Coloração Automática de Variabilidades em Linhas de Produtos de Software

Virgílio Borges de Oliveira, Marco Túlio Valente

Instituto de Informática, PUC Minas

virgilio@cotemig.com.br, mtov@pucminas.br

Resumo. *Neste artigo, apresenta-se um algoritmo para coloração automática de código responsável por implementar variabilidades em linhas de produtos. O algoritmo proposto é baseado em uma ferramenta para implementação de variabilidades chamada CIDE (Colored IDE), a qual permite colorir manualmente trechos de código associados a uma determinada variabilidade. Além disso, essa ferramenta permite gerar projeções de um sistema nas quais o código anotado com uma determinada cor não é mais exibido. O algoritmo proposto foi capaz de extrair automaticamente quatro variabilidades do framework de persistência Prevayler.*

Abstract. *This paper details an algorithm that automatically annotates blocks of code associated to variabilities in software product lines. The proposed algorithm is based on a tool to implement variabilities called CIDE (Colored IDE). CIDE enhances standard IDE with the ability to associate colors to lines of code in charge of implementing variabilities. Furthermore, the tool supports the generation of projections of a system in which the code annotated with a given color is not showed. Using the proposed algorithm, we have automatically extracted four variabilities from the Prevayler persistence framework .*

1 Introdução

Linhas de Produtos de Software (LPS) constitui uma abordagem de desenvolvimento que advoga a construção de famílias de sistemas a partir de um conjunto de componentes e artefatos comuns [16, 4, 18]. O objetivo final é promover o reúso sistemático de componentes de software e, como consequência, incorporar ganhos efetivos de produtividade e qualidade no processo de desenvolvimento de sistemas. Dentre os diferentes modelos para adoção de linhas de produtos, destaca-se o modelo extrativo [10]. Nesse modelo, uma LPS é extraída de um sistema existente, isto é, um sistema implementado sem a preocupação de separar funcionalidades básicas de funcionalidades particulares de um domínio de uso. Essas últimas funcionalidades, na terminologia de LPS, são chamadas de variabilidades [19].

Do ponto de vista de implementação, as soluções para extração de variabilidades podem ser classificadas em dois grandes grupos: baseadas em composição e baseadas em anotações [6]. Soluções baseadas em composição – tais como programação orientada por aspectos – propõem que as variabilidades de uma família de produtos devem ser lexicamente implementadas em módulos distintos. Com isso, na fase de composição de um sistema, desenvolvedores podem escolher os módulos que deverão ser incluídos em

sua compilação. No entanto, tais tecnologias disponibilizam um modelo de granularidade grossa (*coarse-grained*) para extensão de componentes. Por exemplo, em AspectJ desenvolvedores podem instrumentar apenas pontos pré-definidos da execução de um sistema, os quais são chamados de pontos de junção (*join points*). Caso uma variabilidade seja demandada em uma parte do sistema que não corresponde a um ponto de junção, desenvolvedores são obrigados a lançar mão de diversos artifícios para transformar o código base, de forma a permitir a sua extensão por meio de aspectos [15, 11, 14].

Por outro lado, soluções baseadas em anotações permitem marcar e associar trechos arbitrários de código a uma determinada variabilidade. Portanto, disponibilizam um modelo de granularidade mais fina (*fine-grained*) para extensão de componentes. A princípio, esse modelo é mais adequado para extração de variabilidades em sistemas que não foram implementados tendo em vista uma abordagem de desenvolvimento baseada em LPS. Diretivas de pré-processamento, tais como `#ifdef` e `#endif`, consistem na solução baseada em anotações mais conhecida [8]. Porém, tais diretivas tendem a dificultar o entendimento e a evolução de um sistema, pois ficam entrelaçadas em diversas partes do código [17].

A fim de atenuar os problemas típicos de diretivas de pré-processamento, foi recentemente proposta uma nova solução para implementação de variabilidades baseada em anotações, chamada CIDE (*Colored IDE*) [6, 7]. Essa ferramenta permite colorir manualmente trechos de código associados a uma determinada variabilidade (de forma similar, por exemplo, àquela que se usa em editores de texto para mudar a cor de uma sentença). Assim, em vez de estender uma linguagem de programação com diretivas de pré-processamento, essa solução estende uma IDE com recursos para coloração de trechos de código. A vantagem nesse caso é que pode-se facilmente – também via IDE – gerar uma projeção do sistema na qual os trechos de código marcados com uma determinada cor não são exibidos. Dessa forma, disponibiliza-se um mecanismo para separação virtual de interesses, por meio do qual atenua-se o problema de entrelaçamento de código típico de diretivas de pré-processamento.

