Apêndice A

Ambiente mmil

> *Uma linguagem-fonte de sucesso está inclinada a ser implementada em várias máquinas-alvo. Se a linguagem sobreviver, os compiladores para a mesma necessitarão gerar código para várias gerações de máquinas-alvo... por conseguinte, compiladores reorientáveis estão inclinados a desempenhar algum papel. Logo, o projeto de linguagens intermediárias é importante, na medida em que confina detalhes específicos de máquina para um pequeno número de módulos.* *[ASU95]*

Neste apêndice será apresentado o ambiente de geração automática de códigos e documentos para operadores morfológicos, chamado *mathematical morphology intermediary language*, ou *mmil*. Este ambiente faz uso de elementos de alto desempenho que definem a nossa *linguagem intermediária*, também apresentada neste apêndice. O ambiente *mmil* foi desenvolvido no *Adesso*, um ambiente computacional de suporte ao desenvolvimento de aplicações científicas como uma caixa de ferramentas [Mac02].

Para explicar como funciona o ambiente *mmil*, este apêndice será dividido da seguinte forma: na Seção A.1 será apresentada a motivação para criação deste ambiente; na Seção A.2 será apresentada uma introdução ao ambiente desenvolvido, comentando as linguagens de programação utilizadas

e descrevendo a sua arquitetura, destacando a Figura A.2; na Seção A.3 será apresentada a *linguagem intermediária*; na Seção A.4 serão descritos os elementos utilizados pelos operadores que compõem a *linguagem intermediária* do ambiente *mmil*; nas Seções A.5, A.6 e A.7, serão apresentados alguns exemplos ilustrativos da programação usando os elementos definidos, mostrando nas duas primeiras seções os códigos correspondentes em *XML* e *MATLAB*.

Nos exemplos da Seção A.7 não serão mostrados os códigos *XML* e *MATLAB*, pois eles não são códigos para leitores humanos, mas para as máquinas interpretarem. Serão mostradas apenas as expressões matemáticas equivalentes, que podem ser geradas automaticamente através dos códigos em *XML*.

A.1 Motivação

Serão apresentadas nesta seção as várias linhas de pesquisa para o desenvolvimento de software para processamento de imagens. Serão analisados os paradigmas de programação e a organização da informação no processo de desenvolvimento de software; serão apresentados a evolução e os problemas de uma ferramenta particular de processamento de imagens, chamada inicialmente de *MMach*; serão apresentadas também as soluções encontradas na literatura para resolver parte destes problemas; finalmente, será apresentada a nossa solução para estes problemas.

A análise e o processamento de imagens por computador digital possibilita resolver problemas em várias áreas da ciência [BB94, GW92]. Porém, a criação e o uso de ferramentas de processamento de imagens pode muitas vezes impedir o sucesso de uma aplicação devido à complexidade, às limitações e à grande quantidade de transformações e parâmetros existentes [D'O01]. Em *transformações* serão incluídas todas as *operações* (soma, subtração, complemento, etc.), *operadores* (dilatação, erosão, etc.) e *transformações geométricas* (translação, rotação, etc.).

Programação genérica

O estado da arte em linguagens de programação é a *programação genérica* [Aus99]. Assim como o conceito de orientação a objetos revolucionou a programação no início da década de 90, a programação genérica está revolucionando a programação no início desta década, onde o objetivo é generalizar os algoritmos de forma a facilitar a sua reutilização.

O paradigma da *programação genérica* permite construir algoritmos abstratos encapsulando os tipos e estruturas de dados. A linguagem de programação *C++* possibilita a escrita destes tipos de algoritmos usando a biblioteca *STL* (*Standard Template Library*). O paradigma da *programação orientada a objetos* permite abstrair estruturas de dados encapsulando os algoritmos. Outro paradigma bem conhecido dos programadores é a *programação estruturada*, onde os esforços estão em implementar os mesmos para diferentes estruturas e tipos de dados e isto implica na multiplicação de esforços. A programação genérica busca distribuir as atividades de implementação de algoritmos usando diferentes estruturas e tipos de dados.

Essas três atividades de programação podem ser simbolizadas pelo eixo octagonal tridimensional, onde os esforços de construção de estruturas de dados estão no eixo X, dos tipos de dados estão no eixo Y e dos algoritmos estão no eixo Z. Segundo D'Ornellas *et. al.* [DCB+02], *um dos primeiros registros de iniciativas que buscaram organizar e estabelecer os reais limites e elementos envolvidos na programação geral coube à Wirth [Wir76] ao propor que houvesse uma separação semântica no desenvolvimento de programas, tratando algoritmos e estruturas de dados em separado e entendendo que a união de ambas é que viria a definir o programa.* Por esta análise, se os esforços de desenvolvimento forem separados, um esforço total de $X + Z$ é encontrado para os algoritmos e estruturas de dados. Com a evolução da programação percebeu-se que as atividades relacionadas ao tratamento de tipos de dados devem assumir uma atividade distinta. Assim, se essas três atividades forem tratadas em separado e se todas as possibilidades de imple-

mentação forem cobertas, os esforços para a programação são encontrados em $X + Y + Z$. Se isto não ocorrer, é obtido no pior caso um esforço total de $X * Y * Z$ para cobrir todas as possibilidades de implementação. Isto pode ser facilmente verificado na programação estruturada, onde são multiplicados os códigos quando são mudados as estruturas e tipos de dados. O paradigma da programação que ajuda a obter este esforço linear nas atividades de programação é chamado de *programação genérica*, onde as estruturas e tipos de dados estão encapsulados nos algoritmos.

Indo mais além nessa análise da evolução da programação, será analisado a construção de um software e não simplesmente as atividades de programação em uma linguagem específica, não descartando os esforços e as descobertas alcançadas na programação. Considerando um software formado por programas e documentações, os programas podem ser definidos em diferentes linguagens e os documentos podem ser escritos também usando diferentes editores. Uma sugestão para distribuir ainda mais as atividades para a construção de software seria definir mais uma ou duas dimensões no eixo octagonal tridimensional incorporando as atividades de linguagens de programação e editoração. Por exemplo, considere os eixos W para as linguagens de programação e K para os editores de texto. Assim, considere a união dos esforços para o desenvolvimento de um software distribuídos nos eixos X, Y, Z, W e K. Se foi possível, com muito esforço, construir um software na linguagem $C++ \in W$ em $X + Y + Z$, então, se for preciso mudar de linguagem, o software deverá ser reescrito na nova linguagem e torcer para que esta suporte também um esforço de $X + Y + Z$. Analogamente para a documentação escrita em $LaTeX \in K$. O ideal seria definir uma estrutura para armazenamento de informação incorporando algoritmos e documentação, como descrito na próxima seção, e que a compilação desta estrutura gerasse códigos em diferentes linguagens de programação e em diferentes formas de documentações.

Organização da informação

Uma boa metodologia no processo de desenvolvimento de software é definir uma estrutura de armazenamento que possibilite processar os seguintes conceitos [McG99]:

Conteúdo é a informação em si;

Estrutura define a organização da informação;

Apresentação associa a forma de consumir a informação.

