apenas um atributo, `name`, que é do tipo `FunctionID`, definido na sequência com o auxílio do elemento `simpleType`.

Esta definição estabelece que o valor do atributo é uma `string` (tipo simples predefinido) limitada a 32 caracteres. Este tipo de definição é denominada *derivação por restrição* (`FunctionID` é, ainda, uma `string` porém restrita a, no máximo, 32 caracteres).

Além do atributo, o tipo complexo `AdCFunctionType` (e portanto o elemento `AdCFunction`) é formado por uma sequência de três elementos `Return`, `Args` e `Source`. Cada um destes elementos é, por sua vez, um tipo complexo. Note, no entanto, que estes tipos são definidos imediata e anonimamente, isto é, sem especificar um nome para o tipo. O formato da definição é similar.

3.3 XPath

A estrutura de um documento XML pode ser comparada à estrutura de um sistema de arquivos tradicionalmente utilizado nos sistemas operacionais. Ambos podem ser representados por uma árvore onde os nós intermediários são os diretórios, no caso do sistema de arquivos, ou os elementos, no caso do XML. As folhas desta árvore representam os diferentes tipos de arquivos ou o texto e atributos do XML.

O seguinte documento XML pode ser representado como na figura 3.1.

```
<?xml version="1.0"?>
<AdCFunction name="processImage">
    <!-- função criada por RXF -->
    <Return type="adImage" name="out">
        Soma das imagens (out = i0 + i1)
    </Return>
    <Args>
        <Arg name="i0" type="adImage"/>
        <Arg name="i1" type="adImage"/>
        <Arg name="i2" type="adImage">
            i2 = out = i0 + i1
        </Arg>
    </Args>
    <Source file="srcs/processImage.c"/>
    Esta função efetua ...
</AdCFunction>
```

A figura 3.1 mostra os diferentes tipos de nós definidos na especificação da linguagem XML, com exceção do nó tipo *namespace*, não utilizado em nosso documento. Em nossa representação gráfica incluimos, ainda, um nó raiz pai de todos os demais. Os seis tipos de nós de uma árvore XML são os seguintes.

1. **Elementos**
   Os elementos estruturam a árvore XML, visto que são os únicos nós que podem ter filhos. Correspondem aos *tags* da linguagem.

2. **Texto**
   Estes nós contêm o texto contido nos elementos XML.

3. **Atributos**
   Estes nós correspondem aos pares nome/valor especificados nos tags de abertura dos elementos XML.

4. **Instruções de Processamento**
   No documento XML, aparecem como construções do tipo `<?...?>`

5. **Comentários**

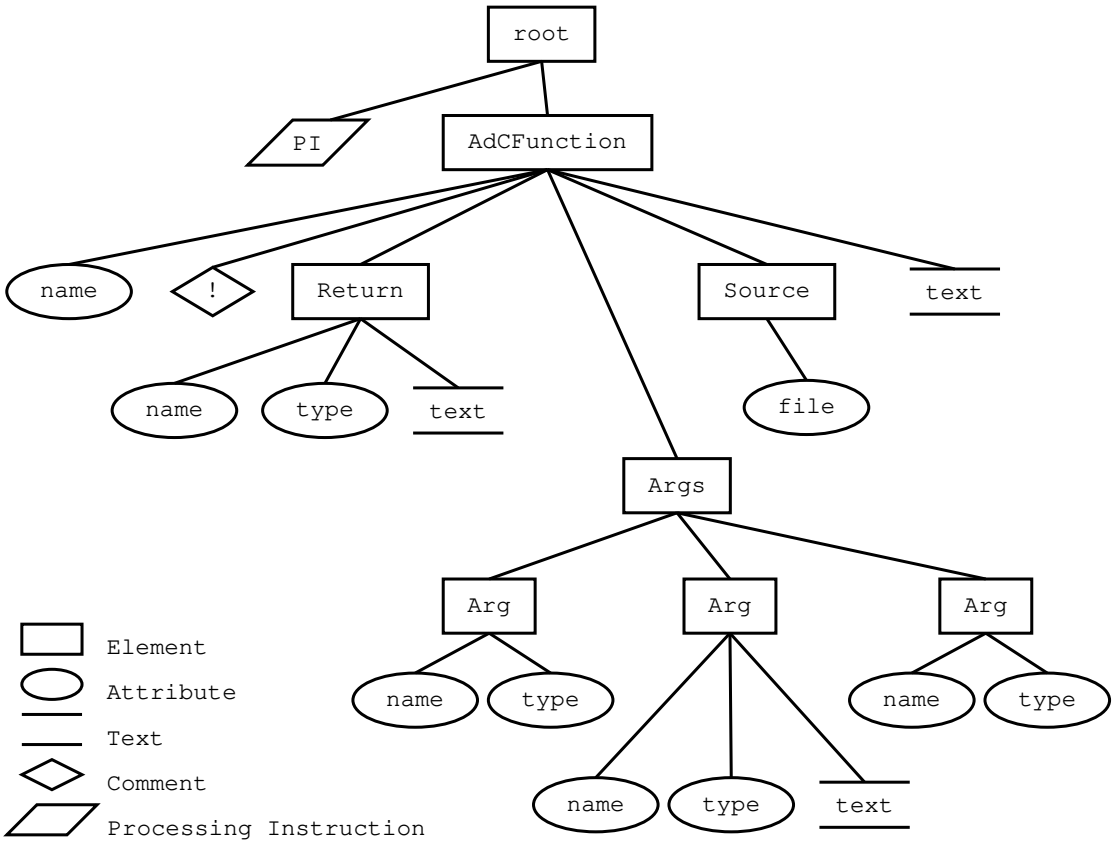

Figura 3.1: Árvore XML

São representados como texto do tipo `<!- ... ->` no documento.

6. **Espaços de Nomes** (não aparecem na figura)
   Estes nós contêm a definição de espaços de nomes usados para caracterizar e diferenciar os nomes de tags em diferentes contextos.

O **XPath**, [12], é uma forma de localizar partes de um documento XML. O nome se origina de sua semelhança com os caminhos utilizados para localização de arquivos e diretórios em um sistema de arquivos.

A construção sintática básica de uma expressão **XPath** é o *location path* ou **caminho**. À maneira do caminho de um arquivo, um *location path* pode ser absoluto, quando o caminho começa pelo nó raiz, ou relativo, no caso da referência ser o *nó corrente*. O *nó corrente* é estabelecido pela aplicação que se utiliza do **XPath**; por exemplo, em um processador XSLT o nó corrente é aquele que constitui o contexto onde um *template* é avaliado.

Um **caminho** é formado por uma série de *location steps* ou **passos**, um para cada nível da árvore XML, conectados pelo caracter `/`.

Cada um desses **passos** é constituído por três componentes:

- um **eixo** (*axis*) que define o relacionamento entre cada passo
- um **teste** (*node test*) que especifica o tipo de nó a ser selecionado e
- um **predicado** (*predicate*), opcional, que filtra ainda mais o conjunto de nós selecionados.

O eixo é separado do teste pela string `::` e o predicado é contido entre colchetes, resultando na seguinte sintaxe para cada um dos passos:

```
axis::node-test[predicate]
```

A especificação **XPath** define os seguintes eixos:

- `self`
  refere-se ao nó corrente

- `parent`
  seleciona o pai do nó corrente

- `child`
  seleciona os filhos do nó corrente

- `attribute`
  refere-se aos atributos do nó corrente

- `ancestor`
  seleciona os ancestrais (pai, avô etc.) do nó corrente

- `descendant`
  seleciona os descendentes (filho, neto etc.) do nó corrente

- `ancestor-or-self`
  além dos ancestrais, seleciona também o nó corrente

- `descendant-or-self`
  além dos descendentes, seleciona também o nó corrente

- `following`
  seleciona todos os nós do documento posteriores ao nó corrente na ordem de leitura do documento

- `preceding`
  seleciona todos os nós do documento que precedem o nó corrente na ordem de leitura do documento

- `following-sibling`
  seleciona todos os irmãos posteriores ao nó corrente na ordem de leitura do documento

- `preceding-sibling`
  seleciona todos os irmãos que precedem o nó corrente na ordem de leitura do documento

- `namespace`
  seleciona nós do tipo *namespace*

Os seguintes testes (*node tests*) são predefinidos: `text()`, que seleciona nós de texto, `comment()`, comentários, `processing-instruction()`, instruções de processamento e `node()`, um teste genérico que é satisfeito por qualquer nó. Outra forma de teste é através da especificação de um nome que identifica um nó (por exemplo, o nome de um atributo). Neste caso, o teste resulta verdadeiro se o tipo do nó identificado pelo nome é igual ao tipo principal do eixo. O teste `*` é verdadeiro para todos os nós do tipo principal do eixo. Cada eixo tem um tipo principal de nó:

- eixo `attribute`: o tipo principal é *atributo*

- eixo `namespace`: o tipo principal é *namespace*

- demais eixos: o tipo principal é *elemento*

Os exemplos a seguir ilustram a utilização de eixos e testes. Os comandos são aplicados ao texto XML representado na figura 3.1:

- `/child::AdCFunction/child::Return`
  seleciona o nó que contém o elemento `Return`. Note a utilização de nomes no teste (`AdCFunction` e `Return`) que, no caso, referem-se aos *tags* dos elementos, já que *elemento* é o tipo principal do eixo.

- `/child::AdCFunction/child::Return/attribute::name`
  seleciona o atributo `name` do elemento `Return`. Aqui, o nome utilizado no teste do eixo `attribute` refere-se a um *atributo*.

- `/descendant-or-self::*`
  seleciona todos os elementos do documento

- `parent::node()`
  se o nó corrente é um dos elementos `Arg`, refere-se ao elemento `Args` situado um nível acima

- `/child::processing-instruction()`
  seleciona a instrução de processamento `<?xml version="1.0"?>`

Para facilitar a formação de caminhos, a norma permite o uso de uma **sintaxe abreviada** que torna as expressões muito parecidas com caminhos de um sistema de arquivos. As seguintes abreviaturas são permitidas:

- um eixo `child::` pode ser omitido:
  - `/child::AdCFunction/child::Return`
    pode ser reescrito como `/AdCFunction/Return`

- o eixo `attribute::` pode ser escrito como `@`
  - `/child::AdCFunction/child::Return/attribute::name`
    é o mesmo que `/AdCFunction/Return/@name`

- `self::node()` é abreviado como `.`

- `parent::node()` pode ser resumido como `..`

- a sequência `//` abrevia o caminho `/descendant-or-self::node()/`
  - se o contexto é dado pelo elemento `Return`, o caminho
    `..//Arg` é o mesmo que
    `parent::node()/descendant-or-self::node()/child::Arg`:
    seleciona todos os elementos `Arg` situados abaixo de `AdCFunction`

O terceiro componente de um passo do caminho é, como vimos, o predicado. Um predicado filtra um conjunto de nós (*node-set*), relativamente a um eixo, para produzir um novo conjunto de nós. Para cada nó do conjunto inicial, uma expressão, definida pelo predicado, é avaliada usando este nó como contexto. Se o resultado é verdadeiro, o nó é incluído no conjunto resultante. Consideremos a expressão:

```
/AdCFunction/Args/Arg
```

O resultado é a seleção dos três nós `Arg` de nosso documento. Especifiquemos um filtro adicional através de um predicado:

```
/AdCFunction/Args/Arg[@name="i1"]
```

A seleção, agora, inclui apenas o elemento `Arg` com atributo `name` igual a `i1`.

Seguem-se alguns exemplos de *queries* utilizando a notação XPath.

- `para` — seleciona os elementos filhos do nó corrente com *tag* `para`
- `*` — seleciona todos os elementos filhos do nó corrente
- `text()` — seleciona todos os nós de texto imediatamente abaixo do nó corrente
- `@name` — seleciona o atributo `name` do nó corrente
- `@*` — seleciona todos os atributos do nó corrente
- `para[1]` — seleciona o primeiro elemento-filho com *tag* `para` do nó corrente
- `para[last()]` — seleciona o último filho do nó corrente
- `*/para` — seleciona todos os netos do nó corrente com *tag* `para`
- `/doc/chapter[5]/section[2]` — seleciona o segundo elemento `section` filho do quinto elemento `chapter`, filho de `doc`
- `chapter//para` — seleciona os descendentes de `chapter`, filho do nó corrente, de *tag* `para`

- `.` — seleciona o nó corrente
- `.//para` — seleciona os descendentes do nó corrente com *tag* `para`
- `..` — seleciona o pai do nó corrente
- `para[@type="warning"]` — seleciona os descendentes do nó corrente com *tag* `para` que tenham um atributo de nome `type` com o valor `warning`

3.4 XSL

A linguagem XSL (*Extensible Style Language*, [9]) é constituída por dois componentes: uma linguagem de transformação (XSLT, [13]) e uma linguagem de formatação. Ambas as linguagens são definidas como uma aplicação XML, ou seja, utilizam a sintaxe XML. A linguagem de transformação provê construções para especificar a transformação de um documento XML em outro documento XML. A segunda linguagem especifica tags para caracterizar *objetos de formatação*.

Examinaremos em seguida algumas características da linguagem de transformação cujos princípios são utilizados como base para a criação de geradores de código do Adesso. Para exemplificar o uso do XSL, consideremos que queremos criar um documento HTML que documente a função C descrita em XML na seção anterior.

A figura 3.2 apresenta o modelo para uma transformação XSL. Um processador XSL aceita como entradas o documento XML a ser transformado e uma *folha de estilo,* escrita em XSLT, e cria um novo documento com a marcação desejada. O código a seguir é um exemplo de folha de estilo para tranformar a definição de uma função C como a apresentada anteriormente em um documento HTML.

```
<xsl:stylesheet>
  <xsl:template match="AdCFunction">
      <html>
          <body>
              <h1>Função <b><xsl:value-of select="@name"/></b></h1>
```

```
            <xsl:value-of select="text()"/>
            <h2>Argumentos</h2>
            <xsl:apply-templates select="Args"/>
            <h2>Retorno</h2>
            <xsl:apply-templates select="Return"/>
        </body>
    </html>
  </xsl:template>

<xsl:template match="Args">
    <dl>
        <xsl:apply-templates select="Arg"/>
    </dl>
  </xsl:template>

<xsl:template match="Arg">
    <dt>
        <xsl:value-of select="@type"/> <xsl:value-of select="@name"/>
    </dt>
    <dd>
        <xsl:value-of select="text()"/>
    </dd>
```

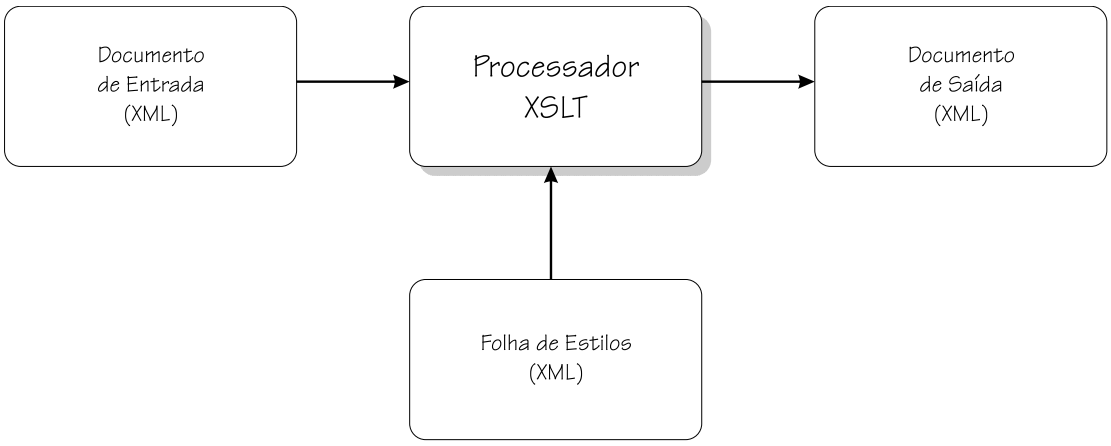

Figura 3.2: Processamento de Estilo

```
  </xsl:template>
</xsl:stylesheet>
```

O processador XML realiza sua função através da aplicação de templates a nós da árvore correspondente ao documento XML de entrada. Cada definição de template indica, através do atributo `match`, os elementos do documento a que ela se aplica. Durante o processamento existe sempre um *nó corrente* que define o contexto para a realização de buscas.

O processamento de nosso estilo se inicia com o template que tem o atributo `match` com o valor `AdCFunction,` que corresponde ao elemento raiz do documento XML. A árvore XML contida no template é transferida para o documento de saída depois que os elementos da linguagem XSLT, xsl:*, são processados.

Os elementos com tag *xsl:value-of* ocasionam a seleção de algum componente do documento de entrada, indicado no atributo *select*, e a sua substituição pelo valor do componente. Assim, por exemplo, o elemento `<xsl:value-of select="@name"/>`
faz com que seja inserido no documento de saída o valor do atributo `name` (@x indica o atributo x) do nó em processamento, no caso o nó AdCFunction.