No entanto, em sistemas mais complexos, a coloração manual de trechos de código associados a variabilidades é uma tarefa tediosa, repetitiva e sujeita a erros. Assim, com o objetivo de auxiliar desenvolvedores a extrair de forma mais rápida uma LPS a partir de um sistema existente, apresenta-se neste artigo um algoritmo para coloração automática de código responsável por implementar variabilidades em linhas de produtos. Basicamente, desenvolvedores devem fornecer como entrada para esse algoritmo um conjunto de unidades sintáticas que são responsáveis pela implementação de uma variabilidade (por exemplo, métodos, campos, classes ou pacotes). O algoritmo proposto automaticamente localiza então todos os pontos do código onde essas unidades são usadas e trata de marcá-los com uma cor escolhida pelo desenvolvedor.

O algoritmo proposto foi aplicado para extração de variabilidades existentes no sistema Prevayler, um *framework* de persistência que já foi utilizado como exemplo em outros trabalhos sobre implementação de linhas de produtos [12, 5]. Na experiência realizada, foram coloridos trechos de código responsáveis por quatro variabilidades desse sistema: replicação, monitor, censura e *snapshot*. Os resultados comprovaram que o algoritmo proposto foi capaz de colorir integralmente os trechos de código responsáveis pela implementação de três dessas variabilidades (replicação, monitor e censura). No

caso da funcionalidade de *snapshot*, o algoritmo alcançou uma coberta de 83% do código responsável por essa funcionalidade.

O restante deste artigo está organizado conforme descrito a seguir. Na Seção 2, apresenta-se uma visão geral da ferramenta CIDE. A Seção 3 descreve o algoritmo para coloração automática de variabilidades proposto para essa ferramenta. A Seção 4 descreve uma experiência de aplicação desse algoritmo na extração de variabilidades existentes no código do sistema Prevayler. A Seção 5 apresenta algumas das lições aprendidas nessa experiência. Por fim, a Seção 6 discute trabalhos relacionados e a Seção 7 apresenta as conclusões.

2 A Ferramenta CIDE

CIDE é uma ferramenta para implementação de variabilidades baseada em anotações [6, 7]. Conforme mostrado na Figura 1, a ferramenta permite que programadores associem cores aos trechos de código responsáveis pela implementação de alguma forma de variabilidade. Nessa figura, as linhas de código responsáveis pelo interesse de sincronização foram marcadas com a cor azul; e as linhas responsáveis pela implementação de *logging* foram marcadas de amarelo[1].

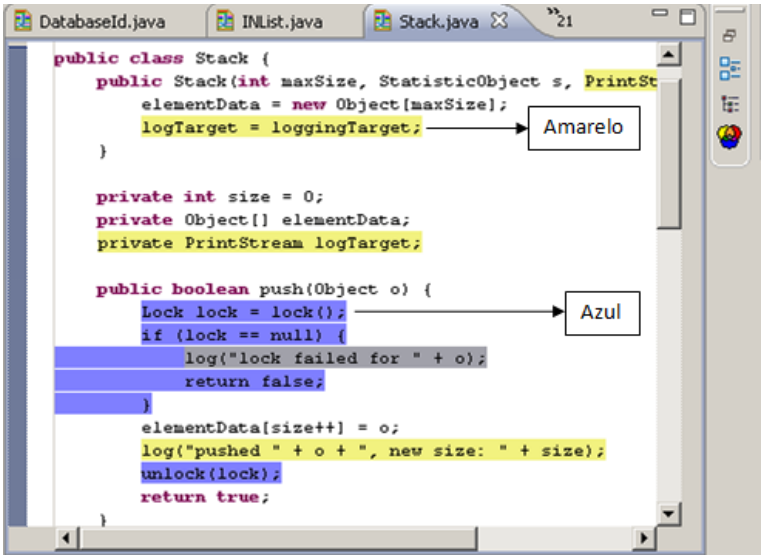

Figura 1. Screenshot da ferramenta CIDE

Além de recursos para coloração de código, a ferramenta CIDE permite que desenvolvedores avancem sequencialmente pelos trechos de código associados a uma determinada cor (usando para isso um botão *avançar*). A ferramenta permite também gerar uma projeção do sistema na qual todo o código associado a uma determinada cor é removido. Por meio dessas projeções, desenvolvedores podem sintetizar os mais diversos produtos possíveis em uma determinada família de sistemas.

