Geração de Sistemas de Transformação

Adeline de Sousa[1 2] **, Paulo Borba**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco
Av. Prof. Luís Freire, s/n – 50740-540 Recife, PE

[2]Qualiti Software Processes
Av. Marquês de Olinda, n°126 – 50030-901 Recife, PE

{adss, phmb}@cin.ufpe.br

Abstract. *As systems become more complex, the need for transformation systems for multiple languages increases. Unfortunately, current transformation systems are not adequate to deal with complex transformations, and to build language-specific systems is too expensive. This paper describes an approach to rapidly generate language-specific transformation systems, using JaTS transformations. The systems generated by this approach are based on JaTS: they use a language for specifying transformations with the same constructors for meta-programming introduced by JaTS, present the same architecture, and share auxiliary components.*

Resumo. *À medida que os sistemas se tornam mais complexos, cresce a necessidade de sistemas de transformação para múltiplas linguagens. Infelizmente, os sistemas de transformação atuais não são adequados para manipular transformações complexas e construir um novo sistema para cada nova linguagem é caro. Este artigo descreve uma abordagem para geração de sistemas de transformação usando transformações de JaTS. Os sistemas gerados por esta abordagem são baseados em JaTS: baseiam-se em uma linguagem de templates similar à de JaTS, têm a mesma arquitetura e compartilham os mesmos componentes auxiliares.*

1. Introdução

À medida que os sistemas se tornam mais complexos, cresce a necessidade de desenvolver componentes em mais de uma linguagem. O desenvolvimento de um sistema Web, por exemplo, envolve pelo menos três linguagens: a linguagem de desenvolvimento do núcleo do sistema (Java [Gosling et al., 1996]), a linguagem de apresentação (JSP [Hall, 2000]+ HTML [htm, 2005]) e a linguagem de configuração (XML [Harold and Means, 2001]).

Portanto, cresce também a necessidade de ferramentas que gerem e mantenham códigos escritos em várias linguagens. Sem isso, a produtividade dos desenvolvedores pode diminuir, tornando alto o custo de se fabricar e manter softwares complexos. Duas alternativas podem diminuir este custo: usar uma ferramenta de transformação de programas independente de linguagem ou usar várias ferramentas de transformação de programas, uma para cada linguagem usada na implementação desses sistemas.

As ferramentas independentes de linguagem têm a vantagem de transformar vários tipos de linguagens. Mas como são puramente sintáticas, não conseguem exprimir certas transformações e também tornam mais difícil exprimir transformações complexas. As ferramentas específicas para uma determinada linguagem, por outro lado, podem realizar

transformações baseadas em semântica e, com maior concisão, expressar transformações elaboradas.

Assim, para evoluir sistemas complexos, o ideal seria dispor de um conjunto de ferramentas específicas para cada linguagem de que o sistema faz uso. A construção de ferramentas deste tipo, entretanto, é bastante custosa. Este trabalho apresenta uma abordagem de programação gerativa [Czarnecki and Eisenecker, 2000] que tenta unir as vantagens dos dois tipos de ferramentas supracitadas, ao permitir a geração de sistemas de transformação específicos a um baixo custo. Tais sistemas formam uma linha de produtos de sistemas de transformação baseados em JaTS [Castor et al., 2001]: possuem uma linguagem de templates com as mesmas construções de metaprogramação, têm a mesma arquitetura e compartilham classes com o JaTS.

Este artigo está definido da seguinte maneira: primeiramente, fornecemos uma visão geral de JaTS na Seção 2, em seguida discutimos como a geração de sistemas é feita na Seção 3. Na Seção 4 apresentamos nossos resultados e , na Seção 5, os trabalhos relacionados. Por fim, apresentamos nossas conclusões na Seção 6.

2. Visão Geral de JaTS

Uma vez que os sistemas gerados nesta abordagem são baseados em JaTS, iremos fornecer uma visão geral da linguagem e da arquitetura desse sistema, a fim de permitir um melhor entendimento da abordagem. Maiores detalhes sobre esse sistema, estão descritos na literatura [Castor and Borba, 2001, Castor, 2001]. É importante ressaltar que o foco deste artigo é o uso de JaTS para gerar novos sistemas de transformação e não em possíveis extensões da linguagem.