Os elementos com tag *xsl:apply-templates* fazem com que os templates definidos no estilo sejam aplicados aos nós filhos do nó corrente, de acordo com seu atributo `match`. O elemento `<xsl:apply-templates select="Args"/>` causa a aplicação do template `<xsl:template match="Args"> ... </xsl:template>` ao nó Args, filho de AdCFunction.

O documento gerado como saída do processamento é apresentado a seguir.

```
<html>
    <body>
        <h1>Função <b>processImage</b></h1>
        Esta função efetua ...
        <h2>Argumentos</h2>
        <dl>
            <dt>adImage i0</dt><dd></dd>
```

```
            <dt>adImage i1</dt><dd></dd>
            <dt>adImage i2</dt><dd>i2 = out = i0 + i1</dd>
            <dt>adInt i3</dt><dd></dd>
        </dl>
        <h2>Retorno</h2>
        ...
    </body>
</html>
```

Capítulo 4

O Processador de Estilos

4.1 O Processo de Transformação

O **Processador de Estilos** do Adesso é o elemento central da arquitetura do processo de transformação do Adesso. É baseado em um processo de transformação de documentos XML inspirado na linguagem XSL (*Extensible Style Language*).

O modelo de transformação utilizado pelo processador de estilos do Adesso é baseado em processos de substituição de texto à maneira dos processadores de macros. O modelo aproveita o sistema de substituição implementado pelo parser da linguagem *Tcl*, que é a linguagem de implementação do Adesso, e faz uso intensivo da interface *DOM* e da linguagem de query *XPath*. O resultado dessa combinação é um sistema poderoso de tranformação de documentos XML, que será descrito nesta seção.

Dois motivos levaram à implementação deste processador de estilos em vez de aproveitar o suporte oferecido pelo XSL. À época desta decisão, a linguagem XSLT ainda não se encontrava estabilizada e era descrita apenas em documentos de trabalho muito instáveis. Por outro lado, a necessidade de geração de código, makefiles e outras saídas não convencionais sugeria a necessidade da flexibilidade oferecida pelas linguagens de scripting e o casamento natural Tcl/XML encorajou a criação do processador de estilos.

Atualmente, consideramos que, mesmo com a crescente ampliação da oferta de processadores *XSLT*, o processador de estilos do Adesso e sua linguagem constituem uma alternativa mais flexível para a especificação de transformações, principalmente para programadores com experiência na linguagem *Tcl*.

O *XSLT*, porém, deve se tornar uma linguagem padrão para transformação de documentos. Com o surgimento de pessoal treinado em *XSLT*, deve haver maior utilização do XSLT para a criação de estilos para o Adesso.

No final desta seção do documento, apresentamos uma comparação entre a linguagens de estilos do Adesso e o XSLT.

4.1.1 Representação de um documento XML

Um documento XML bem formado pode ser representado como uma árvore. Uma árvore é uma estrutura de dados composta de nós conectados que têm origem em um nó chamado raiz. O nó raiz é conectado a seus nós filhos que, por sua vez, se conectam a seus próprios filhos e assim por diante. Os nós que não têm filhos são denominados folhas da árvore. A principal propriedade de uma árvore é que qualquer de seus nós, juntamente com seus nós decendentes, formam uma árvore válida.

O processador de estilos do Adesso utiliza o mesmo modelo conceitual do documento utilizado pelo **XPath**, apresentado na seção 3.3.

Consideremos, como exemplo, o seguinte documento XML:

```
<PERIODIC_TABLE>
    <ATOM STATE="GAS">
        <NAME>Hydrogen</NAME>
        <SYMBOL>H</SYMBOL>
        <ATOMIC_NUMBER>1</ATOMIC_NUMBER>
        <ATOMIC_WEIGHT>1.00794</ATOMIC_WEIGHT>
        <BOILING_POINT UNITS="Kelvin">20.28</BOILING_POINT>
        <MELTING_POINT UNITS="Kelvin">13.81</MELTING_POINT>
        <DENSITY UNITS="grams/cubic centimeter">
```

```
            <!-- At 300K, 1 atm -->
            0.0000899
        </DENSITY>
    </ATOM>

    <ATOM STATE="GAS">
        <NAME>Helium</NAME>
        <SYMBOL>He</SYMBOL>
        <ATOMIC_NUMBER>2</ATOMIC_NUMBER>
        <ATOMIC_WEIGHT>4.0026</ATOMIC_WEIGHT>
        <BOILING_POINT UNITS="Kelvin">4.216</BOILING_POINT>
        <MELTING_POINT UNITS="Kelvin">0.95</MELTING_POINT>
        <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
            0.0001785
        </DENSITY>
    </ATOM>
</PERIODIC_TABLE>
```

A figura 4.1 mostra a árvore utilizada como modelo pelo processador de estilos do Adesso.

4.1.2 Modelo de processamento

O processo de transformação opera a partir desta representação do documento de entrada para gerar um ou mais documentos de saída de acordo com instruções fornecidas por um outro documento XML denominado *folha de estilo*. O documento de entrada deve ser, sempre, um documento XML, assim como a folha de estilo. Os documentos gerados não são, necessariamente, documentos XML. A figura 4.2 ilustra este modelo.

Uma **folha de estilo** do Adesso é um documento XML formado por um conjunto de elementos denominados *templates*. No processo de transformação, o processador caminha pelo documento de entrada procurando, para cada elemento deste documento, um template aplicável. Quando isso ocorre, o texto do template é incluido na saída depois

de efetuadas determinadas substituições, à maneira dos processadores de macros. Estas substituições formam a base da *linguagem de estilos* do Adesso.

Uma folha de estilos é um fragmento de documento contido sob o elemento `AdStylesheet`. Este elemento é um *container* para os templates, elementos `AdTemplate`. O *container* tem dois atributos, `lang` e `target`, que, tomados em conjunto, identificam a folha de estilo.

Cada template tem um atributo `name` que identifica elementos do documento de entrada aos quais se aplica o conteúdo do template. Consideremos a seguinte folha de estilo:

```
<AdStylesheet target="all" lang="example">
    <AdTemplate name="PERIODIC_TABLE">
```

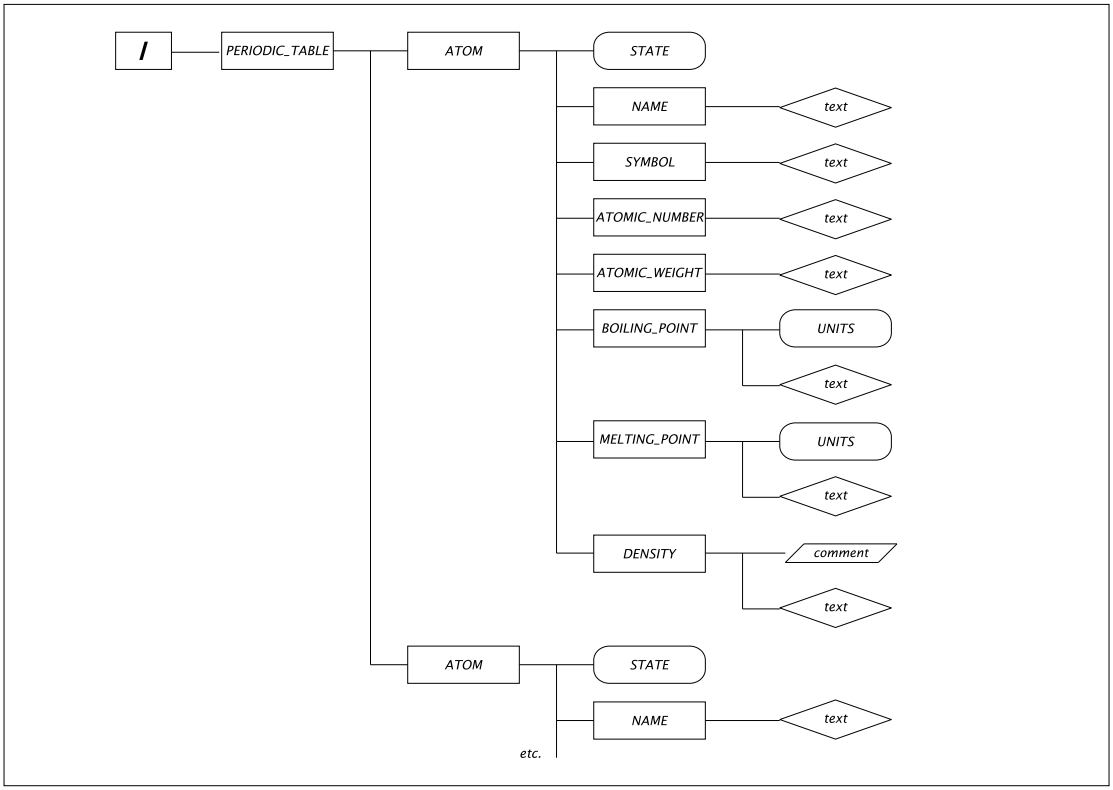

Figura 4.1: Árvore utilizada para representar um documento XML

```
        [sty:tag html]
        [sty:tag ol]
            [sty:apply ATOM]
        [sty:tag /ol]
        [sty:tag /html]
    </AdTemplate>
    <AdTemplate name="ATOM">
        [sty:tag li][sty:tag p]
           The boiling point of [sty:value NAME] is \
           [sty:value BOILING_POINT] degrees \
           [sty:value BOILING_POINT/@UNITS]
        [sty:tag /p][sty:tag /li]
    </AdTemplate>
</AdStylesheet>
```

A aplicação deste estilo ao documento apresentado anteriormente resulta no seguinte documento de saída:

```
<html>
<ol>
    <li><p>
        The boiling point of Hydrogen is  20.28 degrees Kelvin
```

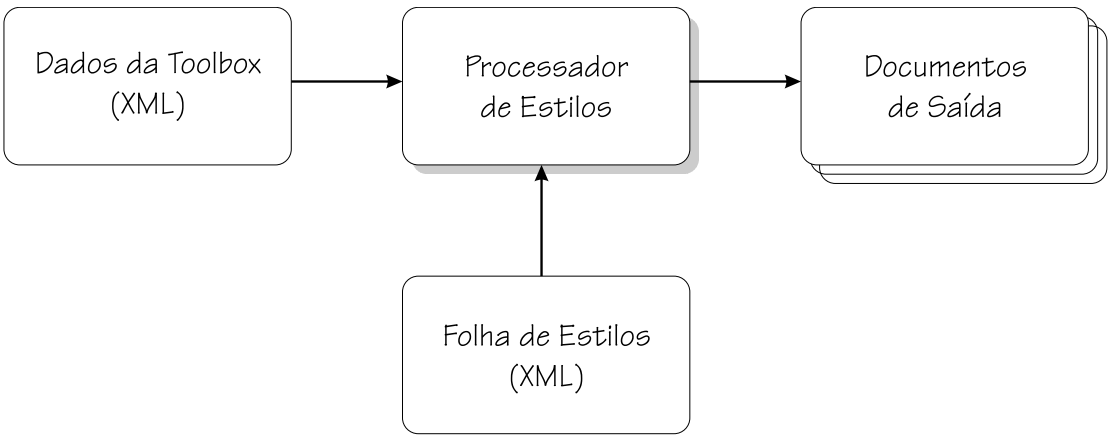

Figura 4.2: O Processador de Estilos

```
    </p></li>
    <li><p>
        The boiling point of Helium is  4.216 degrees Kelvin
    </p></li>
</ol>
</html>
```

Acompanhemos passo a passo a operação do processador de estilos:

1. Inicialmente, o processador de estilos procura um template aplicável ao **elemento-raiz** do documento de entrada. O elemento-raiz de nosso documento é `PERIODIC_TABLE`. Varrendo a nossa folha de estilo, encontra o template `<AdTemplate name="PERIODIC_TABLE">` que se aplica ao nó. O documento de saída é gerado, então, com o texto contido no template selecionado.

2. Antes de constituir a saída do processo, o texto do template é processado visando a substituição dos fragmentos de texto contidos entre colchetes, `[...]`. A primeira palavra do texto entre colchetes é interpretada como um **comando**, que é executado com o restante do texto como argumentos. A string retornada pelo comando é inserida no texto em substituição ao texto entre colchetes.

   O primeiro comando encontrado é `[sty:tag html]`. Este é um comando auxiliar para gerar *tags* XML/HTML: retorna a string `<html>` que passa a constar do documento de saída. Em seguida, o comando `[sty:tag ol]` produz a string `<ol>` na saída.

3. O comando mais importante deste template é o comando `[sty:apply ATOM]`. O argumento deste comando é uma expressão **XPath** que, avaliada tendo o nó corrente como contexto, deve resultar em um conjunto de elementos da árvore de entrada. O comando `sty:apply` então procura aplicar, para cada elemento resultante do query, um template especificado no estilo.

   Se encontra um template adequado, o processo se repete com a substituição do texto do novo template, agora em novo contexto dado pelo elemento selecionado no query XPath.

4. Em nosso exemplo, o query `ATOM` resulta na seleção dos dois elementos filhos do elemento corrente com tag `ATOM`. Assim, o commando `[sty:apply ATOM]` resulta na aplicação do template `<AdTemplate name="ATOM">` para os dois elementos do documento de tag `ATOM`, filhos do elemento-raiz.

5. A primeira aplicação do template se dá no contexto do primeiro filho do elemento-raiz, o átomo de nome *Hydrogen*.

6. A avaliação do template inicia com a emissão dos tags `<li><p>`. Em seguida, o texto `The boiling point of` é emitido.

7. O comando `[sty:value NAME]` é executado. Este comando retorna o valor string dos nós resultantes do query XPath especificado em seu argumento. O valor string de um elemento é a concatenação de todos os nós de texto diretamente subordinados ao elemento.

   No caso, o texto associado ao elemento `NAME` da definição do átomo de hidrogênio é inserido no fluxo de saída, seguido da string `" is "` e do texto do elemento `BOILING_POINT`.

8. O comando `[sty:value BOILING_POINT/@UNITS]` retorna o valor do atributo `UNITS` associado ao elemento `BOILING_POINT`. A sintaxe do **XPath** para se referir a um atributo é sempre `@` seguido do nome do atributo.

9. Os tags abertos pelo template são fechados e o processamento do template `ATOM` para o átomo *Hydrogen* é finalizado. O procedimento é, então, repetido para o segundo átomo do documento, *Helium*.

10. Depois de processados os dois elementos `ATOM`, o processamento continua no ponto seguinte à invocação do template `ATOM`, no template `PERIODIC_TABLE`, onde os tags abertos são fechados e o documento de saída é emitido como resultado do processamento.

Variável	Descrição
`node`	identificador do nó corrente
`nodelist`	lista de nós resultante da expressão **XPath** do comando `sty:apply`
`index`	índice do nó corrente na lista de nós

Tabela 4.1: Contexto para avaliação de templates

4.2 A Linguagem de Estilos

4.2.1 Avaliação de Templates

A maneira principal de controlar a avaliação dos templates que compoem a folha de estilos é através do comando `sty:apply`, como vimos no exemplo anterior. O comando `sty:apply` espera um argumento que constitua uma expressão **XPath** que, uma vez avaliada, resulte em um conjunto de nós da árvore de entrada. Para cada nó deste conjunto, o processador de estilos procura, na folha de estilos, o template mais adequado. As próximas seções devem esclarecer o significado do termo *template mais adequado*.

Uma vez escolhido o template a aplicar, o processador ajusta um pequeno conjunto de variáveis, tabela 4.1, que constituem o contexto para a avaliação do template e procede a substituição dos comandos contidos no template. Neste processo, podem ocorrer novos comandos `sty:apply`, que são aplicados recursivamente. Uma pilha (*stack*) de variáveis de contexto é utilizada para gerenciar contextos aninhados.

Outra maneira de processar documentos é através do comando `sty:foreach`. Como no comando `sty:apply`, ele executa um *query* XPath para obter um conjunto de nós aos quais um template deve ser aplicado. O template é, contudo, especificado no próprio comando, evitando o processo de busca. Vejamos um exemplo. O estilo apresentado anteriormente pode ser re-escrito da seguinte forma, gerando a mesma saída.

```
<AdStylesheet target="all" lang="example">
    <AdTemplate name="PERIODIC_TABLE">
        [sty:tag html]
        [sty:tag ol]
```

```
        [sty:foreach ATOM {
            [sty:tag li][sty:tag p]
                The boiling point of [sty:value NAME] is \
                [sty:value BOILING_POINT] degrees \
                [sty:value BOILING_POINT/@UNITS]
            [sty:tag /p][sty:tag /li]
        }]
        [sty:tag /ol]
        [sty:tag /html]
    </AdTemplate>
</AdStylesheet>
```

Esta folha de estilos apresenta uma maneira diferente de se usar a linguagem de estilos se comparada ao estilo apresentado antes (`sty:apply` vs. `sty:foreach`). A escolha de uma forma ou outra depende, principalmente, do documento de entrada. Na maior parte das vezes, a forma escolhida é uma mistura das duas.