Conforme afirmado, a ferramenta permite que cores sejam usadas para anotar elementos estruturais do código fonte. Esses elementos devem corresponder a nodos opcionais da AST (Árvore de Sintaxe Abstrata) da linguagem fonte. O fato de ser uma solução

[1]Essa figura foi extraída de [7]. Para facilitar seu entendimento quando impressa em preto e branco, as setas indicam as cores usadas na coloração.

baseada em sintaxe abstrata impede que sejam coloridos elementos de código de menor importância, tais como vírgulas, parênteses, chaves etc (visto que tais elementos não são representados na AST). Além disso, como apenas nodos opcionais da AST podem ser coloridos, impede-se também que sejam anotados trechos de código cuja remoção invalidaria sintaticamente o programa base (como, por exemplo, a expressão de um comando `return`). Em resumo, o fato de ser baseada na AST da linguagem alvo aumenta a granularidade das variabilidades que podem ser anotadas pela ferramenta (quando comparado com diretivas de pré-processamento, onde o programador tem total liberdade para anotar trechos arbitrários de código); porém, essa característica também evita que sejam coloridos trechos de código cuja remoção pode levar a uma projeção do sistema com erros de sintaxe.

Quando comparado com soluções baseadas em composição, como programação orientada por aspectos [9] e programação orientada por *features* [2], CIDE propicia um grau de granularidade mais detalhado na marcação de código. Por exemplo, pode-se marcar comandos arbitrários do código (e não apenas comandos no início ou no final de métodos), declarações de parâmetros, declarações de variáveis locais, parâmetros de chamada de métodos, cláusulas `throws` etc.

A Tabela 1 compara a solução oferecida pela ferramenta CIDE com as soluções providas por meio de compilação condicional (CC) e orientação por aspectos (AOP).

Características	**CC**	**AOP**	**CIDE**
Granularidade	Muito fina (trechos de código arbitrários)	Grossa (apenas pontos de junção)	Fina (nodos opcionais da AST)
Separação de Interesses	Não (entrelaçamento e espalhamento de diretivas)	Sim (física, em módulos separados, chamados aspectos)	Sim (virtual, via projeções sem código de determinadas cores)
Verificação sintática	Não (pode-se gerar produtos com erros de sintaxe)	Sim (aspectos são compilados separadamente)	Sim (apenas nodos opcionais da AST podem ser coloridos)

Tabela 1. Comparação entre soluções para implementação de variabilidades

3 Algoritmo para Coloração de Variabilidades

O algoritmo proposto recebe como entrada um conjunto de unidades sintáticas responsáveis pela implementação de uma variabilidade. Na terminologia adotada, essas unidades são chamadas de sementes de uma variabilidade (ou simplesmente, sementes). Os seguintes elementos sintáticos podem ser fornecidos como sementes no algoritmo proposto:

- campos (por exemplo, `Logger.filename`);
- métodos (por exemplo, `Logger.log(String)`);
- classes ou interfaces (por exemplo, `Logger`);
- pacotes (por exemplo, `mysystem.util`).

A Figura 2 apresenta a rotina principal do algoritmo proposto. Basicamente, o algoritmo é dividido em duas fases: propagação e expansão. Na fase de propagação, os trechos de código C_1 que usam diretamente as sementes S fornecidas como entrada são coloridos; em seguida, os trechos de código C_2 que referenciam elementos declarados em

C_1 são coloridos e assim sucessivamente, até que nenhum novo elemento de código possa ser colorido. Já o objetivo da fase de expansão é verificar se a vizinhança de um trecho de código colorido deve também ser colorida. Por exemplo, se o corpo de um comando `while` for colorido, então a expressão desse comando também deve ser colorida. Essas duas fases são descritas com mais detalhes no final da seção.

$$
\begin{array}{l}
Main(VariabilitySeed\ S, Color\ c) = \\
\quad \forall s \in S \rightarrow ColorPropagation(s, c); \\
\quad ColorExpansion();
\end{array}
$$

Figura 2. Rotina principal

As Tabelas 2 e 3 apresentam um conjunto de funções auxiliares usadas nas fases de propagação e expansão de cores. Na Tabela 2, são listadas funções que permitem obter informações sintáticas do programa base, por meio de pesquisas em sua AST. Na Tabela 3, são apresentadas funções que retornam nodos da AST responsáveis por uma determinada tarefa[2]. Por exemplo, a função `call(m)` retorna todos os nodos contendo chamadas ao método `m`. Funções como essa são usadas para obter os nodos que deverão ser coloridos. Por fim, a função *cide(t,p)* é usada para colorir com a cor p todos os nodos t da AST do programa base.

