Algoritmos Morfológicos escritos em XML

Francisco de Assis Zampirolli[1] Roberto de Alencar Lotufo[2]
Rubens Campos Machado[3]

[1] Centro Universitário Senac
Av. Eng. Eusébio Stevaux, 823 – 04696–000 – São Paulo, SP
[2] FEEC – Faculdade de Elétrica e de Computação
6101 – 13083–970 Campinas, SP
[3] CenPRA - Centro de Pesquisa Renato Archer
6162 - 13081-970 - Campinas, SP, Brasil

Resumo Este artigo apresenta a *mmil* (*mathematical morphology intermediary language*), uma forma de representar algoritmos morfológicos usando XML, aplicados em processamento de imagens. Isto foi possível através do uso de um subconjunto da linguagem formal Z. Desta forma, algoritmos morfológicos clássicos foram escritos na *mmil* e, como exemplo, executados em linguagens C e documentados em LaTeX.

1 Introdução

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 reescrita de código. Existe hardware específico para o processamento de imagens e para a morfologia matemática [3], hardware aceleradores nas *CPU's* convencionais 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.

Este trabalho contribui nesta questão, criando uma *linguagem intermediária* para escrever algoritmos morfológicos, e como resultado ter geração automática de código em várias linguagens de programação, junto com suas documentações. Exemplos de algoritmos morfológicos podem ser para implementar dilatação, erosão e transformada de distância [23].

Existem propostas de linguagens de programação independente de arquitetura para processamento de imagens [9, 22, 21], mas nenhuma delas usa XML. Recentemente surgiram normas para diminuir o trabalho de escrita de código, como MDA, OCL e

XMI, definidas pela OMG (*Object Management Group*) [17]. Por exemplo, no MDA (*Model Driven Architecture*) existem modelos independentes de plataforma PIM (*Plataform Indepentent Model*) e modelos específicos de plataforma PSM (*Platform Specific Model*). Podemos dizer que este trabalho define um PIM em XML e as *folhas de estilo* utilizadas poderiam ser consideradas como PSM.

Após esta introdução, a próxima seção descreve sobre os conceitos básicos utilizados. A seção 3 apresenta o ambiente *mmil*. A seção 4 apresenta os nove elementos da *linguagem intermediária*, junto com exemplos em XML. Finalmente, a seção 5 apresentada a conclusão e trabalhos futuros.

2 Conceitos

2.1 Organização da informação

Uma boa metodologia no desenvolvimento de software é definir uma estrutura de armazenamento que possibilite processar os seguintes conceitos [15]:

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 criação de documentos conhecido como LaTeX [13]. 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* [12]. 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.

Nos últimos anos, 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.

2.2 Notação Z

A notação Z é utilizada para especificação formal de problemas e é baseada em teoria dos conjuntos e lógica de primeira ordem. Z foi desenvolvida pelo *Programming Research Group* no *Oxford University Computing Laboratory* na década de 70 e é de domínio público pelo padrão *ISO/IEC 13568:2002* [1].

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* [8], que transforma especificações Z escritas em LaTeX em documentos correspondentes em HTML [8] e isto também pode ser realizado no ambiente proposto neste documento através das *folhas de estilo*. Existem também editores para a *notação Z*, que podem armazenar seus documentos em LaTeX, como o *ZCREATOR* [7] e visualizadores, como o *Z Browser* [16].

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 de algoritmos morfológicos. Neste editor, além de poder visualizar as expressões matemáticas, é possível gerar códigos LaTeX, *HTML, C, MATLAB*, entre outros. Este editor seria uma proposta de continuação deste trabalho e não será discutido neste texto, porém existem trabalhos recentes definindo editores que poderiam ser usados, como o XESB [18].