Documentos XML podem ser classificados entre dois extremos representados pelos seguintes tipos de documentos:

- Documentos cuja organização física (sequência do texto) determina o fluxo de saída dos processos de transformação. Este é o caso típico dos documentos onde a sequência de capítulos e seções se mantém depois de uma transformação.
- Documentos cuja organização física é irrelevante para o processo de transformação. Este caso é o de documentos que representam bases de dados e a tranformação é um processo de extração de dados e formatação de relatórios ou fomulários de entrada de dados (*views*).

Documentos que representam bases de dados, como em nossa tabela periódica, são tratados mais eficientemente com `sty:foreach`, sendo que o mesmo nó pode dar origem a diferentes representações na saída. Outra alternativa útil é a utilização de templates em diferentes modos (atributo `section`). Esta abordagem pode ser denominada *dirigida pelo estilo*, visto que a determinação da ordem e a escolha dos dados do documento de saída é feita pelo estilo.

O primeiro tipo de documento favorece a utilização do comando `sty:apply`, principalmente em sua forma `sty:apply *`, em que a ordem dos nós selecionados é a ordem do documento e a seleção contém elementos de diferentes *tags*. Esta forma de programação pode ser chamada de *dirigida pelo documento*. As listagens seguintes apresentam um exemplo desta abordagem. Note que o estilo é, neste caso, menos sensível a alterações no conteúdo do documento de entrada que, caracteristicamente, apresenta elementos do tipo *mixed*, com texto entremeado de tags.

```
<!-- Fragmento de um documento -->
<Document name="O Processador de Estilos do Adesso">
    <Section name="O Processo de Transformação">
        <Para>
            O <emphasis>Processador de Estilos</emphasis> do Adesso é ...
        </Para>
        <Para>
            O modelo de transformação utilizado pelo ...
        </Para>
    </Section>
    <Section name="Modelo de processamento">
        <Para>O processo de transformação opera a partir ...
            <Itemize>
                <Item>Um processador <strong>XSLT</strong> opera ...</Item>
            </Itemize>
        </Para>
    </Section>
</Document>

<!-- Folha de estilo -->
<AdStylesheet target="html" lang="stxdoc">
    <AdTemplate name="Document">
        [sty:tag !-- Documento resultante --]
        [sty:tag html]
            [sty:tag h1][sty:value @name][sty:tag /h1]
            [sty:apply * "" lev=2]
        [sty:tag /html]
    </AdTemplate>
    <AdTemplate name="Section">
```

```
        [sty:tag h[sty:par lev]][sty:value @name][sty:tag /h[sty:par lev]]
        [sty:apply * "" lev=[expr [sty:par lev] + 1]]
    </AdTemplate>
    <AdTemplate name="Para">
        [tag p][sty:apply *][tag /p]
    </AdTemplate>
    <AdTemplate name="Itemize">
        [sty:tag ul][sty:apply Item][sty:tag /ul]
    </AdTemplate>
    <AdTemplate name="Item">
        [sty:tag li][sty:apply *][sty:tag /li]
    </AdTemplate>
    <AdTemplate name="emphasis">
        [sty:tag i][sty:apply *][sty:tag /i]
    </AdTemplate>
    <AdTemplate name="strong">
        [sty:tag b][sty:apply *][sty:tag /b]
    </AdTemplate>
</AdStylesheet>
```

```
<!-- Documento resultante -->
<html>
<h1>O Processador de Estilos do Adesso</h1>
<h2>O Processo de Transformação</h2>
    <p>O <i>Processador de Estilos</i> do Adesso é ... </p>
    <p>O modelo de transformação utilizado pelo ... </p>
<h2>Modelo de processamento</h2>
    <p>O processo de transformação opera a partir ... </p>
    <ul>
        <li>Um processador <b>XSLT</b> opera ...</li>
    </ul>
</html>
```

4.2.2 Especificação de Templates

Um template é formado pelo conteúdo de um elemento `AdTemplate`. O atributo `name` deste elemento especifica os nós do documento de entrada aos quais o template se aplica. Tal especificação é feita através do tag do elemento, podendo incluir os tags de seus ancestrais a fim de especificar sua posição na árvore. Em sua implementação atual, a linguagem de estilos do Adesso só permite a especificação de templates aplicáveis a nós que representam elementos do documento de entrada. Por exemplo:

- `<AdTemplate name="NAME">` especifica um template que se aplica a qualquer nó de tag `NAME`
- `<AdTemplate name="ATOM/NAME">` se aplica apenas a nós `NAME` que sejam subordinados imediatos de um nó `ATOM`

Além do atributo `name`, o elemento `AdTemplate` pode ter um segundo atributo, denominado `section`. Este atributo é util em situações em que a informação contida em determinado nó da árvore de entrada pode gerar mais de uma representação na saída, dependendo de como o template é aplicado. Consideremos o seguinte trecho de uma folha de estilos:

```
<AdTemplate name="BOILING_POINT">
    The boiling point of [sty:value ../NAME] is \
    [sty:value .] degrees [sty:value @UNITS]
</AdTemplate>


<AdTemplate name="BOILING_POINT" section="units">
    [sty:value @UNITS]
</AdTemplate>
```

- A execução de um comando `[sty:apply BOILING_POINT]` ocasiona a avaliação do primeiro template, incluindo uma string do tipo
  `The boiling point of Hydrogen is 20.28 degrees Kelvin` no fluxo de saída.

- Se a seleção é feita pelo comando `[sty:apply BOILING_POINT units]`, o segundo template é invocado. Neste caso, ele insere apenas o identificador da unidade de temperatura, `Kelvin`.

Na procura por um template aplicável a determinado nó, pode ocorrer que dois ou mais templates sejam aplicáveis. Por exemplo, os templates:

- `<AdTemplate name="BOILING_POINT">` e
- `<AdTemplate name="ATOM/BOILING_POINT">`

são ambos aplicáveis a um nó do tipo `BOILING_POINT` que seja filho de `ATOM`. O segundo template não se aplica, contudo, a nós `BOILING_POINT` que não sejam filhos de `ATOM`.

As regras para a escolha do template a aplicar são as seguintes:

- Prevalece o template com o atributo `name` mais completamente especificado no que se refere à posição hierárquica do nó na árvore. Assim, no exemplo anterior, a escolha recairia sobre o template com `name="ATOM/BOILING_POINT"`.
- No caso dos atributos `name` serem iguais, prevalece o template cuja posição seja a última, considerando a ordem em que o documento de estilos é lido.
- No caso de haver folhas de estilo incluídas, os templates definidos no documento principal têm preferência (veja a seção 4.3).

4.2.3 Parâmetros de Templates

Para a seleção de templates do comando `sty:apply`, é possível a especificação de pares `nome=valor` que são passados como parâmentros ao template selecionado. O parâmetro é definido durante a execução da chamada e destruído após sua complementação. Ao longo da chamada o mesmo parâmetro pode ser utilizado com outros valores em novas seleções. Após a chamada, o parâmetro reassume seu valor anterior. Exemplo:

```
<AdStylesheet target="all" lang="example">
    <AdTemplate name="PERIODIC_TABLE">
            [sty:apply ATOM "" "" mypar=aaa]
    </AdTemplate>
    <AdTemplate name="ATOM">
        before: parameter mypar is [sty:par mypar]
        [sty:apply NAME "" "" mypar=bbb]
        after: parameter mypar is [sty:par mypar]
    </AdTemplate>
    <AdTemplate name="NAME">
        [sty:value .]: parameter mypar is [sty:par mypar]
    </AdTemplate>
</AdStylesheet>
```

O comando `sty:par` retorna o valor do parâmetro cujo nome é especificado em seu argumento. A execução deste estilo tendo como entrada o documento XML apresentado na seção 4.1.1 gera a seguinte saída:

```
before: parameter mypar is aaa
Hydrogen: parameter mypar is bbb
after: parameter mypar is aaa
before: parameter mypar is aaa
Helium: parameter mypar is bbb
after: parameter mypar is aaa
```

4.2.4 Macros

A fim de facilitar a reutilização de trechos de templates em diversos locais, a linguagem provê uma implementação de **macros**. Uma macro é especificada através do elemento `<AdMacro ...>`, dentro de uma folha de estilos. Em um template, a macro é utilizada através do comando `sty:call`, ocasionando a avaliação do texto da macro no mesmo contexto em que vinha sendo avaliado o template que contém a chamada.

Suponhamos, a título de exemplo, que a ocorrência do número atômico de um átomo no documento de saída deva ser formatada sempre usando um determinado fonte, cor e

tamanho. Suponhamos, ainda, que números atômicos ocorram em diversos trechos da saída. Uma boa solução para minimizar o código é a criação de uma macro, como na listagem seguinte. Note que a macro foi implementada para ser usada no contexto de um elemento `ATOM`.

```
<AdMacro name="format-atomic-nb">
    [sty:tag font face="Times" color="blue" size="2"]
        [sty:value ATOMIC_NUMBER]
    [sty:tag /font]
</AdMacro>
<AdTemplate name="ATOM" section="a1">
    <!-- ... -->
    [sty:call format-atomic-nb]
    <!-- ... -->
</AdTemplate>
```

4.2.5 Principais Comandos

- `sty:apply match ?section? ?options? ?parameters?`

  ```
  Options:
      -sort "xpath opts"
      -connect string
      -trim chars
  ```

  Avalia, a partir do nó corrente, a expressão xpath especificada em `match`, que deve retornar um conjunto de elementos.

  Se a opção `-sort` é especificada, ordena o conjunto de elementos de acordo com o valor da opção, uma expressão XPath avaliada no contexto de cada elemento da lista. Tipicamente, esta expressão especifica um atributo do nó. As strings resultantes são ordenadas através do comando `lsort` do Tcl. `opts` pode especificar opções para este comando.

  Para cada elemento da lista de nós, procura um template aplicável. Ajusta o contexto para o nó em questão e aplica o template (executa o processo de substituição).

Concatena as strings resultantes da aplicação do template para cada nó da lista usando a string especificada na opção `-connect` ou uma string vazia, caso `-connect` não exista, como elemento de ligação entre as strings. A opção `-trim` especifica um conjunto de caracteres a serem suprimidos do início ou fim do texto retornado por um template. Retorna a string obtida com a concatenação dos resultados da aplicação do template a cada nó selecionado.

Parâmetros são especificados via strings da forma `par_name=par_value` e são ajustados de forma a serem acessíveis aos templates avaliados a partir deste comando, através do comando `sty:par`.

Exemplos:

- `[sty:apply //AdCFunction]`

  Seleciona todos os elementos `AdCFunction` contidos no documento de entrada. Aplica o template `AdCFunction` (com `section=""`).

- `[sty:apply //AdCFunction fname -sort "name -ascii" style=matlab]`

  Seleciona todos os elementos `AdCFunction` contidos no documento de entrada. Ordena os elementos de acordo com seus atributos `name`, na ordem da tabela *ASCII*. Aplica o template `AdCFunction.fname` a cada elemento, passando o parâmetro `style` com o valor `matlab`.

- `[sty:apply //ATOM fname -connect +]`

  Em nossa tabela periódica, supondo que o template `ATOM.fname` retorne o nome do átomo, o resultado deste comando é a string `'Hydrogen+Helium'`.

- `sty:foreach xpath ?options? ?parameters? { template }`

  Procede como no comando `apply`, aceitando as mesmas opções. Não efetua, porém, a busca por templates: aplica sempre o template especificado em seu argumento `template`. Normalmente, o template é especificado entre chaves, à maneira dos procedimentos *Tcl*, para evitar que o script seja avaliado pelo parser Tcl

quando da formação do comando. A presença das chaves deixa a avaliação para o comando propriamente dito.

Exemplo:

- `[sty:foreach //ATOM -connect ", " {[sty:value NAME]}`

  O resultado é a string `'Hydrogen, Helium'`.

- `sty:value xpath ?options?`

  Avalia, a partir do nó corrente, a expressão xpath especificada em `xpath`, que deve retornar um conjunto de elementos. Se a opção `-sort` é especificada, ordena a lista de nós de acordo com o valor da opção, uma expressão xpath avaliada no contexto de cada nó da lista.

  Retorna o **valor-string** dos nós resultantes, considerando as opções `-connect` e `-trim`, se especificadas. O valor-string de um nó é obtido através do operador XPath `string()` [12], cuja operação é resumida abaixo:

  - se a lista de nós é uma lista de elementos, avalia cada elemento, considerando que o valor-string de um elemento é a concatenação dos nós de texto imediatamente abaixo, conectados por uma string vazia; a concatenação dos valores de cada elemento considera o valor da opção `-connect`;
  - no caso da lista conter apenas um atributo, o valor-string é o valor do atributo;
  - no caso de lista de atributos, é a concatenação dos valores, usando um espaço como conector.

  Exemplo:

  - `[sty:value //ATOM/NAME -connect " + "]`

    O resultado é a string `'Hydrogen + Helium'`.

- `sty:if xpath1 body1 ?elseif xpath2 body2 ...?  ?else bodyN?`

Avalia a expressão `xpath1` e converte para um booleano com o operador `boolean()` da linguagem XPath. Se a expressão resulta verdadeira, retorna o texto `body1`, devidamente substituído. Caso contrário, avalia, da mesma forma, a expressão `xpath2`, se o primeiro `elseif` está presente e assim por diante. Se todos os testes falharem, retorna o texto processado de `bodyN`, se a cáusula `else` é especificada.

Resultados de expressões XPath são convertidos para booleanos da seguinte forma (veja [12]):

- um *número* é convertido para `true` se é diferente de zero;
- uma *lista de nós* resulta `true` se não é vazia;
- uma *string* é `true` se tem comprimento diferente de zero.

Exemplo:

- ```
  [sty:if {//ATOM[ATOMIC_NUMBER=2]} {
  [sty:value NAME] passed
  } else {
  [sty:value NAME] failed
  }]
  ```

  Gera textos diferentes de acordo com o valor do elemento `ATOM/ATOMIC_NUMBER`.

- `sty:iff tclexpr1 body1 ?elseif tclexpr2 body2 ...? ?else bodyN?`

  É similar ao comando `sty:if` mas avalia uma expressão **Tcl** ao invés da expressão XPath. A expressão é usada como argumento para o comando Tcl `expr`.

  Exemplo:

  - ```
    [sty:iff {[sty:par mypar] == 55]} {
    mypar is 55
    } else {
    ```

```
mypar is not 55
}]
```

Testa o valor do parâmetro `mypar`.

- `sty:par parname ?default?`
  `sty:par parname set value`
  `sty:par parname check default`
  `sty:par parname (+|-|*|/|+=) expr`

  A primeira sintaxe retorna o valor do parâmetro `parname`. Se o parâmetro não existe e o argumento `default` não é especificado, lança um erro. Se o parâmetro não existe e o argumento `default` é especificado, este valor é retornado.

  A segunda forma ajusta o valor do parâmetro para `value`. A terceira, verifica se o parâmetro existe; caso não exista cria o parâmetro com o valor `default`.

  Finalmente, a última forma sintática permite a realização de operações aritméticas simples: soma, subtração, multiplicação e divisão do valor do parâmetro pela expressão `expr`. Estas operações não afetam o valor do parâmetro. A última operação `+=`, incrementa o parâmetro de `expr`, afetando o seu valor deste ponto em diante.

- `sty:save filepath script`

  Se o arquivo especificado em `filepath` está desatualizado, tem tamanho igual a zero ou não existe, avalia o script `script` e salva a string resultante em `filepath`.

  Se `filepath` é um caminho relativo, ele é prefixado pelo caminho especificado pela variável do processador de estilos `outdir` seguida pelos atributo `lang` e `target` da folha de estilos. Se é um caminho absoluto, `filepath` é usado diretamente.

  O arquivo é considerado desatualizado se sua data é anterior à data especificada no primeiro atributo `_mtime` encontrado em elementos ascendentes do elemento para o qual está sendo avaliado o template. A busca parte do elemento corrente e prossegue em direção ao elemento-raiz do documento. O atributo `_mtime`

existe em elementos cujo conteúdo provém de arquivos referenciados pelo atributo `xref`; O atributo indica a data do arquivo. No caso do elemento-raiz, `_mtime` reflete a data do documento principal.

O segundo argumento é um script que só é avaliado no caso do arquivo necessitar ser salvo. É importante envolver o argumento em chaves `{...}` para evitar que o script seja avaliado quando o parser Tcl forma o comando. Veja os exemplos abaixo.

Exemplos:

- `[sty:save a/c/d/fun.c {[sty:apply . gencode]}]`

  Se o nó corrente é um nó `AdCFunction` que tem um atributo `xref` indicando o arquivo xml onde as informações sobre a função residem, compara as datas deste arquivo xml e do arquivo `outdir/lang/target/a/c/d/fun.c`. No caso do arquivo xml ser mais novo, executa o comando `sty:apply ...` e salva o texto resultante.

- `[sty:save /a/c/d/fun.c [sty:apply . gencode]]`

  A falta das chaves no segundo argumento faz com que o comando `sty:apply` seja executado mesmo que não haja necessidade de atualização do arquivo `/a/c/d/fun.c`.

- `sty:copy srcdir srcfile ?destfile?`

  Copia o arquivo `srcdir/srcfile` para `destfile`, se o arquivo origem é mais novo que o destino.

  Se `destfile` não é especificado, ele é formado pela variável `outdir` mais os atributos `lang` e `target`, mais o nome do arquivo origem, resultando em `outdir/lang/target/srcfile`. Caso seja um caminho relativo, utiliza o caminho `outdir/lang/target/destfile`. Finalmente, se `destfile` é um caminho absoluto, é usado diretamente.

4.3 Reutilização de Estilos

4.3.1 Inclusão de folhas de estilo

Uma folha de estilo pode ser reutilizada em outra folha de estilos. O elemento `<AdStylesheet ...>` pode aparecer no conteúdo de outro elemento do mesmo tipo para implementar a *inclusão* de estilos. Este elemento pode especificar o estilo incluído através do atributo `xref`, que identifica um arquivo que contém o estilo ou, então, incluí-lo diretamente.

O estilo principal e os estilos incluídos têm seus templates armazenados em diferentes *espaços de nomes* de forma a possibilitar a implementação de um processo de resolução de nomes que permita a especificação de qualquer um deles e a resolução de conflitos.