Como JaTS serve para denominar tanto a linguagem de templates quanto o sistema de transformação de programas Java, iremos usar a notação JaTS-TL quando nos referirmos à linguagem de templates, a fim de evitar confusão. Em JaTS, uma transformação é composta por três elementos: um conjunto de templates do lado esquerdo, uma pré-condição e um conjunto de templates do lado direito. Por simplicidade, vamos assumir que cada um desses conjuntos contém um único elemento. O template do lado esquerdo tem a mesma estrutura sintática do programa ao qual a transformação será aplicada e o template direito, a estrutura do programa que será gerado pela transformação. A pré-condição deve ser verdadeira para que a transformação aconteça.

Primeiramente, uma visão geral da linguagem de templates é fornecida. Em seguida, descrevemos a arquitetura da linha de produtos de sistemas de transformação de programas.

2.1. A Linguagem de Templates

A linguagem para descrever templates possui construções que permitem manipular o programa a ser transformado. Estas construções são usadas no casamento de padrões e na transformação. A seguir, apresentamos as principais construções da linguagem de templates JaTS-TL, com foco naquelas usadas neste trabalho.

Variáveis são a estrutura mais simples de JaTS-TL e servem para armazenar valores do programa sendo transformado. Os tipos de variáveis JaTS correspondem às estruturas sintáticas de Java, como Identificador, Nome, Atributo. Entretanto, existem variáveis que agrupam conjunto de declarações, como o conjunto de declarações de atributo de uma determinada classe.

Outra construção de JaTS-TL é a cláusula opcional, que permite determinar em que trechos de um template o casamento é opcional. Tal construção somente é válida para

as construções da gramática de Java que também são opcionais. Isto permite especificar transformações mais genéricas, abstraindo-se de detalhes irrelevantes ao contexto da transformação. No exemplo a seguir, vemos o uso de variáveis (`#C` e `#SC`) para capturar o nome da classe e o da superclasse e também o uso da cláusula opcional (`#[ ... ]#`), pois a superclasse pode não estar explícita.

```
class #C #[ extends #SC ]# {}
```

As construções de transformação são as declarações executáveis, estruturas que devem ser executadas a fim de se produzir um programa válido em Java. Elas são classificadas em expressões executáveis, declarações condicionais e declarações iterativas.

As expressões executáveis permitem extrair ou modificar informações a partir de algumas estruturas sintáticas. Por exemplo, permitem saber se um determinado atributo é público ou, até mesmo, mudar a visibilidade deste atributo, entre outras operações. O exemplo a seguir mostra como um novo nó sintático pode ser formado (declaração de método), utilizando-se informações de outros nós (declaração de atributo `#att`) através de expressões executáveis(`#< ...>#`).

```
public #<#att.getType()># #<#att.addSuffix("get")># () {
    return #<#att.getName()>#;
}
```

Declarações condicionais funcionam como estruturas *if-then-else*. Elas permitem que a transformação de código seja condicionada à satisfação de uma determinada condição. As declarações iterativas permitem iterar sobre algumas estruturas do programa fonte, gerando como resultado um conjunto de estruturas que obedece a um determinado padrão. O exemplo a seguir ilustra o uso de declaração iterativa (`forall ...`) combinada com o uso de declaração condicional (`#if(...){...}`), para gerar declarações de métodos de acesso para cada atributo de uma classe Java.

```
// para cada atributo presente no conjunto #attributes
forall #att in #attributes {
   // se este atributo for private, gere o método get correspondente
   #if( #att.isPrivate()) {
       public #<#att.getType()># #<#att.addSuffix("get")># () {
           return #<#att.getName()>#;
       }
  }
}
```

2.2. Arquitetura do Sistema JaTS

O sistema é composto de dois módulos: um para as atividades relativas a *parsing* e *pretty-printing* (`IOModule`) e outro para a etapa de transformação (`TransformModule`). A Figura 1 ilustra como estes módulos estão dispostos. Eles são ligados através da interface INode, que é implementada por todos os nós da árvore sintática, provendo os serviços que permitem a transformação.

A Figura 2 mostra os métodos contidos na interface. É importante notar que o módulo de transformação apenas delega para o nó sintático a responsabilidade de fazer a transformação propriamente dita, através de invocações aos métodos *match*, *accept* e *process*, presentes em cada nó da árvore.

Dois padrões de projeto [Gamma et al., 1994] são utilizados na implementação de JaTS: *Interpreter*, para as operações de casamento, processamento e *Visitor* para as operações de impressão da árvore e substituição de valores. E estes padrões estão presentes nas implementações dos sistemas gerados nesta abordagem.