2.3 Morfologia matemática

Uma forma elegante de resolver problemas de processamento de imagens é através da utilização de uma base teórica consistente. Uma destas teorias é a *morfologia matemática* criada na década de 60 por Jean Serra e George Matheron na *École Nationale Superiéure des Mines de Paris*, em Fontainebleau, França. Esta teoria diz que é possível fazer transformações entre reticulados completos[4], os quais são chamados de *operadores morfológicos*. Na morfologia matemática existem quatro classes básicas de operadores: dilatação, erosão, anti-dilatação e anti-erosão, chamadas de *operadores elementares*operador! elementar. Banon e Barrera [2] provaram que todos os operadores morfológicos invariantes por translação podem ser obtidos a partir de combinações de operadores elementares juntamente com as operações de união e intersecção. Além disso, quando um reticulado possui uma *família sup-geradora*, estes operadores podem ser caracterizados por *funções estruturantes*. Usando estes operadores elementares é possível construir uma *linguagem formal*, a *linguagem morfológica*, e sua implementação é chamada *máquina morfológica* [5]. Um exemplo de uma máquina morfológica é a *MMach* [4].

Veja na Tabela 1 a gramática da linguagem morfológica definida por Banon e Barrera [3]. Esta linguagem tem como característica representar operadores como dilatação e 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.

Como exemplo do uso da gramática definida na Tabela 1, veja a seguir a definição da erosão morfológica caracterizada por função estruturante:

Seja $\mathbf{Z}$ o conjunto dos inteiros, $\mathbf{E} \subset \mathbf{Z}^2$ o domínio da imagem e $K = [0, k] \subset \mathbf{Z}$ um intervalo de números inteiros representando os possíveis níveis de cinza da imagem. O operador invariante por translação em níveis de cinza, $\varepsilon_b : K^{\mathbf{E}} \rightarrow K^{\mathbf{E}}$ ($K^{\mathbf{E}}$, lê-se conjuntos de funções de $\mathbf{E}$ em K), é definido como [11]:

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

[4] Um conjunto qualquer com uma relação de ordem é um reticulado completo se todo subconjunto não vazio tem um supremo e um ínfimo. Para detalhes da teoria dos reticulados veja [6].

<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>::= $\vee \mid \wedge$
<operador morfológico>::= $\varepsilon \mid \delta \mid \varepsilon^a \mid \delta^a$
<letra>::= $a \mid b \mid c \mid d$
<dígito>::= $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Tabela 1. Gramática da linguagem morfológica [3].

onde $f \in K^{\mathbf{E}}$, $x \in \mathbf{E}$, $B \subseteq \mathbf{E} \oplus \mathbf{E}$ 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 $b : B \to \mathbf{Z}$.

A teoria dos reticulados estudada em morfologia matemática é abrangente e o leitor interessado pode consultar, por exemplo, os trabalhos de Serra, Banon e Barrera [19, 3].

3 Ambiente *mmil*

O ambiente *mmil* foi o resultado de pesquisas em padrões de algoritmos gerando a tese do autor Zampirolli [23]. Nesta tese as erosões foram implementadas nos padrões de varredura paralelo, seqüencial e por propagação. Este processo também pode ser feito para outros operadores morfológicos, como dilatação, reconstrução e transformada de distância. Além disso, a transformada distância pode ser equivalente a erosão. Assim, os algoritmos clássicos da transformada de distância usando erosões foram reescritos(reimplementados) e classificados através destes padrões, produzindo códigos mais simples e eficientes. Deste estudo surgiram os elementos da linguagem intermediária, definidos na próxima seção.

3.1 Arquitetura

A arquitetura do ambiente *mmil* é ilustrada na Figura 1. Esta figura mostra um editor *GUI* (*Graphic User Interface*) para editar documentos *XML*, como exemplo, XESB [18]. 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 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 documentação (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.

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

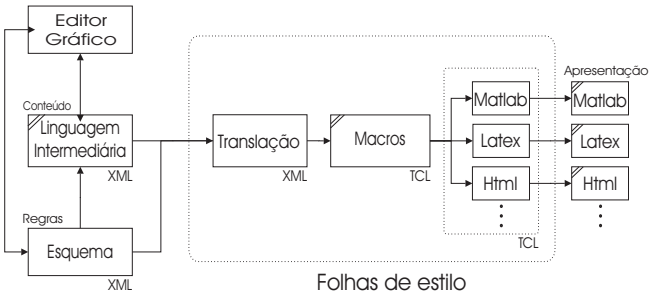

Figura 1. Arquitetura da *mmil*.

A arquitetura da *mmil* foi desenvolvida no *Adesso*, um ambiente computacional de suporte ao desenvolvimento de aplicações científicas [14].