```
<AdStylesheet lang="a" target="x">
    <AdStylesheet lang="b" target="y">
        <AdTemplate name="adt1" section="qq">
            ...
        </AdTemplate>
        <AdTemplate name="adt5" section="qq">
            ...
        </AdTemplate>
    </AdStylesheet>
    <AdStylesheet lang="b" target="z" xref="mystyle.sty"/>
    <AdTemplate name="adt1" section="qq">
        ...
    </AdTemplate>
    <AdTemplate name="adt2" section="qq">
        ...
    </AdTemplate>
</AdStylesheet>
```

Quando uma folha de estilos é lida pelo processador, é criado um *dicionário de templates*. Cada template dá origem a duas chaves no dicionário. A primeira chave é formada pela identificação do template (atributos `name` e `section` de `AdTemplate`) mais a

identificação do estilo que contém o template (atributos `lang` e `target` do elemento `AdStylesheet` imediatamente acima do template). São chamadas de *chaves qualificadas*. É inserida, ainda, uma segunda chave, constituída sem considerar a identificação da folha de estilo do template, usando strings vazias para o valor de `lang` e `target` — *chaves simples* ou *não qualificadas*. Em ambos os casos, a ordem de leitura é a da folha de estilo, sendo que, no caso de templates com a mesma chave, prevalece o template lido por último.

A folha de estilos anterior origina o dicionário de templates apresentado graficamente na figura 4.3, supondo que o estilo contido no arquivo `mystyle.sty` contenha os templates `adt3` e `adt4`. Note que a entrada não qualificada para o template `adt1` aponta para o template do estilo principal, posto que foi lido depois do template de mesmo `name` do estilo `b.y`.

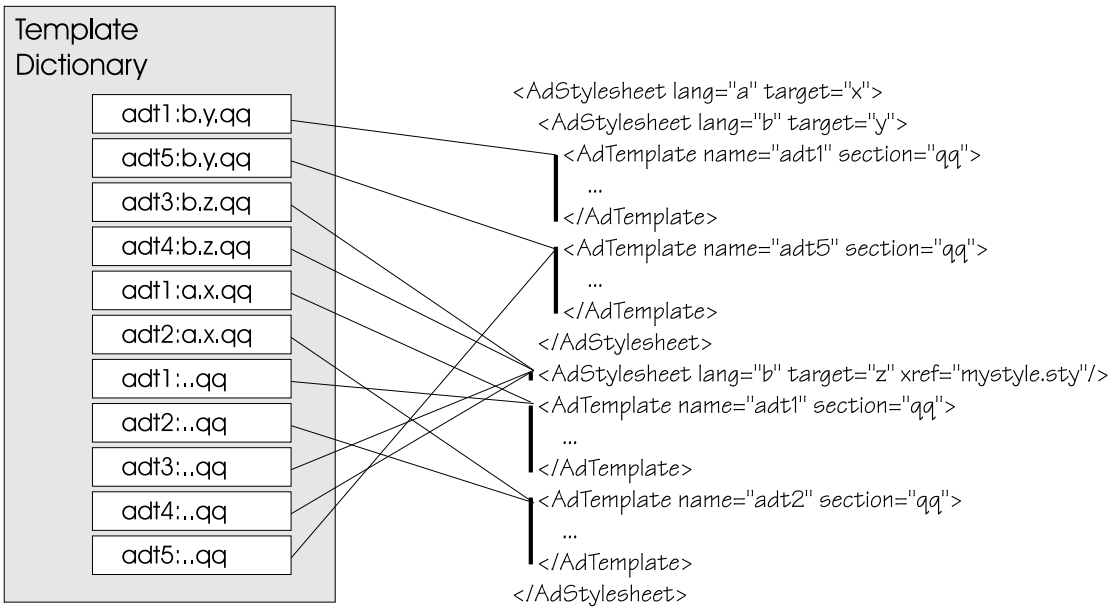

Figura 4.3: Dicionário de Templates

4.3.2 Resolução de nomes

No processo de busca por um template para aplicação, as seguintes chaves são consideradas, nesta ordem, e o primeiro *match* é aproveitado:

- chave qualificada, usando o par `lang/target` da *folha de estilos principal*, que é o estilo que contém todos os demais.

  Ex.: `name:lang.target.section`
  name/section são os argumentos do comando apply e target/lang identificam o estilo principal

- chave qualificada, usando o par `lang/target` do template que contém o comando `apply` que originou a busca.

  Ex.: `name:lang.target.section`
  name/section são os argumentos do comando apply e target/lang identificam o estilo que contém o comando apply

- chave simples.

  Ex.: `name:..section`
  note que target e lang são strings vazias

- chave formada considerando que `section` é uma especificação completa, do tipo `lang.target.section`.

  Ex.: `name:section`

A adoção de tal esquema traz algumas implicações significativas para a programação de estilos. Em primeiro lugar, permite que o estilo principal redefina apenas alguns templates de um estilo incluído, preservando a lógica geral de tal estilo. Contudo, um estilo incluido que não tenha templates redefinidos, opera como se fosse o principal (ou seja, seus templates têm prioridade na busca, quando acionados internamente). A especificação do argumento `section` de um comando `apply` com o prefixo `lang.target.` permite a ativação de um template específico de um estilo incluído.

4.3.3 Macros

As macros definidas em folhas de estilo são organizadas em um *dicionário de macros* e o processo de criação do dicionário e de resolução de nomes é similar ao dos templates, podendo ser utilizado o atributo `section` para a especificação de identificadores qualificados.

4.4 Comandos definidos pelo usuário

A instrução de processamento XML (*Processing Instruction*, PI) `adesso` permite ao autor da folha de estilo estender o conjunto de comandos do processador de estilos através de procedimentos escritos em *Tcl*. As definições contidas na instrução de processamento são avaliadas no espaço de nomes dos comandos pre-existentes, de forma que sua utilização no texto de templates é similar aos demais comandos. Estes procedimentos devem retornar uma string, que será inserida no texto de saída no lugar da chamada. Apresentamos um exemplo na sequência.

```
<AdStylesheet lang="mylang" target="mytarget">

    <AdTemplate name="AdToolbox">
        ...
        [my_proc [sty:value Short]]
        ...
    </AdTemplate>

    ...

    <?adesso
        proc my_proc {txt} {
            return "-- $txt --"
        }
    ?>'

</AdStylesheet>
```

4.5 Comparação com XSLT

Um processador XSLT opera transformando uma árvore XML em outra árvore XML. Diferente do processador de estilos do Adesso, é um processo de construção de árvores XML. Uma transformação XSLT gera um documento de saída que deve ser, necessariamente, pelo menos parte de um documento XML bem formado (fragmento válido de árvore). O processador de estilos do Adesso efetua substituições de texto, aproveitando o parser da linguagem Tcl. Os templates do Adesso, por outro lado, são constituídos apenas de texto plano, sem marcação XML. Assim sendo, pode gerar qualquer tipo de documento de saída.

A especificação dos nós aos quais se aplica um template, no XSLT, é feita através de expressões XPath; os templates podem ser aplicáveis a quaisquer tipos de nós da árvore de entrada. Na linguagem do Adesso, esta especificação é simplificada e os templates se aplicam apenas aos *elementos* (`ELEMENT_NODE`) do documento de entrada.

O processador de estilos do Adesso, além de seus comandos específicos, inspirados no XSLT, pode executar qualquer comando da linguagem Tcl. Pode, inclusive, usando instruções de processamento `adesso`, definir novos comandos Tcl. Esta facilidade para mesclar scripts Tcl com comandos similares ao XSLT, é a principal característica da linguagem, responsável pela sua flexibilidade.

Finalmente, o Adesso utiliza o atributo `section` de seus templates, equivalente ao atributo `mode` dos templates XSLT, para implementar a identificação de templates contidos em estilos incluídos pela folha de estilos principal.

4.6 Implementação

O processador de estilos pode ser utilizado a partir de um *shell*, da console do Adesso ou do Ambiente Integrado de Desenvolvimento; veja o capítulo 7.

A especificação dos comandos é a mesma para uso direto em um *shell* (ou janela do DOS) ou na console do Adesso.

```
% adesso
```

Usado sem argumentos, o comando `adesso` lança a console do Adesso. É preciso haver um sistema de janelas (MSWindows ou XWindows). A figura 4.4 mostra uma tela desta console. A partir da console, é possível a manipulação direta das árvores DOM, do documento de entrada ou da folha de estilos. Os comandos descritos a seguir podem ser executados tanto a partir da console quanto diretamente da linha de comandos.

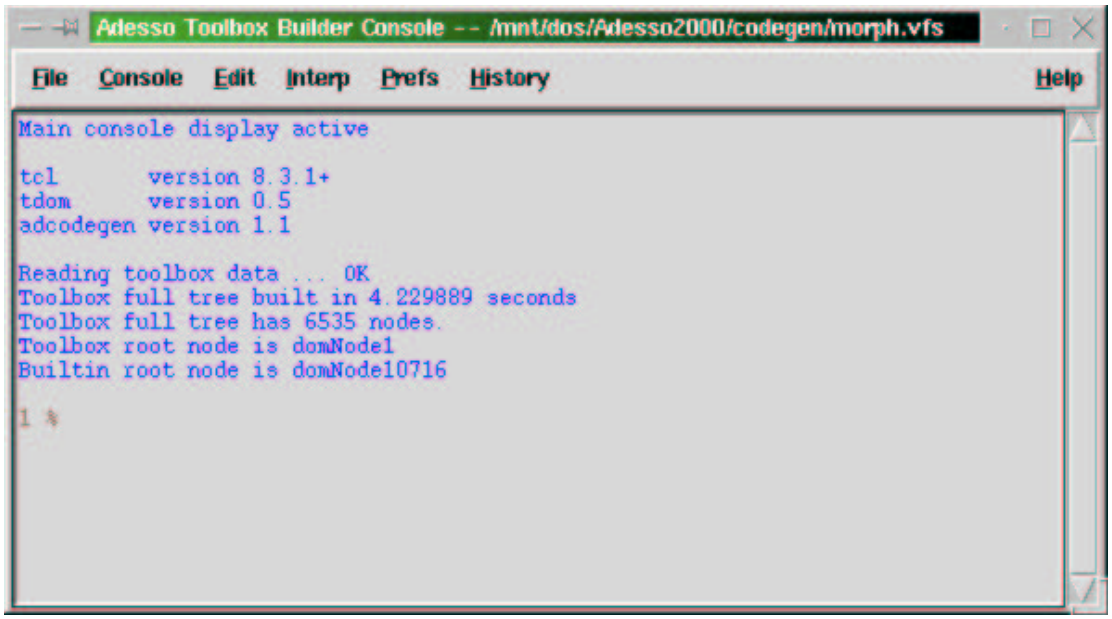

Figura 4.4: Console do Adesso

```
% adesso load -tbx toolboxname
              -sty stylesheetfile
```

Carrega a toolbox e/ou estilos especificados nas opções `-tbx` e `-sty`.

```
% adesso codegen -tbx     toolboxname
                 -in      inputdoc.xml
                 -sty     stylesheet.sty
```

```
-target  mytarget
-lang    mylang
-preproc preprocessorfile.tcl
-out     outputdir
-dbg     debuglevel
```

Este comando executa o processador de estilos do adesso. O resultado da avaliação do template de nível mais alto, caso não seja uma string vazia, é salvo em um arquivo denominado result.out, no diretório de saída.

- `-tbx toolboxname`
  Indica o nome de uma toolbox a ser usada como documento de entrada. Se esta opção é especificada, desconsidera a opção `-in`. Se tanto `-tbx` quanto `-in` são omitidos, usa a toolbox carregada previamente com o comando `adesso load -tbx ...`.

- `-in inputdoc.xml`
  Especifica o caminho do documento de entrada. Se este argumento não é especificado, o processador utiliza a toolbox indicada no argumento `-tbx` ou aquela que foi previamente carregada em sua memória pelo comando
  `adesso load -tbx toolbox`.

- `-sty stylesheet.sty`
  Especifica um arquivo que contém a folha de estilos a utilizar. Caso não exista o argumento, carrega os estilos pré-construídos (*builtin*).

- `-target mytarget`
  `-lang mylang`
  Identificam a folha de estilos a ser utilizada.

- `-out outputdir`
  Especifica o diretório que conterá os produtos da aplicação do estilo. Ajusta o valor da variável `outdir` do processador de estilos (veja a descrição dos comandos `sty:save` e `sty:copy`).

- `-preproc preprocessorfile.tcl`
  Em alguns casos é possível especificar como documento de entrada um arquivo não XML. Nestes casos é necessário especificar um script Tcl que tranforme o arquivo em uma árvore DOM para uso do processador. Esta opção especifica o arquivo que contém o script.

- `-dbg debuglevel`
  Indica, através de um inteiro, o nível de verbosidade desejado.

Capítulo 5

O Modelo de Informação

A fim de modelar as informações de uma *toolbox* de maneira neutra e facilmente acessível a programas escritos em diversas linguagens de programação, o Adesso utiliza uma representação das informações em XML. A utilização do XML facilita a manutenção dos dados e possibilita a utilização de uma grande (e crescente) variedade de ferramentas desenvolvidas por terceiros para edição, validação e transformação das informações.

Esta representação é formalizada através de um documento *XML Schema* que pode ser considerado a definição de uma base de dados orientada a objetos. Os principais elementos deste esquema são apresentados nas seções seguintes.

5.1 Modelagem dos Dados

A figura 5.1 reproduz o modelo de dados utilizado para a representação de uma toolbox. Para ilustrar o mapeamento deste modelo em uma estrutura de dados XML, apresentamos em seguida um esquema XML correspondente ao diagrama.

A representação dos relacionamentos utilizados para modelagem dos dados é feita da seguinte forma.

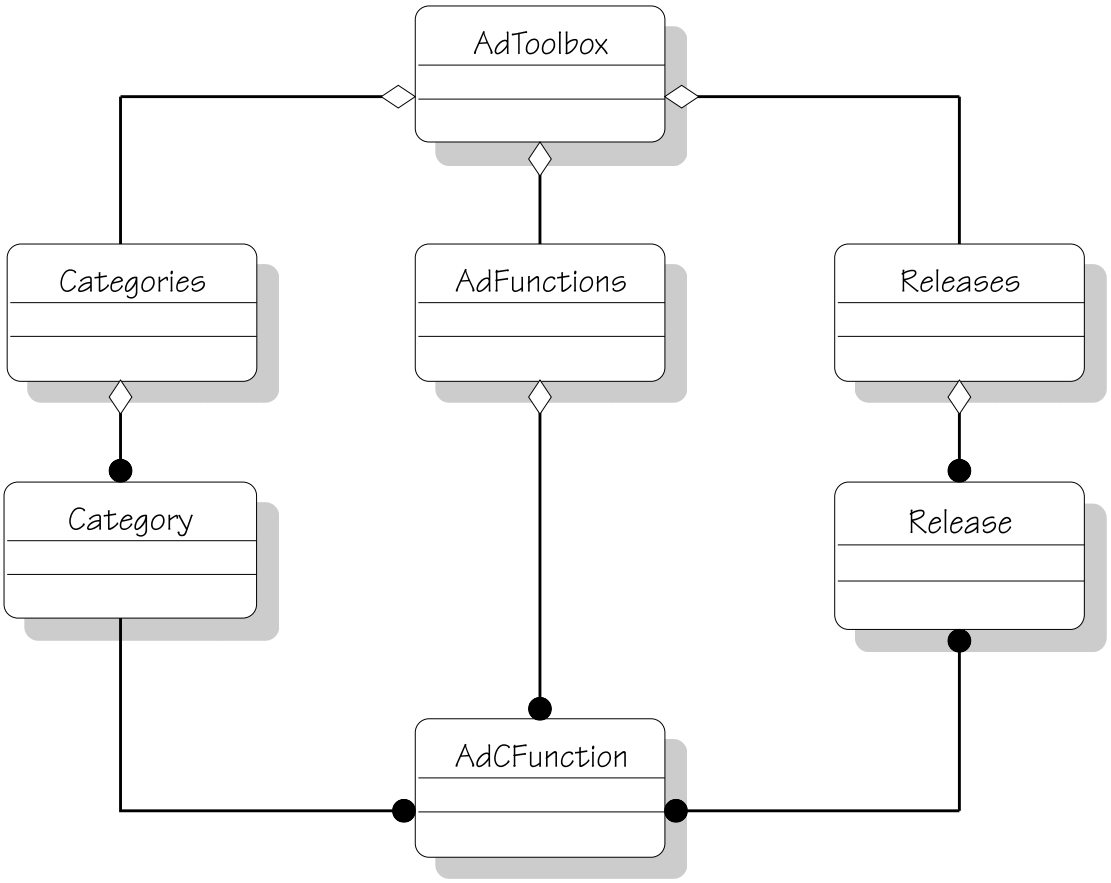

Figura 5.1: Estrutura lógica simplificada de uma toolbox

- *Composição*
  é representada naturalmente pela estrutura em árvore dos documentos XML. Por exemplo, um objeto do tipo `AdToolbox` contém um objeto `Categories` que, por sua vez, agrega múltiplos objetos do tipo `Category`. Esta relação pode ser observada na figura 5.1: o elemento `Category` é parte do elemento `Categories` que é um componente do elemento `AdToolbox`.

- *Referências*
  são modeladas através de atributos com tipo derivado dos tipos básicos do XML

denominados ID, IDREF e IDREFS. Assim o relacionamento que estabelece que uma categoria referencia um ou mais objetos do tipo AdCFunction, é modelado através do atributo AdCFunction.cat, de tipo derivado de IDREF, e do atributo Category.name, derivado de ID. Os tipos básicos não são usados diretamente a fim de possibilitarem uma redução no espaço de busca, pela utilização de tipos derivados.