É de consenso que se estas três partes são separadas uma da outra, uma melhor utilização das informações é alcançada. Uma boa ilustração desses conceitos é realizada no sistema de editoração conhecido como *LaTeX*. O conteúdo é armazenado em um arquivo texto, a estrutura é armazenada em um arquivo de estilo (*book, article, report*, etc.) e a saída é alcançada pelo processador *Tex*. Em contraste, se um autor escrever seu documento usando somente *Tex*, a reutilização do documento é perdida. Por outro lado, é possível converter arquivos *LaTeX* para outros formatos.

Recentemente, com a proliferação da Internet e a necessidade de ter uma ferramenta eficiente de manipulação da informação, surgiu a linguagem *XML* (*EXtensible Markup Language*), onde o conteúdo é armazenado em uma linguagem de marcação, tal como em *HTML*, mas com a possibilidade de criar novos *tags*. A estrutura é definida através de *esquema* (ou *scheme*), que restringe o conteúdo da informação, como aceitar apenas números inteiros em um certo campo. Finalmente, a apresentação é definida através de *folhas de estilo* (ou *stylesheets*), que governam a tradução da informação para um formato de saída.

A.1.1 *MMach*

A *morfologia matemática* é uma das subáreas do processamento de imagens e se baseia em teoria dos conjuntos através de quatro classes de operadores elementares: dilatação, erosão, anti-dilatação e anti-erosão.

Uma implementação de transformações para morfologia matemática foi iniciada em 1992 com o projeto *MMach* (*Morphological Machine*) [BBL94]. Durante a evolução deste projeto foi criada uma biblioteca na linguagem *C*, denominada *MMachLib* [LZHJB97]. Serão apresentadas a seguir as principais características desta biblioteca.

Hierarquia

A *MMachLib* foi concebida de forma hierárquica. Então, muitos dos operadores são formados pela composição de operadores elementares e operações básicas de união e intersecção. Esta característica facilita o desenvolvimento de software [BB94]. Tipicamente as bibliotecas de processamento de imagens têm centena de transformações. Este fato complica a manutenção e cresce a probabilidade de ocorrência de erros na programação, devido à proliferação de código. No caso da *MMachLib*, as transformações hierárquicas foram exploradas com o objetivo de diminuir o volume de código e minimizar a probabilidade de ocorrência destes erros.

Não-hierarquia

O surgimento de algoritmos não-hierárquicos, com implementações com menor tempo de processamento, obrigou a *MMachLib* possuir rotinas especializadas. Em particular, rotinas que utilizam estruturas de filas mostraram-se muito eficientes, como é o caso do *watershed* e da *transformada de distância*. Outro exemplo é a *abertura por área*, em imagens em níveis de cinza, que utiliza filas hierárquicas. Outros algoritmos que estão fornecendo bons resultados são os seqüenciais, devido a sua eficiência, porém possuem a

desvantagem de não serem intuitivos comparados com os algoritmos paralelos.

Obsolescência de recursos

Com o avanço da tecnologia, o software e o hardware estão ficando obsoletos num período de tempo cada vez mais curto. Sistemas operacionais como *MS-Windows* ou *Unix* e programas como *MATLAB* mudam de versão em média a cada três anos, ou menos. Quando isto ocorre, geralmente mudanças nas ferramentas que usam estas plataformas são necessárias. Na prática, quando um software não possui uma boa metodologia, tais mudanças significam muitas vezes implementar tudo de novo! Existe hardware específico para o processamento de imagens e para a morfologia matemática, hardware aceleradores nas *CPU's* convencionais como o *MMX* do *Pentium*, e também a possibilidade de se explorar eficientemente o uso do processamento em paralelo. Para um melhor aproveitamento do hardware, as empresas de software criam novos programas, com desempenho bem superior às versões anteriores. Assim sendo, é preciso atualizar freqüentemente os aplicativos e isto evidencia a necessidade de criar ferramentas capazes de minimizar a árdua tarefa de reescrita de código. Quando isto não ocorre, o custo de reescrita dos códigos pode ser alto e muitas vezes inviabilizar um projeto de hardware especial. O ideal seria com pouco esforço gerar novos códigos que executassem, de modo eficiente, nas diversas arquiteturas disponíveis.

A.1.2 Questões

Este apêndice contribui para responder de forma eficiente as seguintes questões: seria possível criar uma *linguagem intermediária* para escrever operadores morfológicos, e como resultado ter geração automática de código em várias linguagens de programação, junto com suas documentações?

A.1.3 Soluções encontradas na literatura

Apply e *Adapt*

Uma linguagem de programação independente de arquitetura para visão de baixo nível, chamada *Apply*, foi definida por Hamey *et. al.* [HWIC89]. Esta linguagem reduz o problema de escrita de algoritmos para visão de baixo nível, mas não pode ser usada em algoritmos globais, como *histograma* de imagens. A compilação da *Apply* converte um simples procedimento em uma implementação que pode ser executada eficientemente em linguagens como *C*. A linguagem *Adapt* foi definida em seguida por Webb [Web90], onde o processamento de imagens global foi incluído, baseado no modelo de divisão e conquista. Um estudo do desempenho da linguagem *Apply* foi feito por Walace *et. al.* [WWIC89] para vários problemas de visão computacional em máquinas paralelas.

Horus

Horus é um ambiente de processamento de imagens escrito na linguagem *C++* que usa a biblioteca *STL* (*Standard Template Library*) e é baseado na programação genérica e na programação orientada a objetos [KPS00]. Este software classifica as transformações de processamento de imagens em padrões: *pontual paralelo* (negação, complemento, etc); *binário paralelo* (adição, subtração, menor que, etc); *redução* (máximo, mínimo, etc); *convolução generalizada paralela* (convolução, erosão, dilatação, etc); *vizinhança paralela* (mediana, etc); *vizinhança recursiva paralela* (*transformada de distância*, implementações recursivas, etc); *geométrica* (rotação, reflexão, etc). Para cada padrão existe um algoritmo, que tem como parâmetro uma *string* que representa o índice de uma tabela contendo todas as transformações deste padrão. Isto define a programação genérica no *Horus*. Os tipos de dados no *Horus* são definidos através de seis elementos, definindo a

estrutura de dados genérica das imagens.

Horus com operadores morfológicos

Um estudo de padrões de algoritmos morfológicos foi feito por D'Ornellas [D'O01], onde três padrões de algoritmos foram definidos: paralelo, seqüencial e baseado em filas. Classificações semelhantes também foram estudas por outros pesquisadores [Vin92, BHJ97, ZL00]. D'Ornellas [D'O01] adicionou ao *Horus* estes padrões criando um ambiente de programação genérico e orientado a objetos para operadores de morfologia matemática, onde representações abstratas em algoritmos foram implementadas.