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

3.2 Linguagem intermediária

Usualmente os operadores 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*, inspirado na *notação Z* [20] e na *linguagem morfológica* [3]. Esta *linguagem intermediária* permite tradução para diversas linguagens e arquiteturas.

Por exemplo, existem várias formas de implementar um operador morfológico. A Figura 2 ilustra três algoritmos para o operador 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* em XML. 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 cartões aceleradores de processamento de imagens.

4 Elementos da linguagem intermediária

A *notação Z* pode ser armazenada numa linguagem intermediária chamada *linguagem intercâmbio* (*interchange language*) [10] usando elementos (*tags*) para armazenar as informações de forma semelhante à linguagem *XML*. Analogamente, a linguagem *XML*

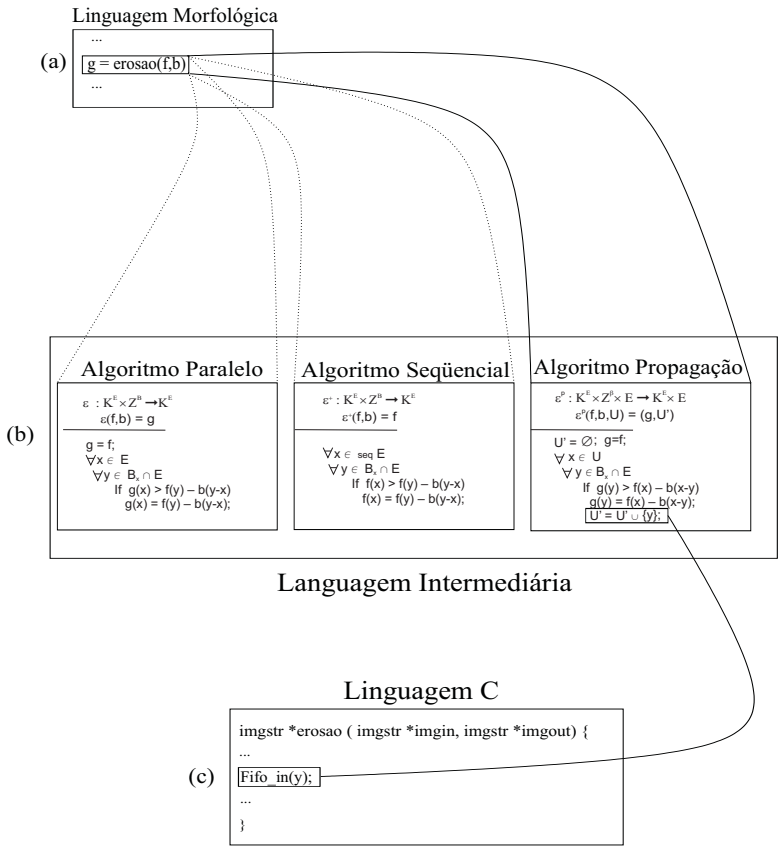

Figura 2. Ilustração da relação entre: (a) *linguagem morfológica*, (b) *linguagem intermediária* e (c) *linguagem C*.

é utilizada para armazenar a linguagem intermediária definida neste trabalho 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 [14].

O elemento *AdFunction* define as transformações implementadas, Figura 3 [5], 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; *Args* – define os argumentos de entrada através dos filhos definidos pelo elemento *Arg* contendo os atributos *name* e *type*; e *Source* – descreve o código da função.

A *linguagem intermediária*, onde são implementados os operadores morfológicos,

[5] Figura gerada pelo software *XMLSpy*.

é definida dentro do elemento *Code*, filho de *Source*, com atributo *lang* recebendo *"mmil"*, veja Figura 3. Será descrito abaixo cada um dos elementos desta linguagem.

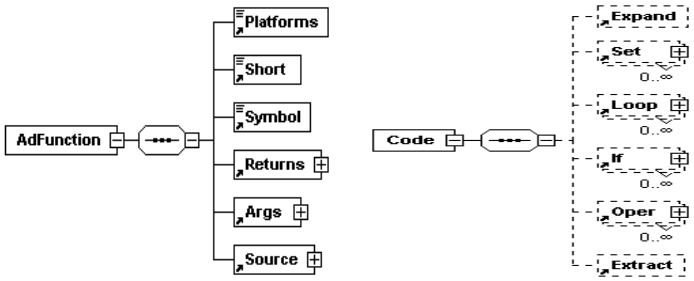

Figura 3. Elementos *AdFunction* e *Code* (filho de *Source*).