```
.
<xsd:schema>
    <xsd:element name="AdToolbox" type="ToolboxType"/>
    <xsd:complexType name="ToolboxType">
        <xsd:sequence>
            <xsd:element name="Categories"><complexType>
                <xsd:element name="Category" type="CatType"
                                              maxOcurrs="unbounded"/>
            </xsd:complexType></element>
            <xsd:element name="Releases"><complexType>
                <xsd:element name="Release" type="RelType"
                                             maxOcurrs="unbounded"/>
            </xsd:complexType></element>
            <xsd:element name="AdFunctions"><complexType>
                <xsd:element name="AdCFunction" type="TbxCFunType"
                                                 maxOcurrs="unbounded"/>
            </xsd:complexType></element>
            ...
        </xsd:sequence>
        <xsd:attribute name="name"   type="string"/>
        <xsd:attribute name="title"  type="string"/>
        <xsd:attribute name="prefix" type="string"/>
        ...
    </xsd:complexType>


.
    <xsd:complexType name="CatType">
        <xsd:attribute name="name" type="CatID"/>
```

```
        ...
    </xsd:complexType>
    <xsd:complexType name="TbxCFunType">
        <xsd:attribute name="cat" type="CatRef"/>
        <xsd:attribute name="dir" type="DirRef"/>
        <xsd:attribute name="rel" type="RelRefs"/>
        ...
    </xsd:complexType>