OLENA

Baseado nas conclusões de D'Ornellas [D'O01], Darbon *et. al.* [DGDL02] criou o ambiente *OLENA*, uma biblioteca dedicada a usuários de morfologia matemática com código fonte disponível. Este trabalho apresenta de forma simples um ambiente de processamento de imagens, onde escreve operadores morfológicos, como dilatação, erosão e *watershed*, usando programação genérica para diversos tipos de dados e dimensões da imagem.

A.1.4 Solução proposta neste apêndice - *mmil*

Será proposta neste texto a seguinte solução: a *MMachLib* seria reescrita em uma linguagem de programação para algoritmos morfológicos, chamada *linguagem intermediária*, fazendo uso de elementos de alto desempenho (definida neste trabalho). Por exemplo, elementos de acesso à fila, de acesso à vizinhança de um pixel, de acesso a todos os pixels da imagem, etc. Nesta linguagem a codificação de quase todos os algoritmos de morfologia matemática seria possível. Existem casos onde mais que uma implementação

para um operador morfológico são necessárias. Por exemplo, a *transformada de distância* pode ser implementada usando filas, ou usando varredura de vizinhança seqüencial, ou usando composições de erosões. O conjunto de transformações implementadas usando a *linguagem intermediária* juntamente com as regras de geração de código (definidas pelos *esquemas* e pelas *folhas de estilo*) formam o nosso ambiente *mmil* (*mathematical morphology intermediary language*). Dado uma arquitetura, uma estimativa do tempo de execução pode ser feita para cada elemento do ambiente *mmil* e baseada nesta estimativa a melhor implementação pode ser escolhida para cada função na biblioteca. A *linguagem intermediária* é compilada usando o melhor algoritmo para a plataforma em questão. Neste ambiente, com o surgimento de novas máquinas e de novas linguagens, a tarefa de migração do código da *MMachLib* consistiria apenas na adaptação das rotinas de geração de código para as novas linguagens ou máquinas.

Linguagem intermediária

Usualmente as transformações escritas em linguagens de programação são implementadas em dois passos. No primeiro passo, poderia especificar através da *linguagem morfológica*. No passo seguinte, poderia implementar em uma linguagem de programação de propósito geral, como *C*. Entretanto, é desejável implementar um sistema mais genérico e flexível de forma que os operadores morfológicos serão descritos em uma *linguagem intermediária*. Esta *linguagem intermediária* permitirá sua tradução eficiente para diversas linguagens e arquiteturas.

Por exemplo, existem várias formas de implementar um operador morfológico. A Figura A.1 ilustra três algoritmos para a erosão: paralelo, seqüencial e por propagação. Esta figura também ilustra a tradução de um comando da *linguagem intermediária* para comandos da linguagem *C*. A implementação que tiver o melhor desempenho numa dada arquitetura será a

escolhida para a tradução.

O principal resultado deste trabalho é a definição da *linguagem intermediária*. Seus elementos devem ser em número reduzido, poderosos na utilidade e que sejam facilmente traduzidos nas diversas arquiteturas disponíveis, mesmo as de alto desempenho. Também se deseja compilar a *linguagem intermediária* para gerar código em diversas linguagens, por exemplo *C*, *MATLAB*, *Python* e *Tcl/Tk*, em várias plataformas, como *MS-Windows* e *Unix*, e em várias arquiteturas, como *MMX* e cartões aceleradores de processamento de imagens.

Apesar do nome *linguagem intermediária* ser bem genérico, neste documento será usado este nome para descrever um passo intermediário entre a programação de alto nível, descrita por uma linguagem natural como a linguagem morfológica, e uma linguagem de programação de propósito geral.

Além disso, existem diversos tipos de usuários no desenvolvimento e no uso de um software de processamento de imagens. Para simplificar, serão definidos dois tipos de usuários. Os usuários que desenvolvem aplicativos, ou seja, usam um software de processamento de imagens para resolver problemas do mundo real, como segmentar imagens médicas usando o *MATLAB*. Em morfologia matemática estes usuários programam usando apenas a linguagem morfológica. Um outro tipo de usuário, que também será chamado projetista de software, é responsável pela criação de um software de processamento de imagens. A *linguagem intermediária* definida neste texto está voltada para este segundo tipo de usuário, para os projetistas de software para processamento de imagens.

A.2 Introdução ao ambiente *mmil*

Será apresentado neste apêndice um ambiente para a geração automática de código para operadores de morfologia matemática, chamado *mmil*. Este ambiente roda na linguagem *XML*, que usa as vantagens de armazenar os

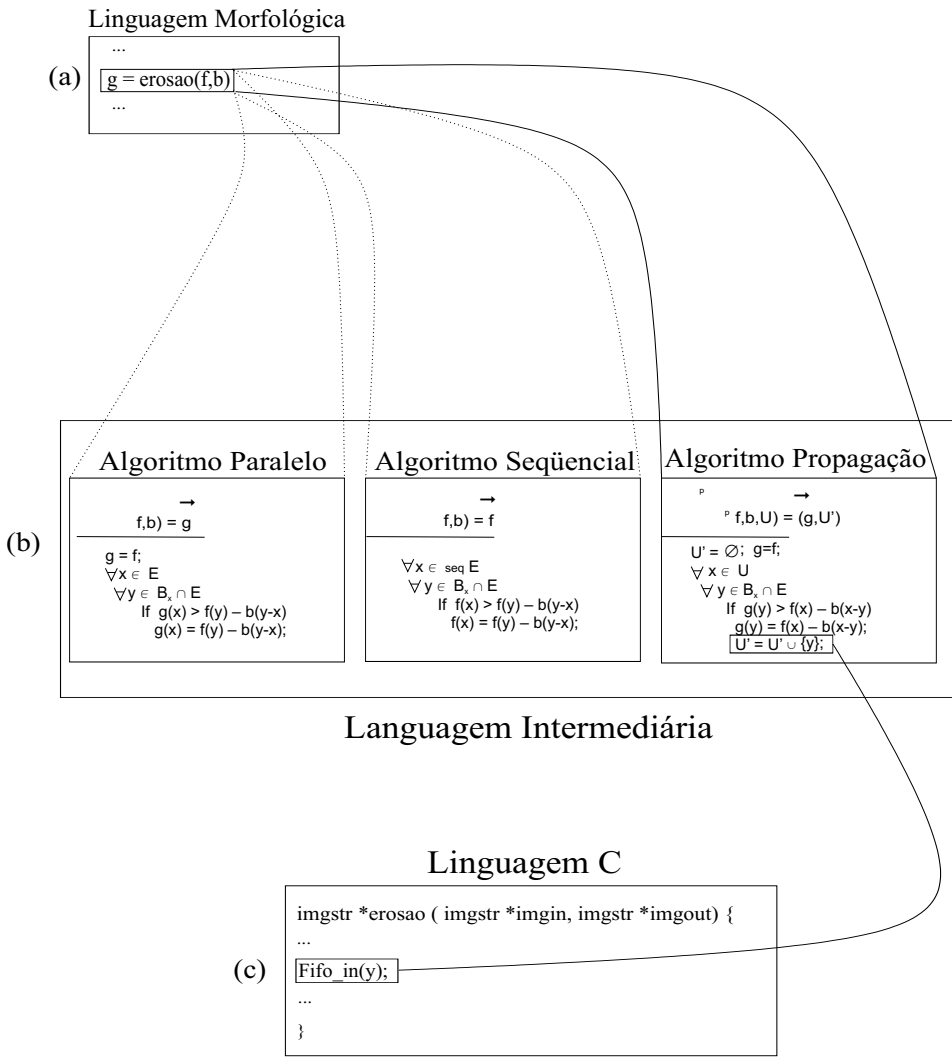

Figura A.1: Ilustração da relação entre: (a) *linguagem morfológica*, (b) *linguagem intermediária* e (c) *linguagem C*.