Função	Retorno
$classes(p)$	classes implementadas no pacote p
$interfaces(p)$	interfaces definidas no pacote p
$meths(t)$	métodos implementados na classe t
$fields(t)$	campos declarados na classe t
$hasType(t)$	variáveis locais e parâmetros formais do tipo t
$impls(i)$	classes que implementam a interface i
$extends(t)$	subclasses da classe t
$formal(p)$	parâmetro formal associado ao parâmetro de chamada p

Tabela 2. Funções de consulta na AST

Função	Retorno
`call(m)`	chamadas do método m
`access(x)`	leitura/escrita ao campo ou variável local x
`declaration(x)`	declaração do campo ou da variável local x
`import(t)`	importação de classes do tipo t
`import(p.*)`	importação de quaisquer classes do pacote p
`new(t)`	instanciações de objetos do tipo t
`implementation(m)`	implementação do método m
`definition(i)`	definição da interface i
`package(p)`	implementação do pacote p

Tabela 3. Funções que retornam nodos da AST

[2]Como essas funções retornam informações sintáticas do código, elas são grafadas em fonte *typewriter*.

$1:\quad ColorPropagation(Package\ p, Color\ c) =$ $(P1)$
$2:\quad\quad \forall t \in classes(p) \rightarrow ColorPropagation(t, c);$
$3:\quad\quad \forall i \in interfaces(p) \rightarrow ColorPropagation(i, c);$
$4:\quad\quad \forall p = \texttt{import(p.*)} \rightarrow cide(p, c);$

$5:\quad ColorPropagation(Class\ t, Color\ c) =$ $(P2)$
$6:\quad\quad \forall m \in meths(t) \rightarrow ColorPropagation(m, c);$
$7:\quad\quad \forall f \in fields(t) \rightarrow ColorPropagation(f, c);$
$8:\quad\quad \forall s \in extends(t) \rightarrow ColorPropagation(s, c);$
$9:\quad\quad \forall i \in hasType(t) \rightarrow ColorPropagation(i, c);$
$10:\quad\quad \forall n = \texttt{new(t)} \rightarrow cide(n, c);$
$11:\quad\quad \forall p = \texttt{import(t)} \rightarrow cide(p, c);$

$12:\quad ColorPropagation(Interface\ i, Color\ c) =$ $(P3)$
$13:\quad\quad p = \texttt{definition(i)} \rightarrow cide(p, c);$
$14:\quad\quad \forall t \in impls(i) \wedge \forall m \in meths(t) \wedge m \in i \rightarrow ColorPropagation(m, c);$
$15:\quad\quad \forall t \in hasType(i) \rightarrow ColorPropagation(t, c);$
$16:\quad\quad \forall p = \texttt{import(i)} \rightarrow cide(p, c);$

$17:\quad ColorPropagation(Method\ m, Color\ c) =$ $(P4)$
$18:\quad\quad p = \texttt{implementation(m)} \rightarrow cide(p, c);$
$19:\quad\quad \forall s = \texttt{call(m)} \rightarrow cide(s, c).$

$20:\quad ColorPropagation(Field\ f, Color\ c) =$ $(P5)$
$21:\quad\quad p = \texttt{declaration(f)} \rightarrow cide(p, c);$
$22:\quad\quad \forall s = \texttt{access(f)} \rightarrow cide(s, c).$

$23:\quad ColorPropagation(LocalVariable\ i, Color\ c) =$ $(P6)$
$24:\quad\quad p = \texttt{declaration(i)} \rightarrow cide(p, c);$
$25:\quad\quad \forall s = \texttt{access(i)} \rightarrow cide(s, c).$

Figura 3. Propagação de Cores

Propagação de Cores: A Figura 3 apresenta o algoritmo de propagação de cores. Basicamente, esse algoritmo propaga recursivamente uma determinada cor por todos os trechos de código que implementam ou usam uma determinada semente. Mais especificamente, as unidades sintáticas que podem ser alvo de propagação de cores incluem pacotes (regra P1), classes (regra P2), interfaces (regra P3), métodos (regra P4), campos (regra P5) e variáveis locais (regra P6). Essas regras são descritas com mais detalhes a seguir:

- Regra P1: Define que para colorir um pacote deve-se propagar a cor pelas classes e interfaces declaradas nesse pacote (linhas 2-3). Deve-se também colorir os comandos `import` envolvendo as classes do pacote (linha 4).

- Regra P2: Define que para colorir uma classe deve-se propagar a cor pelos métodos e campos da classe (linhas 6-7), bem como pelos seguintes usos da classe: subclasses (linha 8), declaração de variáveis locais do tipo da classe (linha 9), instanciações de objetos do tipo da classe (linha 10) e comandos `import` do tipo da classe (linha 11).