    <xsd:simpleType name="CatID" base="ID">...</simpleType>
    <xsd:simpleType name="CatRef" base="IDREF">...</simpleType>
    <xsd:simpleType name="RelID" base="ID">...</simpleType>
    <xsd:simpleType name="RelRefs" base="IDREFS">...</simpleType>
    ...
</xsd:schema>
```

5.2 Estrutura de uma toolbox

A figura 5.2 ilustra o esquema XML de uma toolbox. Na figura, gerada por um programa de edição de XML, a forma de composição de um elemento é indicada por símbolos especiais para sequência (AdToolbox) ou escolha (AdFunctions). O símbolo + indica que existem elementos adicionais não representados.

O elemento raiz de um documento XML que representa uma toolbox é `AdToolbox`. Uma toolbox contém informações sobre o conjunto de componentes que a constitui. Estas informações podem ser classificadas em seis grupos:

- **Identificação da toolbox**
  Dados que especificam o nome da toolbox, prefixo para os nomes dos componentes, versão etc.

- **Documentação global da toolbox**
  Informações para a criação da documentação sobre o conjunto de componentes.

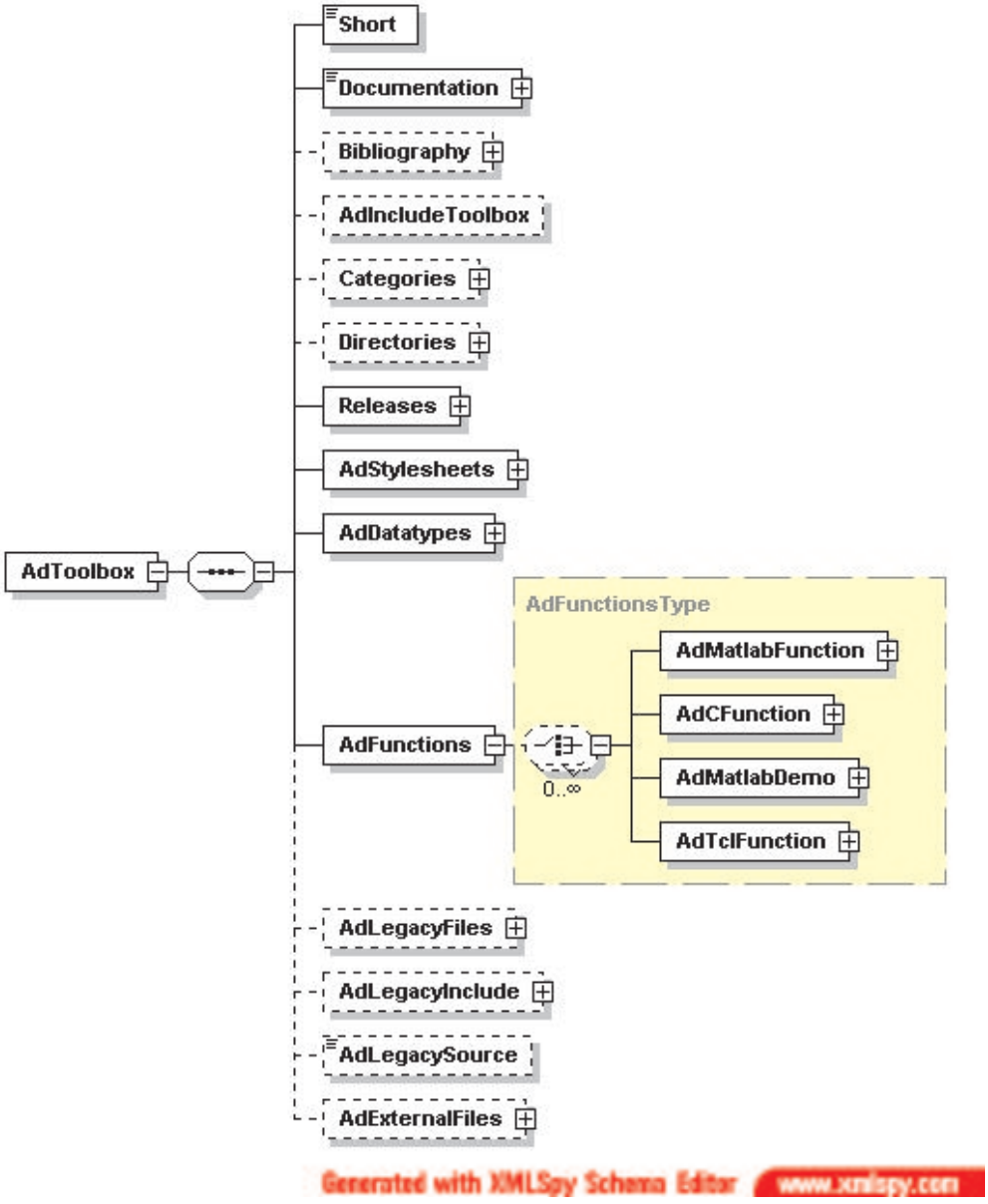

Figura 5.2: Composição de uma toolbox

- **Classificações dos componentes**
  Permitem criar grupos de componentes para fins de documentação, separação em diretórios e criação de *releases*.

- **Componentes da toolbox**
  Coleção de elementos que especificam cada componente da toolbox.

- **Dependências externas**
  Enumera dependências externas como arquivos de dados, bibliotecas, arquivos de inclusão etc.

- **Folhas de estilos específicas**
  Folhas de estilo e mapas de tipos de dados que implementam particularidades da toolbox, sobrecarregando as definições pre-existentes.

5.2.1 Identificação da Toolbox

- **Atributos**

  - **name**
    especifica o nome da toolbox que será usado para nomear os produtos gerados a partir dela.

  - **prefix**
    prefixo a ser usado na formação dos nomes dos comandos criados a partir dos componentes da toolbox.

  - **title**
    nome completo da toolbox.

  - **major, minor e patchlevel**
    a versão oficial da toolbox é dada por uma string com o formato `major.minor`. A versão completa é formada por `major.minor.patchlevel`. patchlevel é incrementado automaticamente a cada checkin.

  - **vdate**
    indica a data em que ocorreu a última mudança de versão oficial (incremento de minor).

  - **cvs**
    revisão, string `Revision` do CVS (controle de versões).
  - **cvsdate**
    data da revisão, string `Date` do CVS.

- **Short**
  Descrição resumida (em uma linha) da toolbox.

5.2.2 Documentação Global da Toolbox

- **Documentation**
  Documentação formatada. Este elemento permite a criação de documentos com marcação XML possibilitando sua transformação para diferentes formatos. A marcação inclui tags para a execução de scripts e utilização dos resultados permitindo a inclusão de demonstrações e exemplos consistentes. Este elemento agrupa os diversos elementos de documentação.

- **Bibliography**
  Lista de referências bibliográficas utilizadas na documentação da toolbox.

5.2.3 Classificações dos Componentes

- **Categories**
  A classificação dos componentes da toolbox em categorias é utilizada para a geração da estrutura da documentação: cada categoria pode corresponder a um capítulo do manual de usuário, por exemplo.

- **Directories**
  A classificação em diretórios é útil para organizar a distribuição dos arquivos em diretórios.

- **Releases**
  Este elemento é um classificador especialmente importante pois é utilizado para a definição da constituição das distribuições da toolbox. Cada grupo especifica os componentes que devem integrar a release e, ainda, um conjunto de símbolos que podem ser utilizados na compilação dos componentes, particularizando assim cada release da toolbox. Um exemplo típico da utilidade destes símbolos é a criação de releases de demonstração ou de uso estudantil com limitações de uso intencionalmente inseridas.

5.2.4 Componentes da Toolbox

- **AdDatatypes**
  Coleção de tipos de dados utilizados pela toolbox.

- **AdFunctions**
  São os componentes contidos na toolbox. Atualmente, os seguintes elementos são suportados:

  - **AdCFunction**
    Especifica informações para uma função escrita em linguagem *C*. A constituição destes elementos é apresentada na seção 5.3.
  - **AdMatlabFunction**
    Especifica informações para uma função escrita em linguagem *MATLAB*.
  - **AdTclFunction**
    Especifica informações para uma função escrita em linguagem *TCL*.

5.2.5 Dependências Externas

- **AdIncludeToolbox**
  Se presente, especifica toolboxes das quais a toolbox depende.

- **AdLegacyFiles**
  Arquivos necessários para a formação de uma distribuição completa da toolbox. Os arquivos listados aqui incluem arquivos fonte de programas C, arquivos de inclusão, arquivos de dados (imagens, configuração) e arquivos binários para diferentes sistemas operacionais.

- **AdLegacyInclude**
  Trecho de texto a ser inserido no arquivo de inclusão da toolbox.

- **AdLegacySource**
  Código **C** a ser incluído na biblioteca da toolbox.

- **AdExternalFiles**
  Arquivos externos, normalmente bibliotecas dinâmicas, dos quais a toolbox depende.

5.2.6 Folhas de Estilo Específicas

- **AdStylesheets**
  Permite a particularização dos geradores de código para esta toolbox, além de conter mapeamentos de tipos de dados específicos.

5.3 Estrutura de uma função C

A figura 5.3 ilustra o esquema XML das informações relativas a funções C. O elemento que define uma função C é **AdCFunction**. As informações relativas a funções escritas em C podem ser classificadas em cinco grupos:

- **Atributos e identificação**
  Permitem classificar e identificar o componente no contexto da toolbox que o contém.

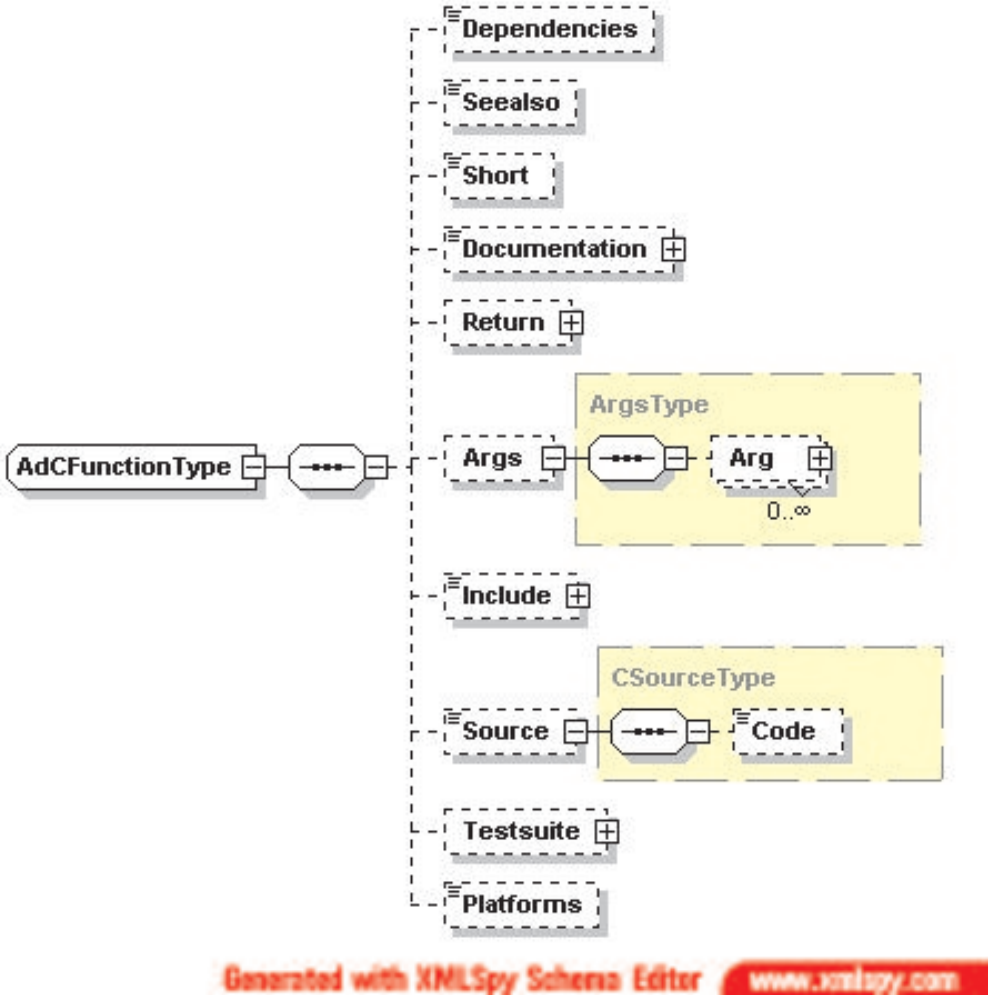

Figura 5.3: Composição de uma função C

- **Documentação**
  Documentam a função e identificam inter-relacionamentos no âmbito da toolbox.

- **Definição do protótipo**
  Definem e documentam valores de retorno e argumentos da função C.

- **Implementação**
  Contêm o código fonte da função ou indicam sua localização.

- **Dependências**
  Especificam dependências entre componentes da toolbox em nível de código.

5.3.1 Atributos e Identificação

- **Atributos**
  - **name**
    Nome pelo qual a função é identificada. É de tipo derivado do tipo primitivo ID, o que implica em que o nome deve ser único em toda a toolbox.
  - **exposed**
    Indica se uma interface para o componente deve ser construída. Uma função pode ter uso apenas interno e não ser exposta para a linguagem de scripting.
  - **documented**
    Indica se deve ser gerada documentação para o componente.
  - **xref**
    Se este atributo existe, indica o nome de um arquivo XML que contém os dados da função.
  - **dir**
    Indica o diretório ao qual pertence o componente. É uma referência a um classificador Directory.
  - **cat**
    Indica a categoria à qual pertence o componente. É uma referência a um classificador Category.
  - **rel**
    Indica uma lista de releases que contêm o componente. É uma lista de referências a classificadores Release.
  - **state**
    Estado da implementação do componente (alpha, beta ou final).

- **Short**
  Descrição resumida da função.

5.3.2 Documentação

- **Documentation**
  Este elemento agrupa os diversos elementos de documentação.

- **Seealso**
  Possibilita a geração de referências cruzadas entre os componentes da toolbox.

5.3.3 Definição do Protótipo

- **Return**
  Em conjunto com o elemento Args, define o protótipo da função. Especifica informações sobre o valor de retorno da função:

  - **name**
    Nome utilizado para identificar o valor de retorno.
  - **type**
    Tipo do valor de retorno. Deve ser uma referência válida a um tipo de dado C (AdCDatatype).
  - **output**
    Indica se o valor de retorno deve ser passado à linguagem de scripting (se ele constitui uma saída da função).
  - **constraint**
    Indica as restrições aplicáveis. As restrições válidas dependem do tipo de dado.

- **Args**
  Informações sobre cada argumento da função. É formado por uma coleção de elementos `Arg`. Cada elemento `Arg` tem os seguintes atributos:

  - **name**
    Nome do argumento.
  - **type**
    Tipo do argumento. Deve ser uma referência válida a um tipo de dado C (AdCDatatype).
  - **dir**
    Indica a direção na qual flui o dado transportado pelo argumento.
  - **constraint**
    Indica as restrições aplicáveis. As restrições válidas dependem do tipo de dado.
  - **optional**
    Especifica se o argumento é opcional, do ponto de vista da interface de scripting.
  - **default**
    Especifica um valor default para o argumento. Útil apenas se o argumento é opcional ou ignorado.
  - **map**
    Este valor é utilizado na localização de typemaps; permite a particularização do gerador de código para casos especiais.

5.3.4 Implementação

- **Source**
  Este elemento especifica o código C que implementa a função. O código pode estar contido em um arquivo externo ou aparecer como o conteúdo do elemento.

5.3.5 Dependências

- **Dependencies**
  Caso este componente dependa de outros componentes da toolbox, estes devem ser listados aqui.

- **Platforms**
  Plataformas (sistemas operacionais) para as quais o componente deve ser construído.

- **ExternalFiles**
  Lista de arquivos externos requeridos pelo componente.

Capítulo 6

Geradores de Código

Nesta seção apresentamos rapidamente as folhas de estilos existentes no sistema para a geração de código e documentação para algumas plataformas de scripting.

Para utilização de uma determinada função escrita em linguagem **C** através de uma plataforma de scripting, é necessário uma função de interface que efetue o mapeamento dos dados envolvidos na chamada, de forma a compatibilizá-los com os tipos existentes na linguagem alvo. Estas funções de interface, também denominadas *wrappers*, são geradas automaticamente pelo Adesso a partir das informações sobre a *assinatura* da função (especificação dos tipos dos argumentos e valor de retorno). A principal tarefa de um gerador de código de interface é o mapeamento dos tipos de dados da linguagem **C** em tipos de dados suportados pela plataforma de scripting.

Outro aspecto fundamental endereçado pelas transformações do Adesso é a documentação da *toolbox* para determinada plataforma. O Adesso conta, hoje, com um sistema de geração de documentação bastante sofisticado, especialmente no caso do *MATLAB*. Os estilos para criação de documentos suportam, como formatos de saída, o HTML, Postscript, PDF, *M-files*.

6.1 Biblioteca Base

As funções **C** contidas em uma *toolbox* dão origem a uma biblioteca dinâmica denominada **Biblioteca Base** que é, então utilizada pelas interfaces geradas automaticamente para as plataformas de scripting suportadas. As seguintes folhas de estilo são responsáveis pela geração dos fontes dessa biblioteca.

- **Código C**
  A partir dos fragmentos de código fonte contido nos elementos `AdToolbox/AdFunctions/AdCFunction`, este estilo gera os arquivos fonte da biblioteca base, além dos arquivos de inclusão necessários para sua utilização.

- **Makefiles**
  São scripts *Tcl* usados para controlar o processo de construção da biblioteca, denominados *Brasfiles* porque utilizam uma extensão do *Tcl*, **bras**. Veja a seção 6.5 para informaçãoes sobre o sistema de construção utilizado pelo Adesso.

6.2 Biblioteca para o MATLAB

O conjunto de geradores de código para o *MATLAB* é o mais completo atualmente. A maior parte dos estilos foi criada para suportar o desenvolvimento da toolbox de morfologia matemática e estendida posteriormente para maior generalidade.

- **Código C das interfaces**
  São gerados com base nas infomações sobre o protótipo da funções **C** da *toolbox*, contidas nos elementos `AdToolbox/AdFunctions/AdCFunction`.

- **Código M**
  Uma *toolbox* pode conter funções escritas na linguagem de scripting do *MATLAB*, *M-functions*. Estes componentes são mantidos em elementos `AdToolbox/AdFunctions/AdMatlabFunction`.

- **Código para Demonstrações (Demos)**
  Um componente especial de uma *toolbox* é constituído por um script *MATLAB* escrito para demonstrar certa característica. Estes componentes estão contidos nos elementos `AdToolbox/AdFunctions/AdMatlabDemo`.

- **Documentação HTML**
  Para um exemplo de documentação HTML gerada pelo Adesso, visite a página `'http://www.mmorph.com/html/index.html'`, que apresenta a documentação da toolbox de morfologia matemática criada com o auxílio do Adesso.

- **Documentação Postscript/PDF**
  Constitui a documentação impressa da toolbox.

- **Makefiles**

6.3 Biblioteca para o Tcl/Tk

O *Tcl/Tk* tem sido bastante usado para a criação de aplicações de processamento de imagens. O exemplo proeminente é o **Prontovideo** — *Image Sequence Segmentation Tool*, [20][21], uma ferramenta para edição de objetos em sequências de vídeo.

- **Código C das interfaces**
  Wrappers para utilização via *Tcl/Tk*

- **Makefiles**

6.4 Biblioteca para o MATLAB Compiler

Estes estilos criam uma biblioteca usada em conjunto com o *MATLAB Compiler* para a criação de aplicações executáveis, que não necessitam do interpretador *MATLAB*, a partir de scripts (*M-files*).

- **Código C das interfaces**
- **Makefiles**

6.5 O Sistema de Construção de Toolboxes

O sistema de construção (interpretação de makefiles) utilizado pelo Adesso está baseado em uma extensão do Tcl denominada Bras, [22]. Versões anteriores do Adesso utilizavam um sistema baseado no autoconf/configure/make, paradigma comum no mundo Unix (usado inclusive para a construção do Tcl/Tk).

A utilização de um sistema de construção baseado no TCL visa facilitar a implantação deste sistema em diferentes sistemas operacionais. Em relação ao sistema baseado no autoconf, a construção via TCL apresenta um ganho significativo em termos de facilidade de instalação, de criação dos makefiles e de integração com o restante do Adesso. Versões executáveis do sistema de construção, na forma de um único arquivo (admake.exe, por exemplo), foram geradas para as plataformas suportadas pelo Adesso (Windows, Linux, Solaris).

Capítulo 7

Ferramentas de Autoria

O Adesso é dotado de uma interface gráfica para controle de suas funcionalidades denominada Ambiente Integrado de Desenvolvimento (IDE). Este ambiente proporciona cesso a:

- criação e edição de componentes;
- controle do processo de geração de código;
- manutenção e atualização do Adesso.

A IDE do Adesso possui um editor genérico de documentos XML, figura 7.1, com alguma especialização para facilitar a edição de toolboxes. Além do editor, o Adesso conta, hoje, com uma console de operação, que possibilita acesso a suas ferramentas de geração de código, a manipulação da árvore XML da toolbox e a várias ferramentas de manutenção do sistema.

Como a base de dados do sistema é mantida no formato XML, existem várias aplicações que podem ser utilizadas para a edição e validação dos dados (veja [14], por exemplo). Como estas tecnologias são novas, é razoável esperar que a diversidade de ferramentas cresça ainda mais, propiciando ao usuário do Adesso um grande repertório de editores.

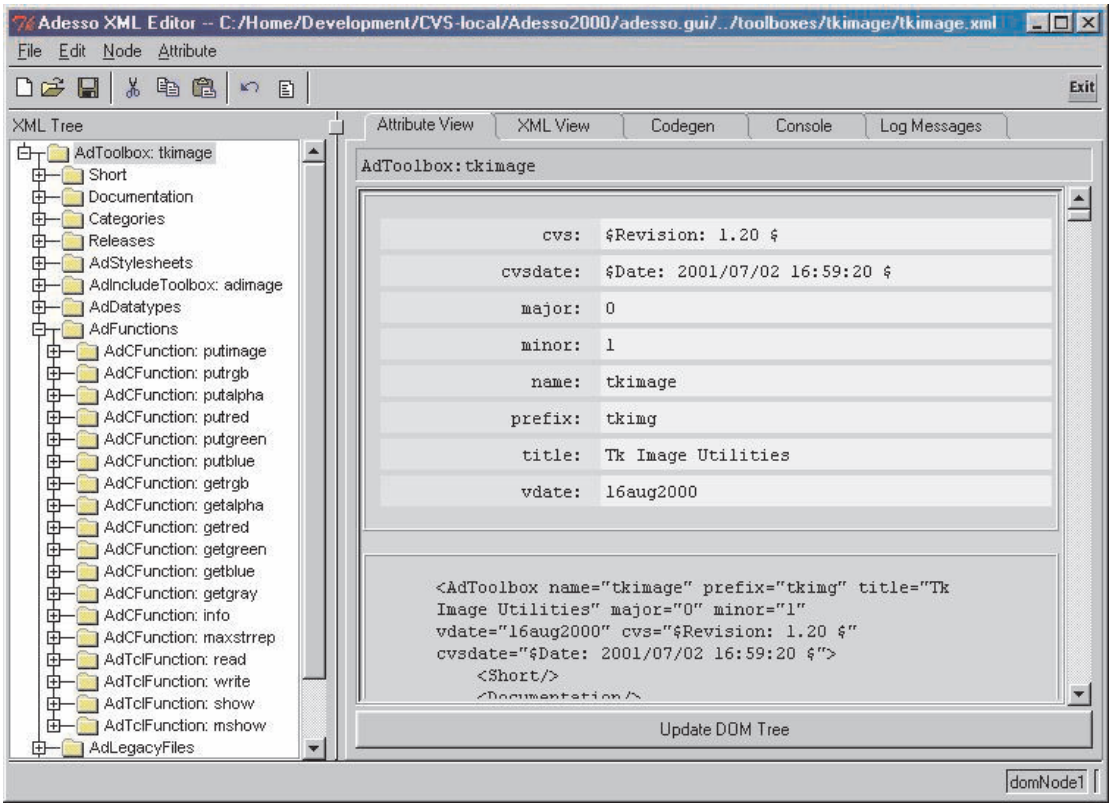

Figura 7.1: Tela do Ambiente Integrado de Desenvolvimento do Adesso

Muitas vezes, contudo, a utilização de ferramentas específicas é mais produtiva. Como a estruturação dos dados do Adesso é completamente especificada pelo seu esquema XML, é possível a criação de um editor especializado para a representação e edição dos dados do Adesso.

Este editor é construído a partir do conhecimento dos tipos de dados e das estruturas presentes em uma toolbox, especificadas no esquema XML associado. Por exemplo, se um item de dado é do tipo enumerado podemos associar a ele uma representação gráfica na forma de um artefato combobox. A especialização de um editor gráfico genérico de XML para um esquema específico pode ser feita através das próprias ferramentas de transformação do Adesso.

A criação de editores especializados para o Adesso é objeto de um projeto de pesquisa em andamento, *Interface Configurável para Desenvolvimento de Aplicações em Processamento de Imagens* [15]. A idéia deste projeto é a implementação do Ambiente Integrado do Adesso utilizando a linguagem Java e as ferramentas de transformação do Adesso. Este projeto deve levar à implementação de um *Ambiente Integrado de Desenvolvimento de Toolboxes*, semelhante às IDE's, *Integrated Development Environments*, comuns em plataformas Windows (O *C++ Visual Studio*, da Microsoft, é um bom exemplo). A utilização do esquema para ajustar os diversos elementos da interface de usuário deve minimizar o impacto causado por alterações na estrutura das informações da toolbox, facilitando ou mesmo automatizando o processo de adaptação.

Capítulo 8

Exemplos de Uso

Neste capítulo, apresentamos alguns exemplos de confecção de estilos para efetuar transformações em uma *toolbox*. Para a criação de uma distribuição de um produto derivado de uma *toolbox*, os estilos podem ser classificados em dois grandes grupos: (a) estilos para documentação e (b) estilos para geração de código.

8.1 Estilos de Documentação

Toda a informação necessária para a criação da documentação de uma toolbox está mantida em seus arquivos XML. A estrutura destes arquivos porém é mais próxima de uma base de dados que de um documento propriamente dito. Por outro lado, documentação de uma toolbox pode assumir diferentes formatos de apresentação. A estrutura dos documentos é, contudo, basicamente a mesma. Este raciocínio leva a considerações sobre a utilização de um formato intermediário de documentação que facilite a subsequente criação de documentos em diferentes formatos. Este formato intermediário, ainda baseado em marcação XML, tem a estrutura desejada para os documentos (seções, listas) além de tags associados à apresentação da informação (ênfase, verbatim). A utilização de formatos padronizados como o *DocBook* [16] nesta etapa traz vantagens interessantes como, por exemplo, a disponibilidade de estilos para a segunda etapa do processo de

transformação, utilizada para a geração dos documentos em formatos específicos como HTML, PDF, LaTeX etc. A figura 8.1 ilustra esta abordagem.

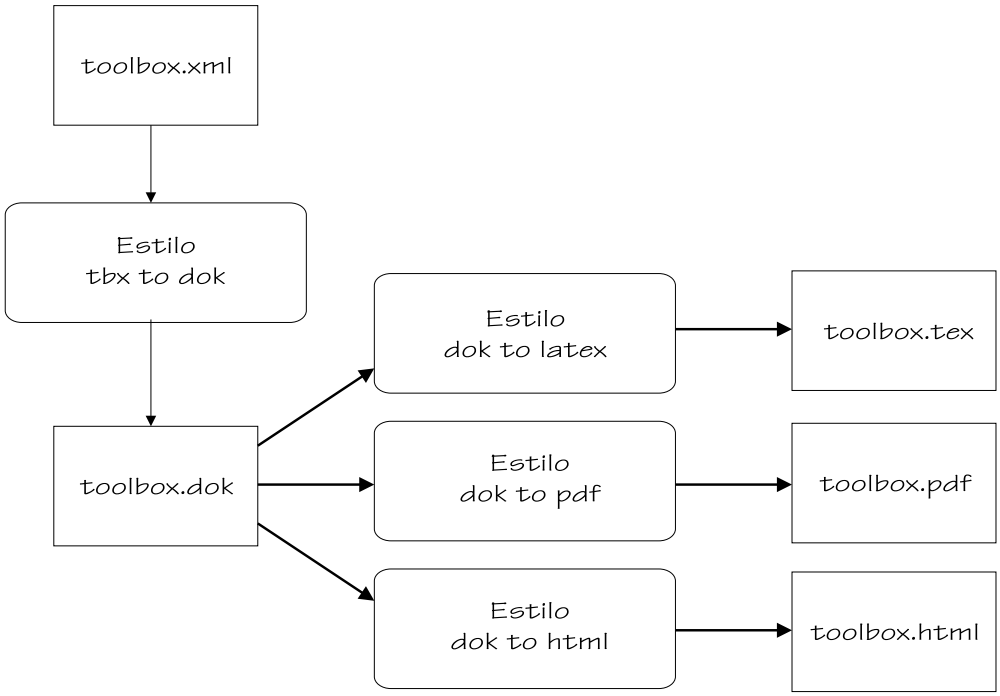

Figura 8.1: Estilos para Documentação

Apresentamos, em seguida, um exemplo de folha de estilo para a geração de um formato intermediário de documentação para uma toolbox. Utilizaremos uma folha de estilos simples que cria um resumo, ou índice, do conteúdo de uma toolbox, com as funções organizadas em categorias. Utilizaremos como formato intermediário um pequeno subconjunto do DTD *DocBook*.

8.1.1 Geração do documento intermediário (DocBook)

Apresentamos a seguir uma descrição detalhada da folha de estilo usada para gerar um documento *DocBook* a partir de uma toolbox.

1. O estilo será denominado `tbx2dok.index`.

```
<AdStylesheet target="index" lang="tbx2dok">
```

2. O template de topo salva o texto gerado em um arquivo com o mesmo nome do arquivo de entrada e extensão `dok`, no diretório especificado pelo parâmetro `outdir`. O processo de tranformação é iniciado com o template `AdToolbox` com seção `doc`.

```
<AdTemplate name="AdToolbox">
    [sty:par outdir check .]
    [sty:save [file join [sty:par outdir] [file rootname \
                         [file tail [sty:var infile]]].dok] \
                         {[sty:apply . doc]}]
</AdTemplate>
```

3. O elemento raiz do documento de saída tem o tag `article`. O título do documento é especificado pelos elementos `article/title` e `article/subtitle`. Os valores destes elementos são extraídos dos atributos `title`, `name`, `major`, `minor` e `vdate` do elemento `AdToolbox` do documento de entrada.

   O primeiro parágrafo, elemento `para`, do documento de saída é formado pelo conteúdo do elemento `Documentation/Descr` da toolbox. Um segundo parágrafo é formado por texto inserido pela folha de estilos. A geração do restante do documento é função do template `Categories/Category`, section `doc`.

   A marcação `<![CDATA[ ... ]]>` é utilizada para possibilitar o uso dos caracteres de marcação XML (`<` e `>`) no texto do template sem que eles sejam interpretados como tags.