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*.

4.2 *Index*

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

Exemplo 1. 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>
```

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 4. Veja Exemplo 2.

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 4.

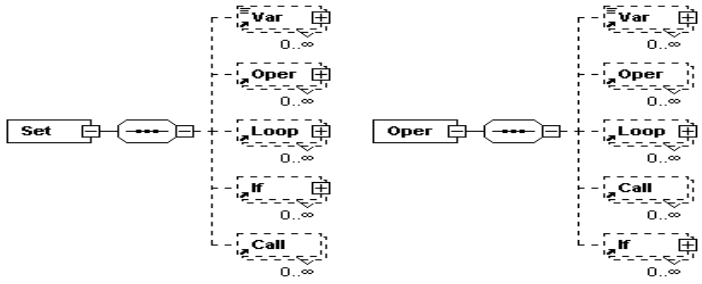

Figura 4. Elementos *Set* e *Oper*.

Exemplo 2. O exemplo abaixo faz a imagem g receber a adição da imagem $f1$ e $f2$. Estas três variáveis possuem o mesmo tamanho $\mathbf{E}$.

```
...
<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 matemática (ou em código em *MATLAB*):

$$g = f1 + f2$$

4.5 *Loop*

Este elemento faz referência a repetições *For* ou *While*, denotado por $\forall$. O laço é associado a um índice e a um domínio através de atributos. O laço *While* requer uma expressão lógica. Veja Figura 5.

Exemplo 3. Neste exemplo será mostrada uma outra versão para a adição de duas imagens. As três variáveis $f1$, $f2$ e g possuem o mesmo tamanho $\mathbf{E}$. Agora a varredura é explícita para o domínio $\mathbf{E}$:

```
...
<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:

$\forall x \in \mathbf{E}$
$g(x) = f1(x) + f2(x)$;

ou ainda, pelo seguinte código em *MATLAB*:

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

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

4.6 *If*

If é um comando condicional. Se a expressão lógica é *True*, o bloco associado ao elemento *If* é executado. Veja Figura 5.

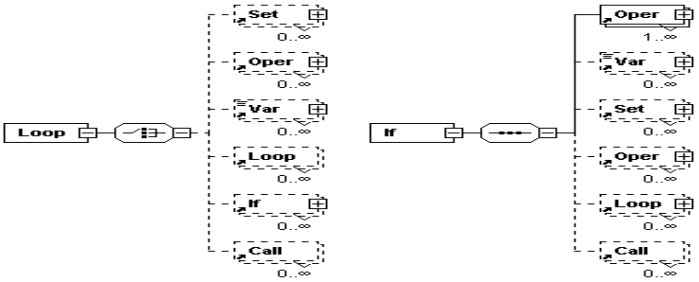

Figura 5. Elementos *Loop* e *If*.

4.7 *Call*

Call é usado para referenciar uma operação implementada pela *mmil*.

Exemplo 4.

```
<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.

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.

4.9 *Extract*

Extract extrai uma imagem pela vizinhança passada como parâmetro. Utilizado pelos operadores morfológicos que usam funções estruturantes.

4.10 Implementação da erosão

Será apresentada uma forma de implementar a erosão usando a linguagem intermediária. A parte central deste exemplo é mostrar a flexibilidade do ambiente desenvolvido, particularmente do elemento *Loop*. Não serão mostrados os códigos *XML* e *MATLAB*, mas apenas as expressões matemáticas equivalentes, que podem ser geradas automaticamente.

A erosão, apresentada na seção 2.3, quando implementada na *mmil* tem a representação apresentada no Algoritmo 1 (veja Figura 6) [6].

Algoritmo 1 Primeiro algoritmo da erosão paralela

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

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

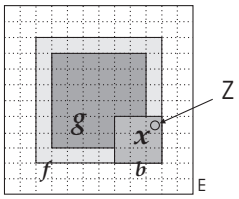

Figura 6. Ilustração do Algoritmo de Erosão.