- Regra P3: Define que para colorir uma interface deve-se propagar a cor pela sua definição (linha 13), bem como pelos seguintes usos da interface: implementações de métodos da interface (linha 14), declaração de variáveis locais do tipo da interface (linha 15) e comandos `import` do tipo da interface (linha 16).

- Regra P4: Define que para colorir um método deve-se propagar a cor pela sua implementação (linha 18), bem como pelas chamadas desse método (linha 19).

- Regra P5: Define que para colorir um campo deve-se propagar a cor pela sua declaração (linha 21), bem como por todos os pontos do sistema onde o campo é lido ou alterado (linha 22).

- Regra P6: Define que para colorir uma variável local deve-se propagar a cor pela sua declaração (linha 24), bem como por todos os pontos do sistema onde a referida variável é lida ou alterada (linha 25).

Expansão de Cores: A Figura 4 apresenta o algoritmo de expansão de cores. Basicamente, esse algoritmo consiste em um laço no qual são sequencialmente chamadas as várias regras de expansão de uma cor por sua vizinhança imediata no código. Esse laço termina quando as expansões invocadas não acrescentam nenhuma cor nova no código do sistema (em relação à iteração anterior).

$ColorExpansion() =$
  do
    $oldcode = code;$
    $BodyExpansion();$
    $ExpExpansion();$
    $ElseStmExpansion();$
    $AssignExpansion();$
    $FormalParameterExpansion();$
  while $code \neq oldcode$

Figura 4. Expansão de Cores

As regras para expansão de cores – apresentadas na Figura 5 – são descritas com mais detalhes a seguir:

- Regra E1: Define que a cor da expressão dos comandos `if`, `while`, `do while`, `switch` ou `for` deve ser expandida para o corpo de tais comandos.

- Regra E2: Define que a cor do corpo de comandos `if`, `while`, `do while`, `switch` ou `for` deve ser expandida para a expressão de tais comandos.

- Regra E3: Define que a cor dos comandos de uma cláusula `else` deve ser expandida para incluir a própria cláusula.

$$BodyExpansion() = \quad (E1)$$
$$s = [\texttt{if(exp) stm}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{while exp stm}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{do stm while exp}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{case(exp) \{stm\}}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{switch(exp) \{stm\}}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) stm}] \wedge color(e_1, c) \wedge color(e_2, c) \wedge color(e_3, c) \rightarrow cide(s, c);$$

$$ExpExpansion() = \quad (E2)$$
$$s = [\texttt{if(exp) stm}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{while exp stm}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{do stm while exp}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{if(exp) }\texttt{stm}_1\texttt{ else }\texttt{stm}_2] \wedge color(\texttt{stm}_1, c) \wedge color(\texttt{stm}_2, c) \rightarrow cide(s, c);$$
$$s = [\texttt{case(exp) \{stm\}}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{switch(exp) \{stm\}}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) stm}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);$$

$$ElseStmExpansion() = \quad (E3)$$
$$s = [\texttt{else stm}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);$$

$$AssignExpansion() = \quad (E4)$$
$$s = [\texttt{i = exp;}] \wedge color(\texttt{i}, c) \rightarrow cide(s, c);$$
$$s = [\texttt{i = exp;}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c).$$

$$FormalParameterExpansion() = \quad (E5)$$
$$s = [\texttt{m}\,(\ldots, \texttt{p}_\texttt{i}, \ldots)] \wedge color(\texttt{p}_\texttt{i}, c) \rightarrow ColorPropagation(formal(\texttt{p}_\texttt{i}), c).$$

Figura 5. Regras para Expansão de Cores

- Regra E4: Define que a cor do lado esquerdo de um comando de atribuição deve ser expandida para seu lado direito e vice-versa.

- Regra E5: Define que a cor de um parâmetro de chamada deve ser propagada para o seu respectivo parâmetro formal.

4 Exemplo: Prevayler

Prevayler é um sistema para persistência de objetos em Java[3]. O sistema possui 2974 LOC e já foi usado como exemplo em trabalhos sobre implementação de linhas de produtos usando orientação por aspectos [5] e orientação por *features* [12]. Para o estudo de caso apresentado neste artigo foram selecionadas quatro variabilidades presentes na versão monolítica desse sistema:

- Replicação: permite a replicação de dados entre um servidor e diversos clientes.
- Monitor: *logging* de eventos internos do sistema.