```
<AdTemplate name="AdToolbox" section="doc"><![CDATA[
    <article>
        <title>
            [sty:value @title]
        </title>
        <subtitle>
            [sty:value @name]\
            V[sty:value @major].[sty:value @minor]\
            [sty:value @vdate]
```

```
        </subtitle>
        <para>
            [sty:value Documentation/Descr]
        </para>
        <para>
            This document presents the C functions of the
            <emphasis>[sty:value @title]</emphasis>\
            organized by category.
        </para>
        [sty:apply Categories/Category doc]
    </article>
]]></AdTemplate>
```

4. O template associado aos elementos de tag `Category` cria novas seções do documento de saída, uma para cada *categoria* especificada na toolbox. O `sty:if` inicial garante que o documento não contenha seções correspondentes a categorias vazias, sem componente algum. O título da seção é derivado do conteúdo do elemento `Category/Title`. Cada seção é formada por ítens de uma lista de definição, correspondentes às funções pertencentes à categoria. O texto de cada item é responsabilidade do template `AdCFunction.doc`.

   Note a expressão *XPath* utilizada para selecionar apenas funções da categoria em foco.

```
<AdTemplate name="Category" section="doc"><![CDATA[
    [sty:if [sty:xpath {//AdCFunction[@cat = "%%"]} \
                        [sty:value @name]] {
        <section>
        <title>[sty:value Title]</title>
        <variablelist>
        <title>C Functions</title>
        [sty:apply [sty:xpath {//AdCFunction[@cat = "%%"]} \
                              [sty:value @name]] doc]
        </variablelist>
        </section>
    }]
]]></AdTemplate>
```

5. Cada função da toolbox é documentada através de um item da lista. O nome da função forma o título do item. O nome da função é obtido com a aplicação de um procedimento Tcl que elimina possíveis *underscores* da string extraída do atributo `name`. Este procedimento é definido em uma instrução de processamento contida na folha de estilo. O item contém, ainda, uma descrição resumida, em uma linha, extraída do elemento `Short`.

```
<AdTemplate name="AdCFunction" section="doc"><![CDATA[
    <varlistentry>
        <term>
            [remove_underscores [sty:value @name]]
        </term>
        <listitem>
        <para>[sty:value Short]</para>
        </listitem>
    </varlistentry>
]]></AdTemplate>
<?adesso
    proc remove_underscores {text} {
        regsub -all "_" $text "" text
        return $text
    }
?>
</AdStylesheet>
```

Para aplicar esta folha de estilo à toolbox de morfologia matemática, executamos os comandos seguintes a partir da console do Adesso, supondo que o estilo esteja no arquivo `$vfsRoot/stylesheets/doc/tbx2dok.sty`:

```
% adesso load -tbx morph
% adesso codegen -target index -lang tbx2dok \
                -sty $vfsRoot/stylesheets/doc/tbx2dok.sty
```

O último comando produz o seguinte documento de saída, arquivo `morph.dok`:

```
<article>
    <title>SDC Morphology Toolbox</title>
```

```
    <subtitle>morph V1.00 07Ago01</subtitle>
    <para>
        The SDC Morphology Toolbox for MATLAB is a powerful
        collection of latest state-of-the-art gray-scale
        morphological tools that can be applied to image
        segmentation, non-linear filtering, pattern recognition
        and image analysis.
    </para>
    <para>
        This document presents the C functions of the
        <emphasis>SDC Morphology Toolbox</emphasis> organized
        by category.
    </para>
    <section>
        <title>Test</title>
        <variablelist>
            <title>C Functions</title>
            <varlistentry>
                <term>mmcpinterval</term>
                <listitem>
                    <para>
                      Copy the interval INTX in the interval INTY.
                    </para>
                </listitem>
            </varlistentry>
            <varlistentry>
                ...
            </varlistentry>
        </variablelist>
    </section>
    <section>
        ...
    </section>
</article>
```

8.1.2 Geração do documento final

Para exemplificar a segunda parte da transformação, apresentamos uma folha de estilo que produz um documento HTML a partir do documento *DocBook* gerado anteriormente.

1. O estilo será denominado `dok.html`.

```
<AdStylesheet target="html" lang="dok">
```

2. O template de topo salva o texto gerado em um arquivo com o mesmo nome do arquivo de entrada e extensão `html`, no diretório especificado pelo parâmetro `outdir`. O processo de tranformação é iniciado com o template `article` com seção `doit`.

```
<AdTemplate name="article">
    [sty:par outdir check .]
    [sty:save [file join [sty:par outdir] [file rootname [file tail\
        [sty:var infile]]].html] {[sty:apply . doit -trim "\n\t"]}]
</AdTemplate>
```

3. A partir deste ponto, o fluxo do processamento é conduzido pela estrutura do documento de entrada, o que é evidenciado pelo uso intensivo de comandos do tipo `[sty:apply *]`. Note a diferença para o estilo anterior, onde a estrutura do documento de saída era definida quase que exclusivamente pela folha de estilo.

   Depois de gerar o prólogo para o documento HTM, o template aciona os templates associados aos filhos do nó `article`. O template `article/title` gera o título do documento. Como elementos de tag `title` podem ocorrer em diversos pontos do documento de entrada, é necessário especificar o template de forma a captar apenas os nós `title` filhos do nó `article`.

```
<AdTemplate name="article" section="doit"><![CDATA[
    <html><head>
    <title>[sty:value .]</title>
    </head><body bgcolor=#FFFF99>
```

```
        [sty:apply * "" lev=3]
    </body></html>
]]></AdTemplate>

<AdTemplate name="article/title">\<![CDATA[
    <center>
        <h1>[sty:value .]</h1>
        [sty:if {../subtitle} {
            <h2>[sty:value ../subtitle]</h2>
        }]
    </center>
]]></AdTemplate>
```

4. Os elementos `section` incrementam o parâmetro `lev` de forma a refletir, nos títulos de cada seção, tratados pelo template `section/title`, o nível de aninhamento das seções.

```
<AdTemplate name="section">
    [sty:apply * "" lev=[sty:par lev + 1]]
</AdTemplate>

<AdTemplate name="section/title">
    [sty:tag h[sty:par lev - 1]]\
        [sty:value .]\
    [sty:tag /h[sty:par lev - 1]]
</AdTemplate>
```

5. Os parágrafos do documento são gerados pelo template seguinte. Note que o processamento continua com os filhos de `para`, a fim de formatar a marcação intra-parágrafo.

```
<AdTemplate name="para"><![CDATA[
    <p>[sty:apply * "" lev=[sty:par lev + 1]]</p>
]]></AdTemplate>
```

6. Os próximos templates formatam uma lista de definição. Utilizam a marcação HTML `<dl>`, `<dt>` e `<dd>` para formatar o elemento `<variablelist>` do *DocBook*.

```
<AdTemplate name="variablelist"><![CDATA[
    <dl><dl>
        [sty:apply * "" lev=[sty:par lev + 1]]
    </dl></dl>
]]></AdTemplate>

<AdTemplate name="varlistentry"><![CDATA[
    <dt><b>[sty:value term]</b><dt>
    <dd>[sty:apply * "" lev=[sty:par lev + 1]]</dd>
]]></AdTemplate>

<AdTemplate name="listitem">
    [sty:apply * "" lev=[sty:par lev + 1]]
</AdTemplate>
```

7. Finalmente, o elemento *DocBook* `<emphasis>` é mapeado no elemento HTML `<i>`.

```
<AdTemplate name="emphasis"><![CDATA[
    <i>[sty:apply * "" lev=[sty:par lev + 1]]</i>
]]></AdTemplate>

</AdStylesheet>
```

Para efetuar a transformação executamos o seguinte comando a partir da console do Adesso:

```
% adesso codegen -target html -lang dok \
                 -sty $vfsRoot/stylesheets/stx/dok2html.sty \
                 -in $srcRoot/$Toolbox/tbx2dok/index/$Toolbox.dok
```

A figura 8.2 mostra o documento obtido, `morph.html`, visualizado através de um navegador HTML.

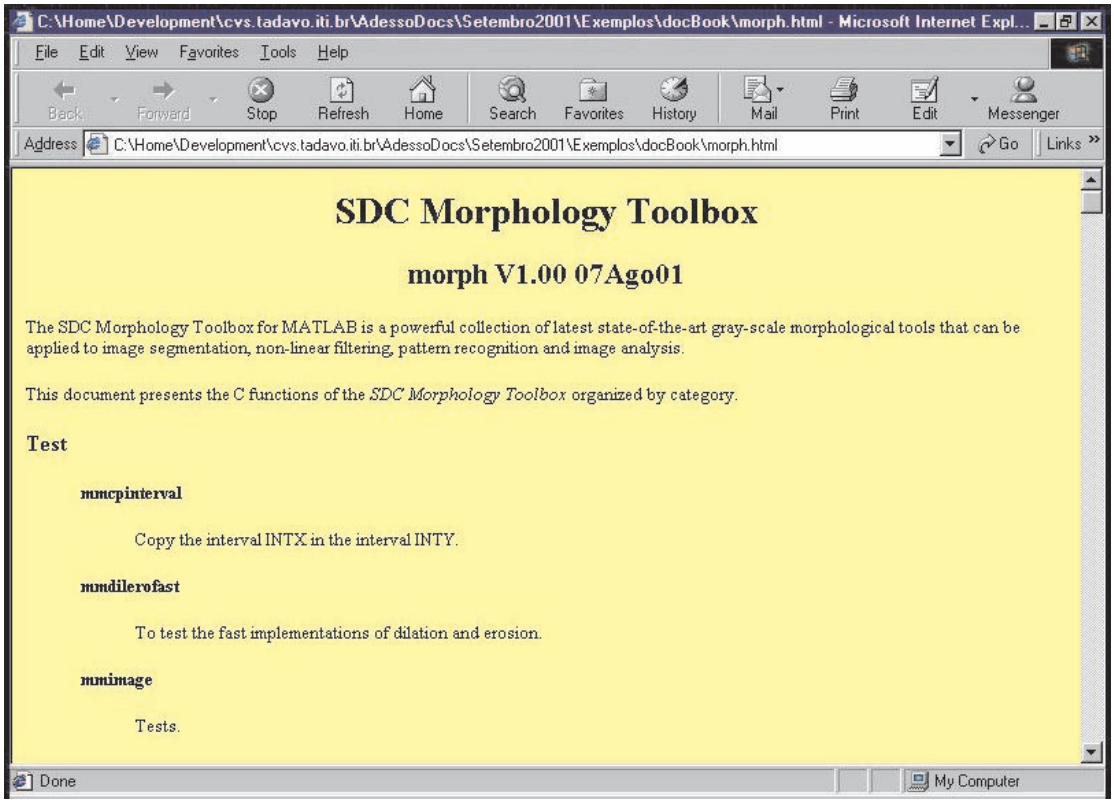

Figura 8.2: Tela do documento gerado

8.2 Geração de Código

O principal objetivo do Adesso é auxiliar seu usuário no processo de criação de extensões escritas em linguagem C para diversas plataformas de scripting. Nesta seção apresentamos exemplos de folhas de estilo para geração de código.

O processo de criação dos arquivos fonte de uma toolbox para determinada linguagem de scripting envolve duas etapas:

- Criação da biblioteca base
- Criação das interfaces para a linguagem de scripting

Utilizaremos a seguinte *toolbox*, bastante simplificada, como documento de entrada dos estilos que desenvolveremos na sequência.

```
<!-- zeta.xml -->
<?xml version="1.0" encoding="ISO-8859-1"?>
<AdToolbox name="zeta" prefix="zz" title="Toolbox Zeta" major="0" minor="1">
    <Short/>
    <Documentation/>
    <Bibliography/>
    <AdIncludeToolbox name="adimage" rel="all"/>
    <AdDatatypes/>
    <AdFunctions>
        <AdCFunction name="clone" exposed="yes" documented="yes">
            <Dependencies/>
            <Seealso/>
            <Short>Create a copy of an image, but without
                   copying the pixel contents</Short>
            <Documentation/>
            <Return name="img_out" type="adimage" output="yes"/>
            <Args>
                <Arg name="img" type="adimage" dir="in"/>
            </Args>
            <Include/>
            <Source>
                <Code><![CDATA[
                    int w, h, d, t;
                    adimage *img_out;
                    if (img == NULL) return NULL;
                    w = img->row_size;
                    h = img->col_size;
                    d = img->num_data_bands;
                    t = adimage_type(img);
                    img_out = adimage_create(w, h, d, t);
                    return img_out;
                ]]></Code>
            </Source>
            <Testsuite><Code/></Testsuite>
```

```
            <Platforms>windows linux sunos</Platforms>
        </AdCFunction>
    </AdFunctions>
    <AdLegacyFiles><Exports/></AdLegacyFiles>
    <AdLegacyInclude/>
    <AdLegacySource/>
    <AdExternalFiles/>
</AdToolbox>
```

8.2.1 Biblioteca Base

A criação dos arquivos fonte da biblioteca a partir dos arquivos XML da toolbox é trivial, já que as funções são definidas através de seus protótipos e seu corpo de código. O seguinte estilo gera os fontes para a biblioteca.

1. O estilo cria um arquivo fonte para cada função da toolbox. O procedimento `sty:cb` indenta o texto do arquivo C de acordo com algumas regras simples.

```
<AdStylesheet target="library" lang="tbx2c">

<AdTemplate name="AdToolbox">
    [sty:par outdir check .]
    [sty:foreach //AdCFunction {
        [sty:save [file join [sty:par outdir] \
                          [sty:value @dir] \
                          [sty:value @name]_lib.c] \
                         {[sty:cb [sty:apply . lib]]}]
    }]
</AdTemplate>
```

2. O template seguinte gera efetivamente o conteúdo dos arquivos fonte.

```
<AdTemplate name="AdCFunction" section="lib">
    /* [sty:value @name]_lib.c --
     *
```

```
     *    Library source for [sty:value @name]
     *
     */
     #include "[sty:value /AdToolbox/@name].h"
     [sty:value Source]
     DLLEXPORT [sty:apply . prototype]
     \{
        [sty:value Source/Code]
     }
</AdTemplate>
```

3. Para gerar o protótipo da função, o tipo C do valor de retorno e dos argumentos deve ser obtido indiretamente através do atributo `type`. O valor deste atributo é utilizado para localizar a definição do tipo de dados, elemento `AdDtatypes/AdDatatype`. O atributo `AdDatatype/@ctype` especifica o tipo C.

```
<AdTemplate name="AdCFunction" section="prototype">
    [sty:value [xpath {//AdCDatatype[@name='%%']/@ctype} \
               [sty:value Return/@type]]]\
    [sty:value @name]([sty:apply Args arglist])
</AdTemplate>

<AdTemplate name="Args" section="arglist">
    [join [sty:apply Arg arglist] ", "]\
</AdTemplate>

<AdTemplate name="Arg" section="arglist">
    {[sty:value [xpath {//AdCDatatype[@name='%%']/@ctype} \
                [sty:value @type]]] [sty:value @name]}
</AdTemplate>

</AdStylesheet>
```

Para aplicar o estilo à nossa toolbox *zeta*, usamos os seguintes comandos na console do Adesso:

```
% cd mydir
```

```
% adesso load -tbx zeta
% adesso codegen -target library -lang tbx2c -sty tbx2c.sty
```

O seguinte arquivo é gerado para a única função da toolbox *zeta*:

```
/* clone_lib.c --
 *
 *    Library source for clone
 *
 */
#include "zeta.h"
DLLEXPORT adimage * clone_lib (adimage * img )
{
    int w, h, d, t;
    adimage *img_out;
    if (img == NULL) return NULL;
    w = img->row_size;
    h = img->col_size;
    d = img->num_data_bands;
    t = adimage_type(img);
    img_out = adimage_create(w, h, d, t);
    return img_out;
}
```

8.2.2 Interfaces para Linguagens de Scripting

Segue um exemplo simplificado de geração de código de interface para o Tcl/Tk.

1. O início é semelhante ao estilo da biblioteca base. É gerado um arquivo C para cada função da toolbox cujo atributo `exposed` seja verdadeiro. Este atributo indica que a função deve ter uma interface com a linguagem de scripting.

   ```
   <AdStylesheet target="wrapper" lang="tbx2tcl">

   <AdStylesheet target="library" lang="tbx2c" xref="tbx2c.sty"/>
   ```

```
<AdTemplate name="AdToolbox">
    [sty:par outdir check .]
    [sty:foreach {//AdCFunction[@exposed="yes"]} {
        [sty:save [file join [sty:par outdir] \
                         [sty:value @dir] \
                         [sty:value @name]_wrapper.c] \
                        {[sty:cb [sty:apply . wrapper]]}]
    }]
</AdTemplate>
```

2. O template `AdCFunction.wrapper` efetua a geração do código. O corpo da função de interface é formado por um conjunto de seções que são preenchidas com código gerado através de *macros* denominadas *typemaps*.