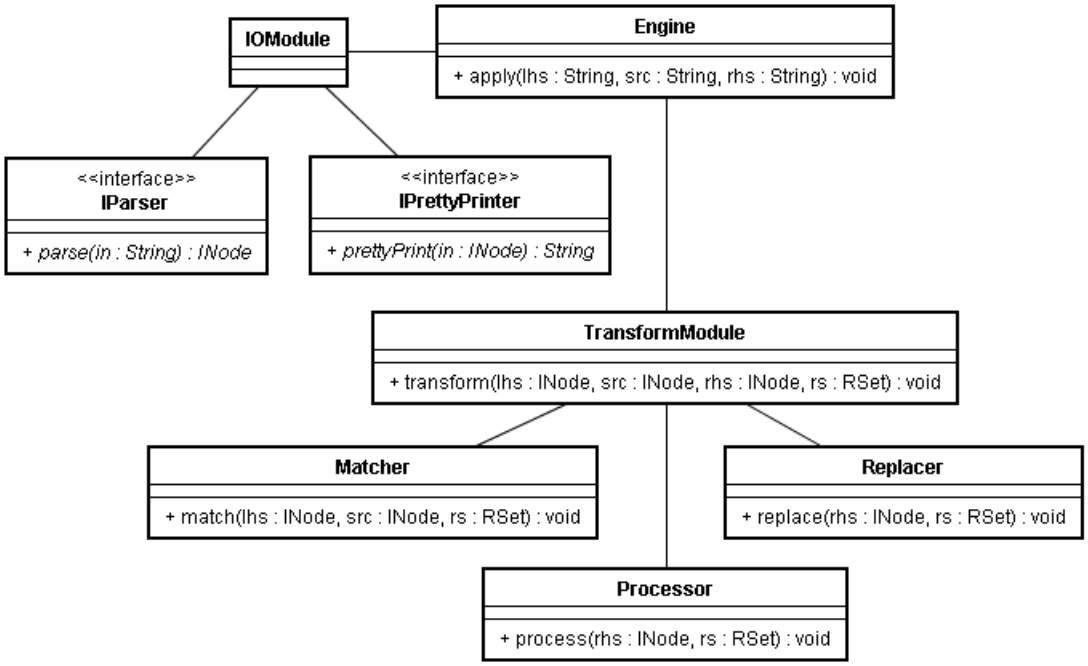

Figura 1: Arquitetura de JaTS

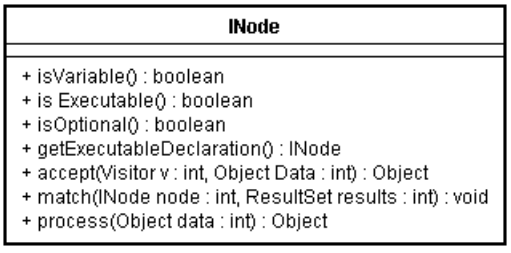

Figura 2: A Interface INode

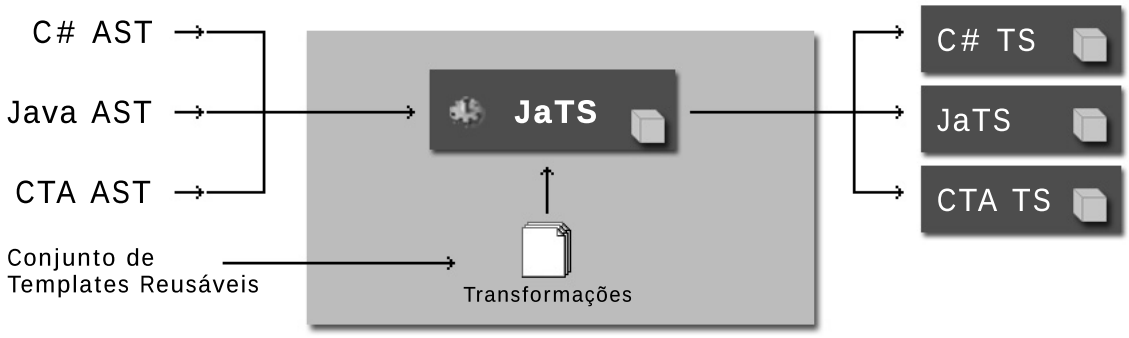

Figura 3: Funcionamento da Geração de Sistemas de Transformação

3. Geração de Sistemas de Transformação

O custo de geração de um sistema de transformação baseado em regras de reescrita é bastante alto. É preciso criar um mecanismo para permitir exprimir padrões de casamento e regras de reescrita, criar o engenho de transformação, etc. Nossa abordagem diminui este custo, usando programação gerativa para gerar automaticamente boa parte do código necessário para um sistema de transformação específico para uma dada linguagem. Em especial, geramos o mecanismo de execução da transformação, assim como a implementação, na árvore sintática, das operações relativas ao casamento de padrões e de regras de reescrita.