[3] `http://www.prevayler.org.`

- *Snapshot*: armazenamento em disco de objetos. Sem esta variabilidade, o armazenamento é feito apenas em memória primária (modo transiente).
- Censura: recuperação dos dados em caso de falha na execução de uma transação. É dependente da variabilidade *snapshot* (isto é, sempre que *censura* for incluída na derivação de um produto, *snapshot* também deverá ser incluída).

Metodologia: Após definidas as quatro variabilidades mencionadas anteriormente, foram extraídas duas linhas de produtos a partir da versão original do sistema Prevayler. Na primeira LPS, a ferramenta CIDE foi aplicada de forma manual, isto é, os trechos de código relativos a cada uma das variabilidades mencionadas anteriormente foram manualmente localizados e então coloridos. O objetivo dessa primeira extração foi gerar uma LPS de referência, que pudesse ser usada para avaliar a precisão do algoritmo proposto.

Na segunda extração, usou-se o algoritmo proposto neste trabalho, de forma a colorir automaticamente os trechos de código associados a cada uma das variabilidades identificadas. O algoritmo foi executado pelo mesmo desenvolvedor responsável pela extração manual. Assim, coube a esse desenvolvedor escolher as sementes, baseado em seu conhecimento do projeto do sistema Prevayler. Para a variabilidade *replicação*, as sementes fornecidas como entrada para o algoritmo foram os seguintes atributos da classe `PrevaylerFactory`:

```
- OldNetwork _network;
- int _serverPort;
- String _remoteServerIpAddress;
- int _remoteServerPort;
- int DEFAULT_REPLICATION_PORT;
```

Esses atributos armazenam o endereço de rede do servidor, do cliente e a porta de comunicação. Para a variabilidade *monitor*, a semente fornecida foi o pacote `org.prevayler.foundation.monitor`. Para a variabilidade censura, a semente fornecida foi o pacote `org.prevayler.implementation.publishing.censorship`. Por fim, para a variabilidade *snapshot*, a semente fornecida foi o pacote `org.prevayler.implementation.snapshot` e dois atributos da classe `PrevaylerFactory`:

```
- Map _snapshotSerializers;
- String _primarySnapshotSuffix;
```

Resultados: A Tabela 4 apresenta os resultados obtidos nas duas extrações. As linhas dessa tabela representam os diversos tipos de nodos da AST que foram coloridos na extração das quatro variabilidades consideradas no experimento (representadas nas colunas da tabela). Por exemplo, dentre os trechos de código coloridos incluem-se blocos de comando, unidades de compilação (isto é, arquivos), expressões, declarações de campos, comandos `if`, etc. A Tabela 5 apresenta o total de *bytes* do código fonte do Prevayler que foi colorido nas duas linhas de produtos extraídas. Os resultados foram gerados pela funcionalidade *Collect Interactions* da ferramenta CIDE. Os pacotes com as classes de demonstração do Prevayler foram desconsiderados no experimento para que a análise fosse realizada apenas no código do *framework*.

Nodos da AST	Variabilidade							
	Replicação		Monitor		Censura		Snapshot	
	M	A	M	A	M	A	M	A
Block	-	-	-	-	-	-	1	2
CompilationUnit	27	27	5	5	3	3	2	2
ExpressionStatement	1	1	4	4	2	2	6	5
FieldDeclaration	5	5	3	3	1	1	4	4
IfStatement	2	2	-	-	-	-	-	1
ImportDeclaration	4	4	4	4	4	4	4	4
MethodDeclaration	4	4	2	2	1	1	11	2
MethodInvocation	-	-	1	1	1	1	1	-
ReturnStatement	1	1	-	-	-	-	-	-
SimpleName	-	-	2	2	-	-	3	3
SingleVariableDeclaration	-	-	2	2	1	1	2	3
SynchronizedStatement	-	-	-	-	-	-	-	1
VariableDeclarationStatement	-	-	-	-	-	-	1	1
Total	44	44	23	23	13	13	37	28

Tabela 4. Resultados (M= extração manual; A= extração via algoritmo proposto)

Variabilidades	Bytes coloridos		A / M
	M	A	
Replicação	45309	45309	100%
Monitor	6725	6725	100%
Censura	4393	4393	100%
Snapshot	13371	11082	83%

Tabela 5. Número de bytes coloridos para extração das duas linhas de produtos (M= extração manual; A= extração via algoritmo proposto)

Como pode ser observado nas Tabelas 4 e 5, para as variabilidades referentes a replicação, monitor e censura, os elementos de código automaticamente coloridos pelo algoritmo proposto corresponderam exatamente àqueles que foram previamente coloridos na experiência de extração manual dessas variabilidades. Portanto, o algoritmo proposto foi capaz de automatizar com sucesso a extração dessas três variabilidades.