dados numa estrutura de árvore. A Figura A.2 mostra a arquitetura do ambiente *mmil*. Esta figura mostra um editor *GUI* (*Graphic User Interface*) para editar documentos *XML*. Em nossos estudos, estes documentos são operadores morfológicos implementados através dos elementos definidos na próxima seção. O conteúdo em *XML* é validado pelo *esquema*, que é uma estrutura contendo um conjunto de regras definidas para os atributos e elementos (nós da árvore) do *XML*. O conteúdo em *XML* é então processado pelas *folhas de estilo* gerando códigos em diversas outras linguagens (*C, MATLAB, PYTHON, TCL/TK*, etc), em diversas plataformas (*UNIX, LINUX, WINDOWS*, etc.), e também para gerar documentos (*LaTeX, HTML*, etc.).

Outra ferramenta usada na máquina de translação da *mmil* é a linguagem *TCL*, que trabalha junto com as *folhas de estilo* no processo.

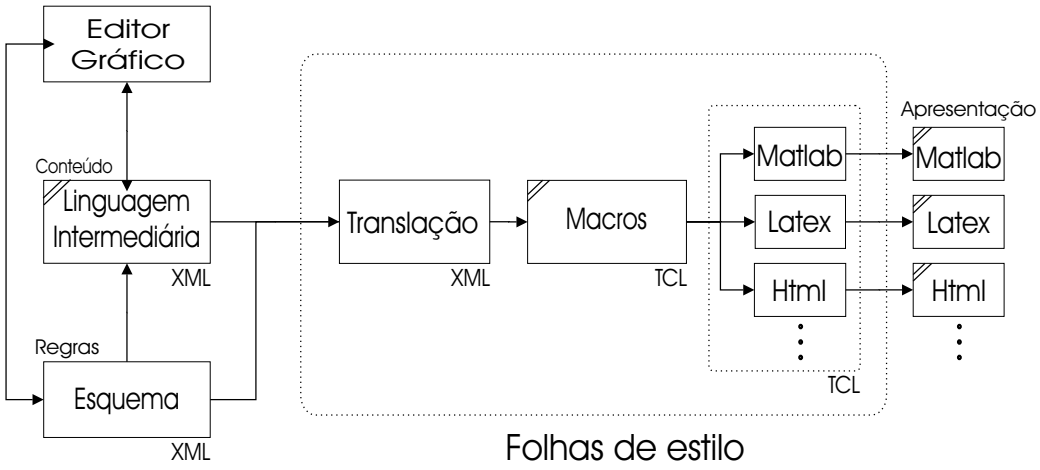

Figura A.2: Arquitetura da *mmil*.

Atualmente existem *folhas de estilo* para gerar códigos nas linguagens *MATLAB* e *C*. Também existe *folha de estilo* para gerar documentos em *LaTeX*.

A arquitetura da *mmil* foi desenvolvida no *Adesso*, um ambiente computacional de suporte ao desenvolvimento de aplicações científicas [Mac02].

Na Seção A.4 será apresentado um breve resumo dos elementos do modelo de informação do *Adesso* necessários para o desenvolvimento da linguagem intermediária.

A.3 Linguagem intermediária

Será apresentado nesta seção uma linguagem para algoritmos morfológicos, chamada *linguagem intermediária*, inspirado na *notação Z* [Spi88] e na *linguagem morfológica* [BB92].

A.3.1 Notação Z

A *notação Z* é utilizada para a especificação de problemas em engenharia de software através de notações matemáticas e possui uma ferramenta denominada *assistente de prova*, utilizada para prova automática das especificações [Pre02].

Como a *notação Z* modela problemas usando notação matemática, é natural usar em paralelo uma linguagem que possibilita editar símbolos matemáticos, como *LaTeX*. Neste sentido, um trabalho para servir de inspiração para mudança entre formatos é o conversor *Zed2XML* [CVM99], que transforma especificações Z escritas em *LaTeX* em documentos correspondentes em HTML [CVM99] e isto também pode ser realizado no ambiente proposto neste documento através das *folhas de estilo*, introduzidas na seção anterior. Existem também editores para a *notação Z*, que podem armazenar seus documentos em *LaTeX*, como o *ZCREATOR* [BC98] e visualizadores, como o *Z Browser* [Mik95].

A *notação Z* possui uma quantidade significativa de símbolos próprios, o que dificulta o aprendizado e a utilização. Por este motivo e baseado nos trabalhos existentes da *notação Z*, é possível criar um editor simplificado, que suporta a escrita dos padrões de algoritmos morfológicos estudados nos capítulos anteriores. Neste editor, além de poder visualizar as expressões

<operador>::=<operador elementar>|<limitante>|<composição>
<limitante>::=<argumento><operação de reticulado><argumento>
<argumento>::=<termo>|<composição>
<termo>::=<operador elementar>| (<limitante>)
<composição>::=<termo><termo>|<composição><termo>
<operador elementar>::=<operador morfológico>$_{\text{<função estruturante>}}$
<função estruturante>::=<letra>|<letra><número>
<número>::=<dígito>|<número><dígito>
<operação de reticulado>::= $\vee$ | $\wedge$
<operador morfológico>::= ε | δ | ε^a | δ^a
<letra>::= a | b | c | d
<dígito>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Tabela A.1: Gramática da linguagem morfológica [BB94].

matemáticas, é possível gerar códigos *LaTeX, HTML, C, MATLAB*, entre outros, como serão visto a seguir. Este editor seria uma proposta de continuação deste trabalho e não será discutido neste texto.

A.3.2 Linguagem morfológica

Existe na literatura uma linguagem formal para os operadores elementares de morfologia matemática [BB92, BB94], chamada *linguagem morfológica*. Veja na Tabela A.1 a gramática da linguagem morfológica. Esta linguagem tem como característica representar a dilatação, a erosão, a anti-dilatação e a anti-erosão por funções estruturantes. Porém, a linguagem intermediária definida neste documento, além de possuir esta característica, possui um vocabulário voltado para as diversas possibilidades de implementações destes operadores elementares.

A.4 Elementos da linguagem intermediária

A *notação Z* pode ser armazenada numa linguagem intermediária chamada *linguagem intercâmbio* (*interchange language*) [Har96] usando ele-

mentos (*tags*) para armazenar as informações de forma semelhante à linguagem *XML*. Analogamente, a linguagem *XML* é utilizada para armazenar a linguagem intermediária definida neste apêndice através de poucos elementos, como serão apresentados nesta seção.