A Figura 3 mostra como o método de geração de sistemas funciona: A árvore sintática da linguagem-objeto[1] (por exemplo, C# [Liberty, 2003]) é fornecida como entrada. Então, o mecanismo gera grande parte do sistema de transformação para esta linguagem (C#TS), a partir de transformações JaTS. Algumas partes da implementação de JaTS são reusadas.

A seguir, mostramos os detalhes do funcionamento desta abordagem. Quatro passos são necessários para a geração destes sistemas: primeiro, a linguagem-objeto é estendida, em seguida, usando-se transformações JaTS, as árvores sintáticas dessas linguagens são adaptadas e os *visitors* são criados. Por fim, os componentes auxiliares, que são compartilhados por todos os sistemas são adicionados. Estes passos são detalhados nas seções seguintes.

3.1. Da Linguagem-objeto à Linguagem de Templates

Os sistemas gerados nessa abordagem são baseados em templates, cuja sintaxe tenta ser o mais próxima possível da sintaxe da linguagem-objeto. Isto é feito para diminuir a distância conceitual entre a linguagem-objeto e a linguagem de codificação das transformações.

Portanto, a primeira etapa na geração de um novo sistema é a extensão sintatica da linguagem-objeto, transformando-a em uma linguagem de templates. Esta etapa é feita manualmente e seu custo é proporcional ao número de produções (tamanho da BNF) da linguagem. Esta extensão é feita em quatro passos:

1. Primeiro, adicionamos palavras-chave correspondentes às estruturas sintáticas da linguagem-objeto. Estas palavras representam os tipos da linguagem de templates.
2. Em seguida, alteramos as produções da gramática da linguagem-objeto, para permitir que se comportem como variáveis ou como expressões executáveis.

[1]Neste artigo, o termo linguagem-objeto serve para denotar a linguagem em que são codificados os programas transformados pelo sistema de transformação.

Por exemplo, para a produção FieldDeclaration da gramática de C#, criamos a produção FieldDeclaration', onde:

```
FieldDeclaration' := FieldDeclaration  |
                     FieldDeclarationExecutable |
                     FieldDeclarationVariable
FieldDeclarationVariable := JType ":" VariableId;
FieldDeclarationExecutable := JType ":" ExecutableDeclaration
```

Onde `JType` é a produção que representa os tipos da linguagem de templates, que correspondem às palavras reservadas adicionadas no passo 1. E `ExecutableDeclaration` é a produção que corresponde às expressões executáveis, mostradas na Seção 2.

3. Agora, para cada elemento que pode ser opcional na linguagem-objeto, como a declaração de superclasse em Java, são feitas alterações para a adição de cláusulas opcionais da linguagem de templates.
4. Por fim, adicionamos as produções para iteração e para transformação condicional (declarações condicionais e declarações iterativas).

Quando aplicável, pode-se adicionar construções de metaprogramação que permitem realizar o casamento de conjunto de declarações. Em JaTS-TL, temos a produção `FieldDeclarationSet`, por exemplo.

Esta extensão é a base para termos um casamento de padrões que utiliza a semântica da linguagem. Por exemplo, em JaTS-TL, isto permite ter um casamento de padrões que não apenas detecta o padrão sintático:

```
[modificador] tipo identificador ["=" inicializador] ";"
```

E casa cada um dos elementos individualmente, mas que também percebe que este padrão é uma declaração de atributo e consegue casá-lo com uma única variável, por exemplo:

```
FieldDeclaration:#fd;
```

Parte destes passos pode ser automatizada, como em MetaJ [Oliveira, 2004]. Um módulo para esta finalidade está em construção.

3.2. Transformação da Árvore Sintática

Segundo foi visto na Seção 2, é com os nós que fica a maior parte da responsabilidade pela transformação. Os métodos responsáveis pelo casamento e processamento são implementados em cada nó, através do padrão *Interpreter*. O nó também tem o método para receber os *visitors* responsáveis pela impressão da árvore e pela substituição de valores. Estes métodos são gerados através de transformações JaTS.

Para isto, primeiramente, são capturadas as informações de cada nó da árvore sintática para poder alterá-lo. Interessa-nos, especialmente, o conjunto de declarações de atributo desta classe, as demais informações dos nós sintáticos, são apenas conservadas. O template [2] a seguir é o lado esquerdo da transformação e é usado para capturar as informações da classe de entrada:

```
ModifierList:#M class #TYPE #[ extends #TYPE_SC ]# {

    FieldDeclarationSet:#TYPE_ATTRS;
    ...
}
```

[2]Todos os templates e trechos de código mostrados aqui serão simplificados por questões de concisão.