No caso da variabilidade *snapshot*, os trechos de código coloridos – quando comparados em termos de *bytes* – corresponderam a uma cobertura de 83% dos trechos de código coloridos na experiência de extração manual dessa variabilidade (conforme mostrado na Tabela 5). Mais especificamente, uma expressão e nove implementações de métodos não foram coloridas (conforme indicado na Tabela 4).

Na Figura 6, mostra-se um exemplo de método que não foi integralmente colorido. Nesse método, o comando `if` (linhas 3-5) não foi colorido pelo algoritmo proposto. Esse comando ativa uma exceção quando um *snapshot* for requisitado com o sistema de persistência desabilitado. No entanto, a dependência entre a variabilidade *snapshot* e o campo `_prevalentSystem` não é estática, mas ocorre apenas quando o valor desse campo for igual a `null`. Por esse motivo, esse campo não foi incluído como uma semente

da variabilidade *snapshot*.

Em outras palavras, na versão atual do algoritmo, um campo f deve ser considerado como semente de uma variabilidade V quando qualquer uso de f denotar código associado a V. No entanto, no sistema Prevayler, existem situações em que a associação entre um campo f e uma variabilidade V não é estática, mas dependente do valor de f em tempo de execução. Como mostrado na Figura 6, essa limitação do algoritmo explica a coloração parcial do código atrelado à variabilidade *snapshot*. No entanto, mesmo com essa limitação, a automatização proporcionada pelo algoritmo de coloração chegou a 83% do código associado de forma manual à funcionalidade de *snapshot*. Nas demais variabilidades consideradas, essa limitação não se manifestou.

```
 1: void takeSnapshot(GenericSnapshotManager snapshotManager) {
 2:  synchronized (this) {
 3:    if (_prevalentSystem == null) {     // comando não-colorido
 4:      throw new Error("No longer allowing snapshots...");
 5:    }
 6:    synchronized (_prevalentSystem) {  // comando colorido
 7:      snapshotManager.writeSnapshot(...);
 8:    }
 9:  }
10:}
```

Figura 6. Exemplo de método parcialmente colorido

5 Discussão

Nesta seção são discutidas algumas características do algoritmo proposto:

- Expressividade: O algoritmo proposto foi capaz de automatizar integralmente a coloração – e, portanto, a extração – de três das quatro variabilidades consideradas no exemplo envolvendo o sistema Prevayler. Na quarta variabilidade (*snapshot*), a cobertura da coloração automática foi de 83%, devido à incapacidade do algoritmo tratar valores dinâmicos de atributos e variáveis locais.

- Facilidade de uso: A execução do algoritmo proposto é bastante simples: basta informar como entrada as sementes. No entanto, para determinar as sementes, o usuário deverá ter algum domínio do sistema alvo. Essencialmente, ele precisa ter conhecimento dos elementos (campos, métodos, classes e pacotes) que são responsáveis pela implementação de uma variabilidade. Além disso, a cobertura proporcionada pelo algoritmo é dependente das sementes informadas como entrada. Por exemplo, caso se esqueça de informar um método como semente, a cobertura alcançada não incluirá as chamadas desse método.

- Granularidade: Como pode ser observado na Tabela 4, diversos nodos da AST coloridos no Prevayler não podem ser diretamente extraídos para módulos oferecidos por soluções baseadas em composição, como aspectos e *features*. Como exemplo, podemos citar expressões, comandos `if`, comandos `import` etc. Por outro lado, todos esses nodos podem ser diretamente coloridos por meio da ferramenta CIDE (seja de forma manual ou usando o algoritmo aqui proposto). Por

esse motivo, na implementação orientada por aspectos do sistema Prevayler [5], diversas transformações tiveram que ser aplicadas no código base a fim de habilitar a extração de aspectos [15].

6 Trabalhos Relacionados

Os principais trabalhos relacionados podem ser agrupados em duas categorias: relatos de experiência e ferramentas para extração de *features*.

Relatos de Experiência: Lopes-Herrejon, Batory e Cook apresentam uma avaliação de diversas tecnologias de modularização, usando como estudo de caso uma gramática simples de expressões [13]. As variabilidades no caso são as diversas operações aritméticas que podem ser incorporadas a essa gramática. Devido à simplicidade desse exemplo, não foram avaliadas extensões de granularidade fina, como aquelas viabilizadas pela ferramenta CIDE.