Antes de definir os elementos da linguagem intermediária, será apresentado um breve resumo de onde estes elementos foram incluídos na estrutura do *Adesso*. Para mais detalhes deste ambiente consulte [Mac02]. Seja o elemento chamado *AdFunctions*, com os filhos definidos pelo elemento *AdFunction*. Veja Figura A.3 [1].

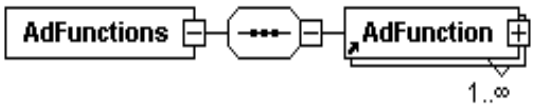

Figura A.3: Elemento *AdFunctions*.

O elemento *AdFunction* define as transformações implementadas, Figura A.4, e tem como filhos: *Platforms* – define as plataformas onde serão gerados os códigos; *Short* – descreve uma descrição da função; *Symbol* – associa um símbolo matemático; *Returns* – define os argumentos de retorno, veja Figura A.5; *Args* – define os argumentos de entrada através dos filhos definidos pelo elemento *Arg* contendo os atributos *name* e *type*, veja Figura A.6; e *Source* – descreve o código da função, veja Figura A.7.

A *linguagem intermediária*, onde são implementados os operadores morfológicos, é definida dentro do elemento *Code*, filho de *Source*, com atributo *lang* recebendo *"mmil"*, veja Figuras A.7 e A.8.

A *linguagem intermediária* é definida através de nove elementos: *Var* para variáveis, *Index* para acessar índices de vetor, *Set* para atribuições, *Oper* para transformações em geral, *Loop* para controles de repetição, *If* para operação condicional, *Call* para acessar outras funções já implementadas na *linguagem intermediária*, *Expand* e *Extract* para transformações que usam vizinhança.

[1] As Figuras A.3 até A.14 foram geradas pelo software *XMLSpy*.

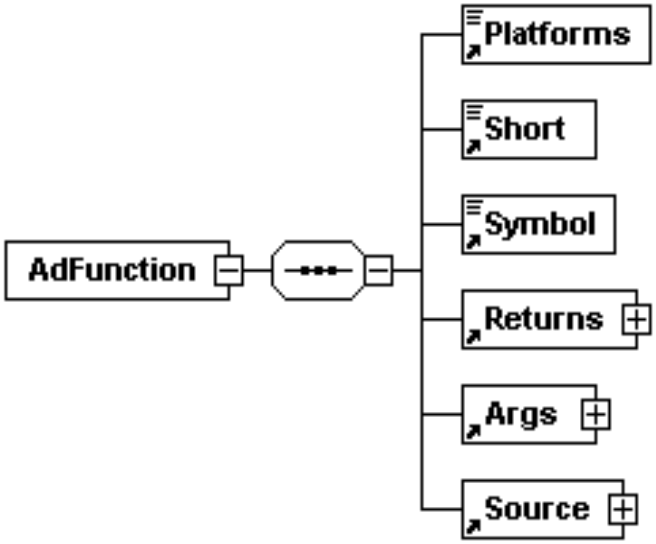

Figura A.4: Elemento *AdFunction*.

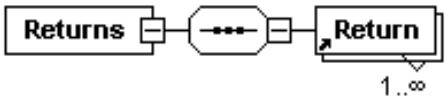

Figura A.5: Elemento *Returns*.

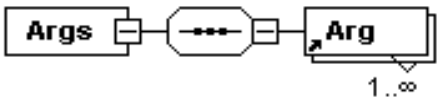

Figura A.6: Elemento *Args*.

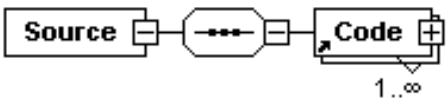

Figura A.7: Elemento *Source*.

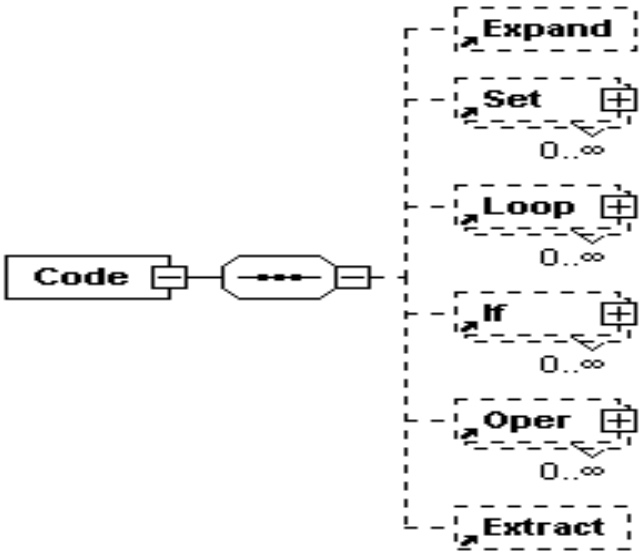

Figura A.8: Elemento *Code*.

Será descrito abaixo cada um destes elementos.

A.4.1 *Var*

Var é usado para acessar variável. Os seguintes casos podem ocorrer: o conteúdo é um valor escalar, uma matriz, ou um elemento de uma matriz (para o caso 2D). Neste último caso, um elemento de uma matriz é acessado pelo uso recursivo do elemento *Var* e pelo elemento *Index*. Veja Figura A.9.

Figura A.9: Elemento *Var*.

A.4.2 *Index*

Index é usado para acessar os elementos de uma matriz e é usado junto com o elemento *Var*. Veja Figura A.10.

Figura A.10: Elemento *Index*.

A.4.3 *Set*

Set é usado para especificar atribuições, onde existem duas partes, *left* e *right*. O conteúdo da parte *right* é associado a parte *left*. Associado à *left* existe o elemento *Var* e associado a *right* pode existir um dos filhos mostrados na Figura A.11.

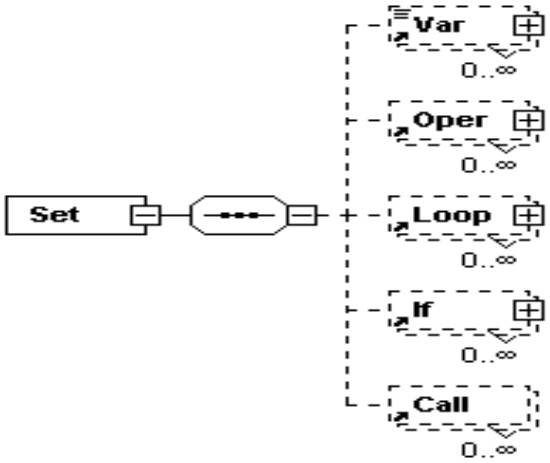

Figura A.11: Elemento *Set*.