Em seguida, usamos o template de geração para gerar os métodos relativos à transformação. Opcionalmente, podem ser geradas estruturas auxiliares, como métodos de acesso aos atributos, construtores, método *equals()*. O template a seguir mostra como o método para casamento é gerado. Templates similares são usados para gerar a implementação dos métodos *process* e *accept* da interface `INode`. A pré-condição para que esta transformação ocorra é que o conjunto de declarações de atributos presentes no nó sintático de entrada não seja vazio. Esta transformação é aplicada a todos os nós da árvore sintática.

```
ModifierList:#M class #TYPE #[ extends #TYPE_SC ]# implements INode {

    FieldDeclarationSet:#TYPE_ATTRS;
    ...
    public void match(INode node, RSet results){
        ...
        if (!this.matchesAsAVariable(node, results, INode.NO_TYPE)) {

            let #varName = #<#TYPE.toVariableName()>#;
                #attName = #<#A.getName()>#;

            in
            RSet rs = (RSet) results.clone();
            #TYPE  #varName = (#TYPE) node;

            forall #A #in #TYPE_ATTRS {
                this.#attName.match(#varName.#attName, results);
            }
            results.merge(rs);
        }
    }
}
```

Para ilustrar o que esta transformação faz, veremos um exemplo de classe de entrada e a saída produzida por esta transformação. A classe a seguir, faz parte da implementação da AST (Abstract Syntax Tree) de C# e representa uma declaração de método (contendo somente os atributos):

```
public class MethodDeclaration {

    private AttributeList attributes;
    private JModifierList modifiers;
    private JType returnType;
    private JName name;
    private JParameterList parameters;
    private JBlock body;
}
```

Após a aplicação da transformação supracitada, ela ficará assim:

```
public class MethodDeclaration implements INode {

    private JModifierList modifiers;

    private JBlock body;
    public void match(INode node, RSet results){
        if (!this.matchesAsAVariable(node,results,INode.NO_TYPE)) {
```

```
            MethodDeclaration metDec = (MethodDeclaration) node;
            this.attributes.match(metDec.getAttributes(), results);
            this.modifiers.match(metDec.getModifiers(), results);
            this.returnType.match(metDec.getResultType(), results);
            this.name.match(metDec.name, results);
            this.parameters.match(metDec.parameters, results);
            this.body.match(metDec.body, results);
        }
    }
```

Ou seja, para cada atributo presente na classe, foi gerada uma linha do método match.

3.3. Geração de Visitors

Para a geração dos *visitors*, usamos uma transformação composta por três templates: um para capturar informações do nó sintático para qual será gerado o método *visit*, um para capturar as informações do *visitor* que está sendo transformado e um template para a geração do método *visit* e conservação dos métodos previamente existentes. Os templates para capturar informação do nó sintático e do *visitor* são similares ao apresentado na seção anterior. A seguir mostramos o template de geração do *visitor*:

```
...
public class ReplacementVisitor #[extends AbstractVisitor]# {
    ...
    MethodDeclarationSet:#V_MDS;

    public Object visit(#TYPE node, Object data){

        forall #A #in #TYPE_ATTRS {
           let #tmpvar  = #<#A.addPrefix("tmp")>#;
               #varType = #<#A.getType()>#;

           in

           #varType #tmpVar = node.#<#vd.addPrefix("get")>#();
           if (#tmpVar.isVariable()) {
               node.#<#vd.addPrefix("set")># (
                  (#<#A.getType()>#)this.getValueOf(#tmpVar));
           }


        return node;
    }
}
```

De forma similar, existem templates para geração do método *visit* correspondente a cada nó dentro dos *visitors* de clonagem e de impressão. O *visitor* de impressão é o que necessita de maior intervenção do programador já que, a partir da sintaxe abstrata (representada pelas classes de entrada do sistema) não é possível obter a sintaxe concreta. Então, o programador precisa inserir o código que monta a `String` correspondente ao nó sintático.

3.4. Componentes Auxiliares

Na implementação de JaTS há vários componentes de apoio ao processo de transformação. A maioria deles, depende apenas da interface `INode`, assim, pode ser

reusada por todos os sistemas gerados utilizando-se esta abordagem. Classes auxiliares como `TransformHelper`, `ResultSet`, `ExecutableDeclaration` são reutilizadas na íntegra. Outras sofrem pequenas adaptações, como mudança de pacote e inclusão de interface.