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

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

[6] Para detalhes da notação de algoritmos utilizado neste trabalho consulte Zampirolli [23] ou Seção 2.5 de `www.zamp.pro.br/pub/tese.pdf`.

O atributo i especifica um elemento do domínio de f, isto é, um elemento de $\mathbf{E}$. A expressão $\bigwedge_{\forall y \in B_x \cap \mathbf{E}}\{...\}$ 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:

```
<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>
```

5 Conclusões e trabalhos futuros

Foi apresentado neste trabalho *mmil* (*mathematical morphology intermediary language*), um ambiente que escreve algoritmos (operadores) morfológicos em *XML* e, ao ser compilado, gera de forma automática código em diversas linguagens de programação e também documentação. Além disso, foi visto que ao gerar documentação em LaTeX, é possível não mais trabalhar com linguagens de programação para descrever um algoritmo morfológico, 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. 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

1. ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.
2. G.J.F. Banon and J. Barrera. Decomposition of mappings between complete lattices by mathematical morphology, Part I: general lattices. *Signal Processing*, 30:299–327, 1993.

3. G.J.F. Banon and J. Barrera. *Bases da morfologia matemática para analise de imagens binárias*. IX Escola de Computação, Recife, Brasil, 1994.
4. J. Barrera, G.F. Banon, and R.A. Lotufo. A mathematical morphology toolbox for the KHOROS system. In *Image Algebra and Morphological Image Processing V*, pages 241–252, Bellingham, Julho 1994. SPIE.
5. J. Barrera and G.J.F. Banon. Expressiveness of the morphological language. In *Image Algebra and Morphological Image Processing III*, pages 264–274, San Diego, California, 1992. SPIE.
6. G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, Rhode Island, 1967.
7. J. Bowen and D. Chippington. Z on the web using java. *Proc. 11th Int. Conf. on the Z Formal Method (ZUM)*, 1493:66–80, Setembro 1998.
8. P. Ciancarini, F. Vitali, and C. Mascolo. Managing complex documents over the www: a case study for XML. Technical report UBLCS-99-06, Department of Computer Science, Mura Anteo Zamboni, 7, Italy, 1999.
9. L.G.C. Hamey, J.A. Webb, and Wu I-Chen. An architecture independent programming language for low-level vision. *Computer Vision, Graphics and Image Processing*, 48(2):246–264, Junho 1989.
10. A. Harry. *Formal Methods - Fact File: VDM and Z*. John Wiley and Son Ltd, 1996.
11. 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.
12. Donald E. Knuth. *The TEX Book*. Addison-Wesley Publishing Company, Reading, MA, 15th edition, 1989.
13. Leslie Lamport. *LATEX: A Document Preparation System*. Addison-Wesley Publishing Company, Reading, MA, 1986 (also see 2nd edition, 1994).
14. R.C. Machado. Adesso - ambiente para desenvolvimento de software científico. Dissertação de Mestrado, Universidade Estadual de Campinas - UNICAMP, Campinas, SP, Brasil, 2002.
15. S. McGrath. *XML: Aplicações Práticas*. Campus, Rio de Janeiro, 1999.
16. 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.
17. OMG – Object Management Group, Acesso em 2005. www.omg.org.
18. R. Queirós and J. P. Leal. Xesb (xml editor schema based) um editor para as massas. In João Correia Lopes José Carlos Ramalho, Alberto Simões, editor, *XATA2005, XML: Aplicações e Tecnologias Associadas (Vila Verde, Braga, 10 e 11 de Fevereiro de 2005)*, Portugal, Fevereiro 2005. ISMM, Universidade do Minho. http://hdl.handle.net/1822/865.
19. J. Serra, editor. *Image Analysis and Mathematical Morphology - Volume II: Theoretical Advances*. Academic Press, London, 1988.
20. J.M. Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, 1988.
21. R.S. Walace, J.A. Webb, and 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.
22. J.A. Webb. Architecture-independent global image processing. In *10th International Conference on Pattern Recognition*, volume 2, pages 623–628, Atlantic City, NJ, USA, Junho 1990.
23. F.A. Zampirolli. *Transformada de distância por morfologia matemática*. Doutor, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, Campinas, SP, Brasil, 2003.

This article was processed using the LATEX macro package with LLNCS style