```
<AdTemplate name="AdCFunction" section="wrapper">
    [sty:apply . init]
    /* [sty:value @name]_wrapper.c --
     *
     *    TCL Wrapper for function [sty:value @name]
     *
     */
    #include "tcl.h"
    #include "[sty:value /AdToolbox/@name].h"

    int
    [sty:value /AdToolbox/@prefix]_[sty:value @name]_wrapper(\
                                    ClientData clientData,    \
                                    Tcl_Interp *interp,       \
                                    int objc, Tcl_Obj *CONST objv[])
    {
        Tcl_Obj *tcl_result;

        /* == Declaração == */
        [apply . decl]

        /* == Conversão == */
        [apply . convert]
```

```
        /* == Verificação == */
        [apply . check]

        /* == Chamada do procedimento base == */
        [apply . call]

        /* == Saída == */
        [apply . output]

        /* == Liberação de memória == */
        [apply . free]

        return TCL_OK;
    }
</AdTemplate>
```

3. *Declaração*
   Nesta seção as variáveis correspondentes aos argumentos da função e ao valor de retorno são declaradas.

```
<AdTemplate name="AdCFunction" section="decl">
    [sty:foreach Args/Arg {
        [sty:typemap decl [att dir] [att type] [att map]]
    }]
    [sty:foreach Return {
        [sty:typemap decl ret [att type] [att map]]
    }]
</AdTemplate>
```

4. *Conversão*
   Argumentos de entrada da função, `dir="in"`, são convertidos de sua representação na linguagem Tcl para sua representação na linguagem C, a ser passada ao procedimento da biblioteca base.

```
<AdTemplate name="AdCFunction" section="convert">
    [sty:foreach {Args/Arg[@dir='in']} {
```

```
        [typemap convert [att dir] [att type] [att map]]
    }]
</AdTemplate>
```

5. *Verificação*
   Esta seção complementa a seção de conversão, oferecendo uma oportunidade para testes de consistência entre os argumentos de entrada.

```
<AdTemplate name="AdCFunction" section="check">
    [sty:foreach {Args/Arg[@dir='in']} {
        [typemap check [att dir] [att type] [att map]]
    }]
</AdTemplate>
```

6. *Chamada*
   Aqui, o procedimento base, parte da biblioteca base, é executado para gerar os valores de saída da função.

```
<AdTemplate name="AdCFunction" section="call">
    [sty:value Return/@name] = [sty:apply tbx2c.library.libname]\
                               ([sty:apply Args tbx2c.library.calllist]);
</AdTemplate>
```

7. *Saída*
   Os valores de saída gerados pela chamada anterior são convertidos para a representação em Tcl e passados ao interpretador.

```
<AdTemplate name="AdCFunction" section="output">
    [sty:foreach Return {
        [typemap output ret [att type] [att map]]
    }]
    [sty:foreach {Args/Arg[@dir='out']} {
        [typemap output [att dir] [att type] [att map]]
    }]
</AdTemplate>
```

8. *Liberação de memória*
   Porções de memória eventualmente alocados pelas funções chamadas nesta interface são liberadas nesta seção.

```
<AdTemplate name="AdCFunction" section="free">
    [sty:foreach Return {
        [typemap free ret [att type] [att map]]
    }]
    [sty:foreach Args/Arg {
        [typemap free [att dir] [att type] [att map]]
    }]
</AdTemplate>
```

9. *Mapas de tipos*

   O trecho seguinte do estilo mostra um mapeamento de tipo. O tipo `adimage` corresponde a uma estrutura em C que representa uma imagem,

   `AdDatatype/@ctype = 'adimage *'.`

   Os mapas são avaliados no contexto de um nó tipo `Arg` ou `Return`, dependendo do caso. Por exemplo, o comando

   ```
   [typemap output in adimage ""]
   ```

   causa a avaliação do mapa

   ```
   AdTypemap[type="adimage"]/Map[dir="in" and section="output"]
   ```

   no contexto de um nó `Arg`.

   Os mapas com atributo `dir="in"`, `dir="inout"` e `dir="out"` são avaliados para nós `Arg`; mapas com `dir="ret"` referem-se a nós `Return`.

   ```
   <AdTypemap type="adimage">

       <Map dir="in" section="decl">
           [att _ctype] [att name] = NULL;
           int __[att name] = 0;
       </Map>
       <Map dir="in" section="convert">
           [att name] = ([att _ctype])tclObj2adPointer("[att _alias]", \
   ```

```
                        [att _obj]);
            if([att name] == NULL) {
                return usage_error(interp, "Invalid argument: [att name]");
            }
        </Map>
        <Map dir="in" section="free">
            if(__[att name] &amp;amp;&amp;amp; [att name])
                freeadPointer("[att _alias]", [att name]);
        </Map>
        <Map dir="ret" section="decl">
            [att _ctype] [att name] = NULL;
        </Map>
        <Map dir="ret" section="output">
            if(![att name]) {
                Tcl_SetObjResult(interp, Tcl_NewStringObj(adErrorMsg(), -1));
                return TCL_ERROR;
            }
            Tcl_SetObjResult(interp, adPointer2tclObj("[att _alias]", \
                        (void *)[att name]));
        </Map>

    </AdTypemap>
```

Para aplicar o estilo à nossa toolbox *zeta*, usamos os seguintes comandos na console do Adesso:

```
% cd mydir
% adesso load -tbx zeta
% adesso codegen -target wrapper -lang tbx2tcl -sty tbx2tcl.sty
```

Apresentamos a seguir o código gerado para a toolbox *zeta*:

```
/* clone_wrapper.c --
 *
 *    TCL Wrapper for function clone
 *
 */
```

```
#include "tcl.h"
#include "zeta.h"
int
clone_wrapper(ClientData clientData, Tcl_Interp *interp, int objc,
                                                Tcl_Obj *CONST objv[])
{
    Tcl_Obj *tcl_result;
    /* == Declaração == */
    adimage * img = NULL;
    int __img = 0;
    adimage * img_out = NULL;
    /* == Conversão == */
    img = (adimage *)tclObj2adPointer("adimage", objv[0]);
    if(img == NULL) {
        return usage_error(interp, "Invalid argument: img");
    }
    /* == Verificação == */
    /* == Chamada do procedimento base == */
    img_out = clone_lib (img );
    /* == Saída == */
    if(!img_out) {
        Tcl_SetObjResult(interp, Tcl_NewStringObj(adErrorMsg(), -1));
        return TCL_ERROR;
    }
    Tcl_SetObjResult(interp, adPointer2tclObj("adimage", (void *)img_out));
    tcl_result = Tcl_GetObjResult(interp);
    /* == Liberação de memória == */
    if(__img && img) freeadPointer("adimage", img);
    return TCL_OK;
}
```

Para completar o conjunto de arquivos necessários para a construção da toolbox, devemos criar um “Makefile” e um arquivo de índice (`pkgIndex.tcl`) que permita a identificação de nosso binário como um módulo de extensão do Tcl/Tk.

O trecho de sessão seguinte ilustra a utilização de nossa toolbox:

```
c:\> tclsh
```

```
% package require zeta
0.1
% zz::clone {{{1 2 3} {5 6 7}}}
{{376 122 131} {222 456 2118}}
```

Capítulo 9

Conclusões

Este projeto criou um sistema de desenvolvimento de software científico e possibilitou a criação de algumas aplicações que demonstraram sua viabilidade.

O conceito fundamental que embasa o sistema desenvolvido é a busca da separação do conteúdo líquido de um projeto científico de suas diferentes apresentações ou interfaces de utilização, seja no que se refere ao código propriamente dito ou à documentação associada. O mesmo conteúdo pode dar origem a diferentes apresentações dependendo da transformação efetuada, de maneira similar aos sistemas de bases de dados, onde diferentes relatórios e telas de entrada de dados refletem o mesmo conjunto de informações com diferentes finalidades. A estes sistemas tradicionais, este trabalho acrescenta um sistema de transformação de documentos inspirado nos padrões de documentação SGML/XML, inclusive para a geração de código.

O sistema resultante, denominado Adesso, permite que diferentes *produtos* sejam criados a partir do mesmo conjunto de informações, organizado através de documentos XML, através da aplicação de diferentes *estilos de transformação*.

A utilização do Adesso para o desenvolvimento de várias *toolboxes* comprovou a eficácia do processo de transformação de documentos como elemento central da criação e distribuição de conjuntos de componentes de software. O processador de estilos, implementado através da incorporação de algumas construções adicionais à linguagem Tcl,

é uma ferramenta de produtividade que se mostrou flexível e simples, principalmente para programadores com alguma experiência na linguagem Tcl. A grande variedade de estilos de transformação desenvolvidos ocasionou o aperfeiçoamento gradativo do processador de estilos e sua linguagem.

As principais contribuições deste trabalho são listadas a seguir.

- A manutenção das informações das *toolboxes* em bases de dados estruturadas é importante para garantir a consistência dos documentos e código gerados para cada *toolbox*. O modelo de desenvolvimento de software baseado em componentes, presente em outros sistemas como o Khoros [2][3], por exemplo, forma a base do Adesso. Neste sistema, contudo, este modelo é fatorado para separação das informações relativas à *plataforma de integração*, que são incorporadas às folhas de estilo. Desta forma, uma mesma *toolbox*, processada com diferentes folhas de estilo, dá origem a diferentes *produtos*, ou seja, pode ser utilizada em diferentes *plataformas de integração*.

- Outra contribuição relevante é a utilização do XML para representação das informações manipuladas pelo Adesso. O XML possibilitou o aproveitamento de ferramentas existentes, como o **tDOM**, por exemplo, para o desenvolvimento do sistema de transformação do Adesso. A oferta crescente de ferramentas para criação, manutenção e transformação de documentos XML deverá ter impacto importante em desenvolvimentos futuros do Adesso.

- O processador de estilos é a principal contribuição do projeto Adesso. O processador de estilos do Adesso oferece uma alternativa mais simples e, em alguns aspectos, mais poderosa que o XSLT para a transformação de documentos XML.

- Foram desenvolvidos ao longo deste projeto diversas folhas de estilo para geração de documentação e código de *toolboxes*, utilizando a linguagem de estilos do Adesso. Duas *plataformas de integração* foram focalizadas com maior ênfase até aqui: o MATLAB e o Tcl/Tk. Alguns trabalhos estão em andamento para incluir suporte para criação de toolboxes para a linguagem Python [17][18].

- Finalmente, destacamos o desenvolvimento de várias toolboxes e algumas aplicações. Dentre as toolboxes criadas, a toolbox de Morfologia Matemática deve ser destacada [19]; trata-se de um conjunto extenso de funções de processamento de imagens que pode ser utilizado no MATLAB, Tcl/Tk e, em breve, Python. A principal aplicação desenvolvida foi o *ProntoVideo*, um sistema de segmentação assistida de vídeos digitais [20], [21]. O *ProntoVideo* foi implementado em Tcl/Tk, usando o suporte provido por cinco toolboxes criadas com o Adesso.

9.1 Trabalhos Futuros

Os trabalhos realizados no âmbito deste projeto originaram uma série de possibilidades de continuação, seja para aperfeiçoamento do sistema através do desenvolvimento de aplicações, seja para a exploração de sua utilização em ambientes baseados na Internet, aproveitando sua proximidade com estes sistemas devido à sua utilização do XML. Algumas destas possibilidades são enumeradas em seguida.

- **Adesso Web**
  Este projeto visa a integração do Adesso a um sistema de autoria baseado na Internet. A utilização do Adesso em conjunto com um sistema de gerência de conteúdo para a Web, como o Zope, por exemplo, abre novas possibilidades para o desenvolvimento de software científico, enfatizando a cooperação entre os desenvolvedores. O Zope, um forte candidato para constituir a base de um sistema deste tipo, é um sistema de gerência de documentos para a Web fortemente orientado a objetos (baseado na linguagem Python), com uma base de dados como *back-end* e um sistema de controle de sessão e segurança. Outro ponto que pode ser explorado pelo projeto é a criação de parques de compilação que viabilizaria a criação centralizada de binários para diferentes plataformas computacionais.

- **Ambiente de Programação Visual Integrado ao Adesso**
  A utilização de linguagens visuais para o desenvolvimento de aplicações científicas foi muito popularizada por sistemas como o Khoros e o Matlab/Simulink.

Os altos custos de um sistema como o Matlab e as restrições impostas ao uso do Khoros ultimamente criam uma boa oportunidade para o surgimento de novos sistemas deste tipo. O Adesso pode ser a base de um sistema de programação visual, tendo sido, inclusive, concebido visando este tipo de aplicação.

- **Integração do Adesso e Sistemas de Controle de Versões**
  O desenvolvimento de toolboxes para o Adesso tem sido feito com o suporte de um sistema de controle de versões, o CVS. Uma maior integração entre os dois sistemas é muito desejável para facilitar o uso do CVS e facilitar a gerência de configurações.

- **Ferramentas para Análise de Programas**
  As ferramentas simples de importação de código legado de que dispõe o Adesso podem constituir a base de um sistema mais completo para análise e importação de programas existentes.

Referências Bibliográficas

[1] J. M. Adán-Coello; M. F. Magalhães; *STER's Multilevel Programming Model for Distributed Hard Real-Time Systems* - Microprocessing and Mircoprogramming, vol. 34, 1992 - North-Holland;

[2] R. M. S. Luppi, D. M. Kligerman, A. X. Falcão, U. M. Braga Neto, A. J. Vieira e R. A. Lotufo; *V3DTOOLS: A KHOROS Toolbox for 3D Imaging*, World Congress on Medical Physics and Biomedical Engineering 94, Rio de Janeiro, agosto de 1994.

[3] J. Barrera, G. J. F. Banon, R. A. Lotufo, and R. Hirata Jr.; *MMach: a Mathematical Morphology Toolbox for the Khoros System*, Journal of Electronic Imaging, vol 7, no. 1, pp.174-210, January 1998.

[4] J. K. Ousterhout; *Scripting: Higher Level Programming fot the 21st Century*, IEEE Computer, March 1998.
`http://www.scriptics.com/people/john.ousterhout/scripting.html`

[5] W3C Recommendation 10-Feb-98; *Extensible Markup Markup (XML) 1.0*
`http://www.w3.org/TR/REC-xml-19980210`

[6] W3C Working Draft 7 April 2000; *XML Schema Part 0: Primer*
`http://www.w3.org/TR/xmlschema-0`

[7] W3C Working Draft 7 April 2000; *XML Schema Part 1: Structures*
`http://www.w3.org/TR/xmlschema-1`

[8] W3C Working Draft 7 April 2000; *XML Schema Part 2: Datatypes*
http://www.w3.org/TR/xmlschema-2

[9] W3C Working Draft 27 Mach 2000; *Extensible Stylesheet Language (XSL) Version 1.0*
http://www.w3.org/TR/xsl

[10] W3C Recommendation 16-Oct-98; *Document Object Model (DOM) Level 1 Specification Version 1.0*
http://www.w3.org/TR/REC-DOM-Level-1

[11] Jochen Loewer; *tDOM — A fast XML/DOM/XPath package for Tcl written in C*
http://sdf.lonestar.org/ loewerj/tdom.cgi

[12] W3C Recommendation 16-Nov-99; *XML Path Language (XPath) Version 1.0*
http://www.w3.org/TR/xpath

[13] W3C Recommendation 16-Nov-99; *XSL Transformations (XSLT) Version 1.0*
http://www.w3.org/TR/xslt

[14] IBM; *Xeena, a visual XML editor*
http://www.alphaworks.ibm.com/aw.nsf/techmain/xeena

[15] André Vital Saúde; *Interface Configurável de Desenvolvimento de Aplicações para Processamento de Imagens*. Plano de mestrado, 2000.

[16] Norman Walsh, Leonard Muellner; *DocBook: The Definitive Guide*, O'Reilly, ISBN 1-56592-580-7, 1999.

[17] Alexandre Gonçalves Silva, *Processamento de imagens com Python*. Plano de Mestrado, 2000. Bolsa FAPESP Proc. n. 00/13671-0.

[18] A. G. Silva, R.A. Lotufo, R. C. Machado, *Toolbox of Image Processing for Numerical Python*, Poster, Sibgrapi 2001, Florianópolis, outubro de 2001.

[19] SDC; *SDC Morphology Toolbox for Matlab*
`http://www.mmorph.com`

[20] R. Koo; *Segmentação Assistida de Imagens e Vídeos Digitais*, relatório final, Projeto PIPE, FAPESP n. 97/13306-6, setembro de 2001.

[21] R. Lotufo, R. Machado, F. Flores, A. Falcão, R. Koo, G. Mazzela, R. Costa; *Prontovideo – An image sequence segmentation tool applied to video edition*, Poster, Sibgrapi 2001, Florianópolis, outubro de 2001.

[22] Harald Kirsch; *bras – A Rule Based Command Execution with Tcl*
`http://wsd.iitb.fhg.de/ kir/brashome`