4. Resultados

Esta abordagem de geração de novos sistemas de transformação foi testada em algumas oportunidades, revelando um desempenho satisfatório. Inicialmente, foi testada para geração de sistemas de transformação de linguagens orientadas a objeto, mas um teste para uma linguagem de especificação baseada em Circus [Sherif and He, 2002] está em andamento. Até o momento, o sistema gerado tem se mostrado adequado para as realizar as transformações de programas desta linguagem. As linguagens para as quais tanto a geração do sistema quanto o sistema de transformação gerado foram testados são: Java, C# e OO1 [oo1, 2005]. Esta última, uma linguagem bastante simples, desenvolvida em âmbito acadêmico apenas para ilustrar conceitos de orientação a objetos.

O primeiro teste foi o *bootstrapping* de JaTS, para permitir a evolução da linguagem de entrada, já que a linguagem Java sofreu algumas alterações desde que o sistema foi construído. Esse foi o teste que apresentou os melhores resultados, já que poucas alterações precisaram ser feitas manualmente: o código da árvore sintática foi gerado automaticamente, com um percentual de aproximadamente 92%. O código dos *visitors* também foi alterado automaticamente e pouco esforço foi necessário para a alteração do parser, pois somente foi necessário adicionar as novas construções. JaTS encontra-se atualmente em uso como engenho de geração de código Java da ferramenta QualitiCoder [cod, 2005].

O segundo teste foi a geração de um sistema de transformação para a linguagem C#. Como esta linguagem possui muitas similaridades sintáticas com Java, não foi criado um novo sistema, mas uma extensão de JaTS, para que este suportasse a linguagem C#. Portanto, o número de classes reusadas foi bem maior: todas as construções sintáticas em comum tiveram seu código reusado. A geração dos nós foi feita de forma automática, com um percentual de quase 90% e a transformação dos *visitors*, como em JaTS, foi totalmente automatizada. O esforço aqui residiu em transformar a linguagem C# em uma linguagem de templates. Novamente, as similaridades sintáticas ajudaram bastante e muito de JaTS pôde ser reusado. Assim como o JaTS, C#TS tem sido usado no QualitiCoder, porém, como engenho de geração de código C#.

O terceiro teste foi realizado com a linguagem OO1, uma linguagem bastante diferente de Java, com implementação da árvore sintática completamente diferente da implementação que usamos em JaTS. Neste, o esforço de modificação da árvore foi grande, pois alguns *refactorings* [Fowler et al., 1999] tiveram de ser realizados antes de se aplicar as transformações (por exemplo, alteração da hierarquia dos nós sintáticos). Entretanto, após os refactorings, o código necessário para o sistema de transformação foi 80% gerado automaticamente. Diferente dos testes anteriores, aqui, algumas classes tiveram de ser desenvolvidas, a fim de que as transformações pudessem ser utilizadas (por exemplo a classe que armazenava declarações de métodos).

A Tabela 1 traz um resumo destes números. Que são calculados dividindo-se o *LOC* (número de linhas de código) do sistema logo após a aplicação das transformações de geração pelo *LOC* do sistema pronto. O *LOC* anterior à geração não é considerado porque somente os atributos da classe é que são importantes na geração. Como é a geração que introduz os métodos necessários ao sistema de transformação, queremos mostrar o quanto a quantidade de código gerada representa no sistema de transformação final. Se

considerarmos o *LOC* inicial, este percentual praticamente não se altera, já que as linhas de código são mantidas na solução final. O número de linhas de código foi coletado usando a ferramenta JavaNCSS [jav, 2005].

Apesar de não terem sido obtidos através de experimentos formais [Travassos et al., 2003], estes números refletem o potencial de JaTS para gerar novos sistemas de transformação. Mesmo considerando-se margem de erro de 10%, percebe-se que a solução é adequada. O custo de extensão sintática da linguagem-objeto ainda existe e, por isso, um módulo para automatizar esta extensão está sendo desenvolvido.

Tabela 1: Resultado da Geração de Novos Sistemas de Transformação

Linguagem	Esforço de Adaptação do Parser	Geração de código AST	Geração de Código Visitors	Reuso de Componentes
Java	baixo	92.44%	97.04%	total
C#	médio	89.61%	95.27%	total
OO1	alto	80.17%	97.25%	com alterações

5. Trabalhos relacionados

Existem algumas outras abordagens para permitir que sistemas de transformação possam transformar diferentes linguagens de programação. Em geral, estas ferramentas são dotadas de mecanismos que permitem uma certa independência de linguagem-objeto. Para conseguir isso, este mecanismos, via de regra, perdem expressividade e só permitem regras de reescrita particularmente simples. Em nossa abordagem, pretendemos não perder o poder de expressividade da linguagem de templates e, por isso, pretendemos criar um sistema de transformação para cada linguagem que pretendemos usar como linguagem-objeto.