A.4.4 *Oper*

Oper especifica uma transformação, que é dividida em três padrões: *pontual, global* e *geométrico*. O padrão *pontual* tem as operações: *adição* $(+)$, *subtração* $(-)$, *diferença* $(\neq)$, *igualdade* $(==)$, *união* $(\vee)$, *intersecção* $(\wedge)$, *negação* $(\sim)$, etc. O padrão *global* tem as operações *máximo* $(\bigvee)$, *mínimo* $(\bigwedge)$, etc. O padrão *geométrico* tem as transformações *translação* (B_x é a translação do conjunto B pelo vetor x), *find*, *fronteira* (∂), etc. Veja Figura A.12. Estas opções são definidas através dos seguintes elementos:

name especifica a operação;

i especifica um índice quando a operação é uma translação.

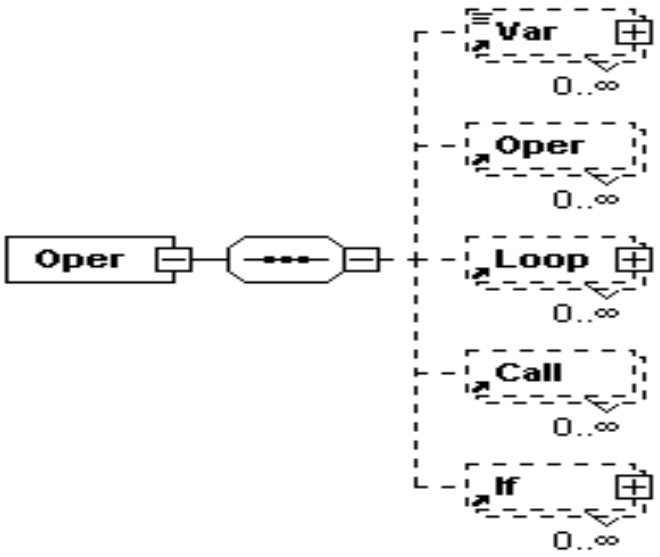

Figura A.12: Elemento *Oper*.

A.4.5 *Loop*

Loop este elemento faz referência a repetições *For* ou *While*, denotado por $\forall$. O laço é associado a um índice e a uma especificação de um domínio

através de atributos. O laço *While* requer uma expressão lógica. Veja Figura A.13. Para este elemento serão definidos os seguintes atributos:

name especifica o tipo de laço;

i especifica o índice para o laço;

D especifica o domínio de i;

Dse especifica o domínio da função estruturante;

oper especifica uma operação associada ao laço;

it especifica o índice se existir uma translação dentro de *Loop*;

Ds especifica o conjunto de armazenamento usado nos operadores por propagação;

s especifica o índice que vai percorrer o conjunto definido em *Ds*;

out especifica uma saída;

A.4.6 *If*

If é um comando condicional. Se a expressão lógica é *True*, o bloco associado ao elemento *If* é executado. Veja Figura A.14.

A.4.7 *Call*

Call é usado para referenciar uma operação implementada pela *mmil*. Serão definidos os seguintes elementos:

name especifica o nome da operação;

i1 especifica a primeira entrada;

i2 especifica a segunda entrada;

o1 especifica a saída.

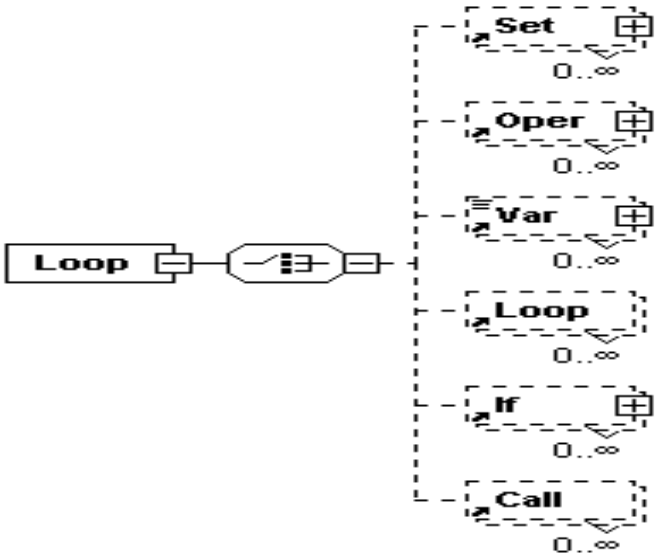

Figura A.13: Elemento *Loop*.

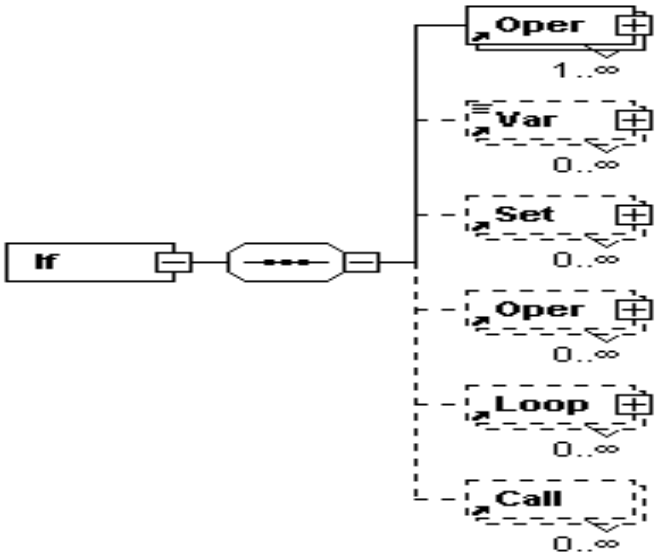

Figura A.14: Elemento *If*.

A.4.8 *Expand*

Expand expande uma imagem pela vizinhança passada como parâmetro. É passado também o valor a ser atribuído na borda expandida. Por exemplo, se for usado vizinhança 3×3, a imagem será expandida de uma linha e uma coluna ao redor da imagem. Para este elemento serão definidos os seguintes atributos:

Out especifica a saída expandida da imagem definida no atributo *In*;

Out1 especifica outra imagem com dimensões equivalentes à definida em *Out*, porém com valores definidos por *Value*;

In especifica a entrada;

SE especifica a vizinhança que define quanto vai expandir;

Value especifica o valor a ser atribuído à região expandida.

A.4.9 *Extract*

Extract extrai uma imagem pela vizinhança passada como parâmetro. Utilizado pelos operadores morfológicos que usam funções estruturantes. Para este elemento serão definidos os seguintes atributos:

Out especifica a saída;

In especifica a entrada;

SE especifica a vizinhança que define quanto vai diminuir a entrada.

A.5 Exemplo de uso dos elementos *Var* e *Index*

Nesta seção serão apresentados alguns exemplos ilustrativos da programação usando os elementos definidos anteriormente. Inicialmente, serão

mostrados exemplos simples com os elementos *Var* e *Index*. Em seguida, serão apresentados dois exemplos da operação de adição, mostrando o código *XML*, a expressão matemática e o código *MATLAB*. Finalmente, serão descritas três versões da erosão, dependendo da forma em que o laço é feito.