5.1. TXL

TXL [Cordy, 2004] é um sistema de transformação baseado em regras de reescrita, independente de linguagem-objeto. Um metaprograma em TXL é composto pela descrição da gramática da linguagem-objeto e por um conjunto de regras de transformação estruturais especificados como pares padrão/ regra de reescrita.

As regras de reescrita em TXL são homomórficas, ou seja, retornam sempre uma árvore do mesmo tipo da árvore recebida como parâmetro. Desta forma, é possível garantir que um programa TXL sempre gera saídas bem formadas de acordo com a sintaxe fornecida. Em JaTS, a garantia está no fato de que ele trata apenas uma linguagem e, portanto, conhece sua sintaxe. Por exemplo, uma declaração iterativa pode gerar uma declaração de atributo dentro do corpo de uma classe, mas não dentro do corpo de um método.

TXL é um sistema bastante maduro, tendo sido usado em diversas aplicações, porém, tanto as regras de reescrita, quanto os padrões para casamento que são suportados são simples, o que impede seu uso em transformações complexas, como as requeridas para implementar reestruturações de programas.

5.2. DMS

DMS (Design Maintenance System) [Baxter et al., 2004], como TXL também é um sistema de transformação baseado em regras de reescrita com suporte a múltiplas linguagens, atualmente suporta uma ampla gama de linguagens, desde linguagens de scripting, passando por linguagens estruturais até linguagens orientadas a objeto. Em especial, pode,

facilmente, ser utilizado para transformar código escrito em uma linguagem A em código escrito em outra linguagem B, por exemplo, transformar código Java em código C#.

Uma transformação em DMS é composta por quatro elementos: o domínio (na notação de DMS, um domínio é um parser e um prettyprinter para uma dada linguagem) de entrada, o domínio de saída, as variáveis da transformação e a regra de reescrita, que é composta por um par padrão de entrada/padrão de saída, escrito, respectivamente, utilizando-se os domínios de entrada e de saída.

DMS tem sido usado em vários projetos (acadêmicos e comerciais) nas várias áreas em que a transformação de programas pode ser utilizada, tem a vantagem de poder ser utilizado em software de larga escala, com milhares de linhas de código. Entretanto, assim como em TXL, as transformações que podem ser expressas pelas regras de reescrita são bastante simples. Os construtores de metaprogramação são simples e, adicionar uma nova linguagem a este sistema é bastante custoso.

5.3. Stratego

Stratego [Visser, 2001] é uma linguagem para especificação de sistemas de transformação baseados em estratégias de reescrita [Visser, 2000]. Um programa em Stratego é composto da especificação da gramática da linguagem-objeto, regras de transformação e estratégias de reescrita. A gramática da linguagem objeto é descrita, utilizando-se o formalismo SDF (Syntax Definition Formalism) [Visser, 1997].

Stratego traz a vantagem de usar um formalismo bastante poderoso para expressar as novas linguagens e tem sido usado em diversas aplicações. Uma especialmente útil, permite embutir linguagens de domínio específico em linguagens de propósito geral [Visser and Bravenboer, 2004]. Em Stratego, as regras de reescrita são escritas usando-se a sintaxe concreta da linguagem-objeto, entretanto, as regras de transformação são simples, não admitem, por exemplo, iteração sobre elementos, declarações condicionais.

5.4. MetaJ

MetaJ [Oliveira, 2004, Oliveira et al., 2004] é um ambiente para metaprogramação baseado no paradigma orientado a objetos extensível para qualquer linguagem. Essa extensão é feita mediante a definição de *plug-ins*, responsáveis por tratar as particularidades de cada linguagem. Diferente de Stratego e de TXL, MetaJ apresenta uma construção que permite o casamento opcional, como em JaTS, mas não oferece cosntruções como declarações iterativas e condicionais.

Para estender MetaJ e incluir uma nova linguagem, devemos implementar um *prettyprinter* para esta linguagem. A linguagem de templates é gerada de forma automática e os nós da árvore sintática herdam de nós genéricos. Ao fazer uso de uma árvore genérica, MetaJ acaba por perder expressividade no casamento de padrões.

Em termos de adição de uma nova linguagem, este trabalho é o que mais se assemelha ao nosso. Porém, como não queremos perder poder de expressividade, não reusamos um framework para tratar múltiplas linguagens. Ao invés disso, optamos por gerar vários sistemas, cada um contendo as peculiaridades da respectiva linguagem. Consequentemente, com maior poder de expressividade. Para reduzir os custos de criação de um novo sistema, simplificamos parte do trabalho, utilizando programação gerativa para gerar os componentes de um sistema de transformação.

6. Conclusões

Neste artigo, mostramos uma abordagem para a geração de uma linha de produtos de sistemas de transformação baseados em um conjunto de construções de metaprogramação

introduzidos por JaTS. O custo de geração de um novo sistema, utilizando-se as transformações já implementadas, é baixo quando se compara aos benefícios de se ter sistema de transformação que leva em consideração a semântica da linguagem-objeto e, conseqüentemente, pode exprimir transformações elaboradas de forma mais simples e concisa.

Os resultados obtidos na etapa de transformação da árvore sintática e na geração dos *visitors*, mostram que esta solução é adequada para a criação de novos sistemas de transformação.

Para a extensão da sintaxe da linguagem-objeto, o trabalho ainda é 100% manual. Nos três testes realizados, entretanto, este custo não foi muito alto, devido a particularidades das linguagens testadas. De forma que o ganho obtido na etapa de transformação da árvore e de geração dos *visitors* aliado ao reuso de componentes de apoio foi realmente representativo.

Um módulo para adaptação do parser da linguagem-objeto será desenvolvido para que esta atividade possa ser realizada de forma semi-automática, reduzindo o custo da geração destes sistemas. Este é um importante passo para permitir que os novos sistemas gerados através deste mecanismo possam ser utilizados na construção de sistemas cada vez mais complexos.

7. Agradecimentos

Gostaríamos de agradecer aos membros do ***Software Productivity Group***, em especial a Vander Alves, Gustavo Santos, Ivan Cardim, Tiago Massoni e Rohit Gheyi pelas sugestões e críticas.

Referências

(2005). *Html Markup Language*. W3C, http://www.w3.org/MarkUp/.

(2005). *JavaNCSS: A Source Measurement Suite for Java*. http://www.kclee.de/clemens/java/javancss/.

(2005). *Linguagem Orientada a Objetos 1*. CIn, http://www.cin.ufpe.br/ in1007/linguagens/ OrientadaObjetos1/orientadaObjetos1.html.

(2005). Qualiti coder. http://coder.qualiti.com.

Baxter, I. D., Pidgeon, C., and Mehlich, M. (2004). DMS®: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Washington, DC, USA. IEEE Computer Society.

Castor, F. (2001). Definição de uma Linguagem para Especificar Transformações em Java. Trabalho de Graduação.

Castor, F. and Borba, P. (2001). A language for specifying Java transformations. In *SBLP '01: Proceedings of the 5th Brazilian Symposium on Programming Languages*, pages 236–251, Curitiba, PR, BRAZIL.

Castor, F., Oliveira, K., Souza, A., Santos, G., and Borba, P. (2001). JaTS: A Java transformation system. In *XV Simpósio Brasileiro de Engenharia de Software*, pages 374–379.

Cordy, J. R. (2004). TXL - A language for programming language tools and applications. In *Proceedings of LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications*, pages 1–27.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools and Applications*. Addison-Wesley.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (1996). *The Java Language Specification*. Java Series. Addison-Wesley, 2th edition.

Hall, M. (2000). *Core Servlets and Java Server Pages*. Prentice Hall.

Harold, E. and Means, W. (2001). *XML in a Nutshell*. O'Reilly.

Liberty, J. (2003). *Programming in C#*. O' Reilly.

Oliveira, A. (2004). MetaJ: Um ambiente para metaprogramação em java. Master's thesis, Universidade Federal de Minas Gerais.

Oliveira, A., Braga, T., Maia, M., and Bigonha, R. (2004). Metaj: An extensible environment for metaprogramming in java. In *VIII Simpósio Brasileiro de Linguagens de Programação*.

Sherif, A. and He, J. (2002). Towards a time model for circus. In *ICFEM*, pages 613–624.

Travassos, G., Gurov, D., and Amaral, E. (2003). Introdução a Engenharia de Software Experimental. Technical report, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil.

Visser, E. (1997). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Amsterdam.

Visser, E. (2000). *The Stratego Reference Manual*. Institute of Information and Computing Sciences, Utrecht University, 0.5 edition. Technical Documentation.

Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357+.

Visser, E. and Bravenboer, M. (2004). Concrete syntax for objects. domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th ACM/SIGSOFT Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 365–383.