Exemplo A.1 *Este é um simples exemplo do uso do elemento Var.*

```
<Var>0</Var>
```

O retorno deste elemento é o valor zero. ◇

Exemplo A.2 *O valor de saída do elemento Var é a variável f que pode ser um valor escalar ou uma matriz, dependendo da forma em que foi definida.*

```
<Var>f</Var>
```

◇

Exemplo A.3 *Este exemplo usa os elementos Var e Index, devolvendo o conteúdo de um elemento de variável hist, definida por um elemento de f. Este exemplo pode ser reescrito simplesmente por hist*$(f(x))$.

```
<Var>
   hist
   <Index>
     <Var>f<Index>x</Index></Var>
   </Index>
</Var>
```

◇

A.6 Exemplos de adição em imagens

Nesta seção serão apresentados dois exemplos da operação de adição entre duas imagens com o mesmo domínio.

Exemplo A.4 *O exemplo abaixo faz a imagem g receber a adição da imagem $f1$ e $f2$. Estas três variáveis possuem o mesmo tamanho $\mathbb{E}$.*

```
...
<Set>
  <Var>g</Var>
  <Oper name="Add">
    <Var>f1</Var>
    <Var>f2</Var>
  </Oper>
</Set>
```

Este código em XML pode ser apresentado através da seguinte expressão:

$$g = f1 + f2,$$

ou ainda pelo seguinte código em MATLAB:

```
g = f1 + f2;
```

◇

Exemplo A.5 *Neste outro exemplo será mostrada uma outra versão para a adição de duas imagens. As três variáveis $f1, f2$ e g possuem o mesmo tamanho $\mathbb{E}$. Agora a varredura é explícita para o domínio $\mathbb{E}$:*

```
...
<Loop name="for" i="x" D="f1">
  <Set>
    <Var>g<Index>x</Index></Var>
```

```
    <Oper name="Add">
      <Var>f1<Index>x</Index></Var>
      <Var>f2<Index>x</Index></Var>
    </Oper>
  </Set>
</Loop>
```

Este código pode ser visto como a seguinte expressão (apresentada também no Algoritmo 2):

$\forall x \in \mathbb{E}$
$\quad g(x) = f_1(x) + f_2(x);$

ou ainda, pelo seguinte código em MATLAB:

```
for x=D(f1)
  g(x) = f1(x) + f2(x);
end
```

onde $\forall x \in \mathbb{E}$ *ou* $x = D(f1)$ *são todos os pixels da imagem* $f1$. ◇

Elemento *Call*

Exemplo A.6

```
<Call name="ero" i1="f" i2="b" o1="g"/>
```

Este exemplo é equivalente a $g = \varepsilon_b(f)$*, isto é,* g *recebe a erosão da imagem* f *pela função estruturante* b. ◇

A.7 Diferentes formas de implementar a erosão

Serão apresentadas três diferentes formas de implementar a erosão usando a linguagem intermediária através de três expressões matemáticas diferentes, porém equivalentes. A parte central deste exemplo é mostrar a flexibilidade do ambiente desenvolvido, particularmente do elemento *Loop*.

Nos exemplos a seguir não serão mostrados os códigos *XML* e *MATLAB*, pois não são códigos para leitores humanos, mas para as máquinas interpretarem. Serão mostradas apenas as expressões matemáticas equivalentes, que podem ser geradas automaticamente.

Exemplo A.7 *Seja $\mathbb{Z}$ o conjunto dos inteiros, $\mathbb{E} \subset \mathbb{Z}^2$ o domínio da imagem e $K = [0, k] \subset \mathbb{Z}$ um intervalo de números inteiros representando os possíveis níveis de cinza da imagem. O operador invariante por translação em níveis de cinza, $\varepsilon_b : K^{\mathbb{E}} \to K^{\mathbb{E}}$ ($K^{\mathbb{E}}$, lê-se conjuntos de funções de $\mathbb{E}$ em K), é definido como [Hei91]:*

$$\varepsilon_b(f)(x) = \min\{f(y) \dot{-} b(y - x) : y \in B_x \cap \mathbb{E}\},$$

onde $f \in K^{\mathbb{E}}$, $x \in \mathbb{E}$, $B \subseteq \mathbb{E} \oplus \mathbb{E}$ e B é chamado elemento estruturante), $B_x = \{y + x,\ y \in B\}$ (translação de B por x) e b é uma função estruturante definida em B com $b : B \to \mathbb{Z}$.

A equação acima, quando implementada na mmil, sem o tratamento de borda, tem a representação apresentada no Algoritmo 20 (veja Figura 2.4).

A expressão $\forall x \in \mathbb{E}$ é especificada pelo elemento Loop com o atributo for:

```
<Loop name="for" i="x" D="f">
```

O atributo i especifica um elemento do domínio de f, isto é, um elemento de $\mathbb{E}$. A expressão $\bigwedge_{\forall y \in B_x \cap \mathbb{E}}\{...\}$ é também especificada pelo elemento Loop, com um atributo adicional oper recebendo Min representando a operação de redução $\bigwedge$, que é a operação de mínimo:

Algoritmo 20 Primeiro algoritmo da erosão paralela

$\varepsilon^1 : K^{\mathbb{E}} \times \mathbb{Z}^B \longrightarrow K^{\mathbb{E}}$
$\quad \varepsilon^1(f, b) = g$

$\forall x \in \mathbb{E}$
$\quad g(x) = \bigwedge_{\forall y \in B_x} \{f(y) \dot{-} b(y - x)\};$

```
<Loop name="for" i="y" Dse="b" oper="Min">
```

O atributo Dse acima indica que o laço irá varrer o domínio da função estruturante b. O parâmetro $b(y - x)$ é obtido pelos elementos Var, Index e Oper. Neste último caso, dois atributos são passados, um indicando a operação de translação e o outro indicando o índice de translação:

```
<Oper name="Transl" i="x">
  <Var>b<Index>y</Index></Var>
</Oper>
```

$\diamond$

Exemplo A.8

Neste outro exemplo, apresentado no Algoritmo 21, será obtido a erosão da imagem f pela função estruturante b, fazendo a intersecção da translação de f por y, subtraindo de b em y (veja Figura 2.5).

Algoritmo 21 Segundo algoritmo da erosão paralela

$\varepsilon^2 : K^{\mathbb{E}} \times \mathbb{Z}^B \longrightarrow K^{\mathbb{E}}$
$\quad \varepsilon^2(f, b) = g$

$\forall x \in \mathbb{E}$
$\quad g = \bigcap_{\forall y \in B_x} \{f_y \dot{-} b(y)\};$

A principal diferença deste exemplo e do exemplo anterior, com relação ao código gerado, é com o parâmetro f_y, que tem como retorno uma imagem

transladada por y. Este tratamento diferenciado da operação de translação ocorre através do contexto onde o parâmetro f aparece no código XML.

$\diamond$

Exemplo A.9

Neste último exemplo, apresentado no Algoritmo 22, será obtido a erosão de cada pixel x da imagem g fazendo a redução, através da operação de mínimo de todas as sub-imagens de f subtraídas de b (veja também Figura 2.6).

Algoritmo 22 Terceiro algoritmo da erosão paralela

$\varepsilon^3 : K^{\mathbb{E}} \times \mathbb{Z}^B \longrightarrow K^{\mathbb{E}}$
$\quad \varepsilon^3(f, b) = g$

$\forall x \in \mathbb{E}$
$\quad g = \bigwedge\{f(x)_{y,\ \forall y \in B} \dot{-} b\};$

Note que neste exemplo $f(x)_{y,\ \forall y \in B}$ retorna uma sub-imagem de f com centro no pixel x tendo o mesmo domínio de b.

$\diamond$

A.8 Conclusões e trabalhos futuros

Foi apresentado neste apêndice *mmil*, *mathematical morphology independent language*, um ambiente que escreve transformações morfológicas em *XML* e, ao ser compilado, gera de forma automática código em diversas linguagens de programação e também documentos. Além disso, foi visto que ao gerar documentos em *LaTeX*, é possível não mais trabalhar com linguagens de programação para descrever uma transformação morfológica, mas com expressões matemáticas, e a compilação destas expressões (armazenadas em *XML*) são códigos executáveis.

Trabalhos futuros

Para concluir este ambiente está faltando validar os códigos gerados para a linguagem de programação C, corrigir eventuais erros de geração de código, melhorar e completar a documentação do ambiente e a documentação gerada e finalmente testar a eficiência dos códigos gerados de forma automática. Por exemplo, é desejável saber quais das três operações de erosão implementadas na Seção A.7 é a mais eficiente na linguagem *C* ou *MATLAB*, rodando em um *PC* com *Windows 98* ou *LINUX*. Falta também fazer a otimização do código gerado.

Como resultado adicional deste ambiente, fazendo poucas modificações, é possível construir uma estrutura em *XML* contendo todas as informações necessárias de um artigo, a compilação é gerada num formato qualquer (como *pdf*) pronta para a submissão, e é possível também testar o pseudo-código (representados por expressões matemáticas) gerando códigos para a linguagem de programação desejada. Assim será eliminado o trabalho de implementar um pseudo-código contido em um artigo.

Referências Bibliográficas

[ASU95] A.V. Aho, R. Sethi e J.D. Ullman. *Compiladores: Princípios, Técnicas e Ferramentas*. LTC, Rio de Janeiro, Brasil, 1995.

[Aus99] M.H. Austern. *Generic Programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Publishing Company, London, 1999.

[BB92] J. Barrera e G.J.F. Banon. Expressiveness of the morphological language. No *Image Algebra and Morphological Image Processing III*, páginas 264–274, San Diego, California, 1992. SPIE.

[BB94] G.J.F. Banon e J. Barrera. *Bases da morfologia matemática para analise de imagens binárias*. IX Escola de Computação, Recife, Brasil, 1994.

[BBL94] J. Barrera, G.F. Banon e R.A. Lotufo. A mathematical morphology toolbox for the KHOROS system. No *Image Algebra and Morphological Image Processing V*, páginas 241–252, Bellingham, Julho 1994. SPIE.

[BC98] J. Bowen e D. Chippington. Z on the web using java. *Proc. 11th Int. Conf. on the Z Formal Method (ZUM)*, 1493:66–80, Setembro 1998.

[BHJ97] J. Barrera e R. Hirata Jr. Fast algorithms to compute the elementary operators of mathematical morphology. No *Brazilian Symposium on Computer Graphics and Image Processing*, páginas 163–170, São Paulo, Brasil, Outubro 1997.

[CVM99] P. Ciancarini, F. Vitali e C. Mascolo. Managing complex documents over the www: a case study for XML. Relatório Técnico UBLCS-99-06, Department of Computer Science, Mura Anteo Zamboni, 7, Italy, 1999.

[DCB+02] M.C. D'Ornellas, M. Carrard, T. Baldissera, G. Peccini e R. Disconzi. Programação genérica com C++ e STL. Relatório Técnico, Universidade Federal de Santa Maria, Santa Maria, RS, Brasil, 2002.

[D'O01] M.C. D'Ornellas. *Algorithmic Patterns for Morphological Image Processing*. Tese de Doutorado, University of Amsterdam, Amsterdam, 2001.

[DGDL02] J. Darbon, T. Géraud e A. Duret-Lutz. Generic implementation of morphological image operators. *ISMM*, 2002.

[GW92] R.C. Gonzalez e R.E. Woods. *Digital Image Processing*. Addison-Wesley Publishing Company, 1992.

[Har96] A. Harry. *Formal Methods - Fact File: VDM and Z*. John Wiley and Son Ltd, 1996.

[Hei91] H.J.A.M. Heijmans. Theoretical aspects of gray-level morphology. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):568–581, Junho 1991.

[HWIC89] L.G.C. Hamey, J.A. Webb e Wu I-Chen. An architecture independent programming language for low-level vision. *Computer Vision, Graphics and Image Processing*, 48(2):246–264, Junho 1989.

[KPS00] D. Koelma, E. Poll e F. Seinstra. Horus release 0.9.2. Relatório Técnico, University of Amsterdam, Amsterdam, 2000.

[LZHJB97] R.A. Lotufo, F.A. Zampirolli, R. Hirata Jr. e J Barrera. MMachLib functions and MMach operators. *Brazilian Workshop on Mathematical Morphology*, Fevereiro 1997.

[Mac02] R.C. Machado. Adesso - ambiente para desenvolvimento de software científico. Dissertação de Mestrado, Universidade Estadual de Campinas - UNICAMP, Campinas, SP, Brasil, 2002.

[McG99] S. McGrath. *XML: Aplicações Práticas*. Campus, Rio de Janeiro, 1999.

[Mik95] L. Mikusiak. Z browser: A tool for visualization of z specifications. *Proc. 9th Int. Conf. on the Z Formal Specification Notation (ZUM)*, 967:510–525, Setembro 1995.

[Pre02] R.S. Pressman. *Engenharia de Software*. McGraw-Hill, Rio de Janeiro, 5ed. edition, 2002.

[Spi88] J.M. Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, 1988.

[Vin92] L. Vincent. Morphological algorithms. No E. R. Dougherty, editor, *Mathematical Morphology in Image Processing*, capítulo 8, páginas 255–288. Marcel Dekker, New York, 1992.

[Web90] J.A. Webb. Architecture-independent global image processing. No *10th International Conference on Pattern Recognition*, volume 2, páginas 623–628, Atlantic City, NJ, USA, Junho 1990.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[WWIC89] R.S. Walace, J.A. Webb e Wu I-Chen. Machine-independent image processing: performance of apply on diverse architectures. *Computer Vision, Graphics and Image Processing*, 48(2):265–276, Junho 1989.

[ZL00] F.A. Zampirolli e R.A. Lotufo. Classification of the distance transformation algorithms under the mathematical morphology approach. No *Brazilian Symposium on Computer Graphics and Image Processing*, Gramado, RS, Brasil, Outubro 2000.