Kästner, Apel e Batory documentam o uso de AspectJ para refatorar um sistema gerenciador de bancos de dados (Oracle Berkley DB) em uma linha de produtos [11]. Nesse caso, por ser um sistema de maior porte e complexidade, os autores reportam diversas limitações de AspectJ para tratar variabilidades de granularidade fina. Por exemplo, afirmam que foi difícil implementar diversas *features* porque elas demandavam extensões em comandos internos de métodos. A solução nesse caso normalmente consistiu na extração de métodos vazios (*hook methods*), o que quase sempre acabava por tornar o código base "estranho". Essa necessidade de extração de métodos *hook* foi também observada por Murphy et al. em outro estudo sobre extração e modularização de *features* [14]. Por fim, as limitações de AspectJ para modularizar extensões de granularidade fina implementadas por meio de compilação condicional foram reportadas recentemente por Adams et al., em um estudo de caso envolvendo uma máquina virtual para linguagens dinâmicas (Parrot VM) [1].

Ferramentas: Liu, Batory e Lenguaer descrevem uma teoria e uma ferramenta para refatoração de *features* a partir de código existente [12]. A teoria proposta é baseada em equações algébricas que permitem descrever composições de programas. Essa teoria também trata o problema de *features* que atuam sobre outras *features*. Por exemplo, uma *feature* de *logging* `L`, que atua sobre o código base `B` e sobre o código de uma *feature* opcional `F`. Nesse caso, `L` não pode ser implementada em um único módulo, requerendo dois módulos (um módulo que atuará apenas sobre o código base `B` e outro que atuará sobre o módulo `F`). No caso da solução proposta pela ferramenta CIDE, esse tipo de *feature* requer que o o código de `L` presente em `F` receba duas cores. O algoritmo descrito neste artigo também é capaz de automatizar essas situações de aplicação de múltiplas cores a um mesmo trecho de código. Por fim, os autores descrevem uma ferramenta que – baseada na teoria proposta – é capaz de extrair automaticamente *features* de sistemas legados. Essa ferramenta usa a linguagem AHEAD para modularização e composição das *features* extraídas [2]. Um estudo de caso envolvendo o sistema Prevayler é também apresentado. No entanto, AHEAD é uma solução de granularidade grossa para extensão de programas (assim como AspectJ). Apesar disso, os autores não detalham eventuais limitações que tenham sido observadas na extração de variabilidades do sistema Prevayler. Apenas men-

cionam que um reordenamento manual dos comandos de um método pode ser demandado pela ferramenta proposta e "que diversas iterações podem ser requeridas até se atingir uma refatoração aceitável".

AOP-Migrator é uma ferramenta baseada em AspectJ que disponibiliza suporte a um conjunto pré-definido de refatorações para extração de aspectos [3]. Assim como a ferramenta descrita anteriormente, ela também requer intervenção manual de programadores caso os interesses extraídos não estejam presentes em pontos do programa que possam ser instrumentados pela linguagem de definição de conjuntos de junção de AspectJ. Esse tipo de limitação não existe em variabilidades implementadas por meio da ferramenta CIDE (e do algoritmo de coloração de *features* apresentado neste artigo).

7 Conclusões

Neste trabalho apresentou-se um algoritmo para coloração automática de código responsável por implementar variabilidades em linhas de produtos. O algoritmo foi projetado como uma extensão da ferramenta CIDE, a qual concilia um modelo de granularidade fina para marcação de variabilidades com recursos para separação virtual de interesses. O algoritmo também foi aplicado com sucesso na extração de quatro *features* do sistema Prevayler: replicação, monitor, *snapshot* e censura.

Atualmente, um primeiro protótipo do algoritmo proposto encontra-se implementado e integrado à ferramenta CIDE. Como trabalho futuro, pretendemos: (a) estender o algoritmo proposto para tratar algumas construções de Java não suportadas nesta primeira versão (notadamente, tratamento de exceções); (b) realizar novos estudos de caso; (c) possivelmente enriquecer o algoritmo proposto com recursos para definição de sementes dinâmicas (a fim de atenuar, por exemplo, a limitação descrita na extração do recurso de *snapshot* do sistema Prevayler); (d) realizar uma comparação mais detalhada entre a solução provida pela ferramenta CIDE e soluções baseadas em composição.

As duas linhas de produtos de software extraídas neste artigo estão disponíveis em: `http://www.inf.pucminas.br/prof/mtov/prevayler_cide.zip`.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq. Gostaríamos de agradecer a Christian Käestner e Sven Apel pela cessão do código fonte do CIDE e pelo esclarecimento de diversas dúvidas sobre a ferramenta.

Referências

[1] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *8th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, pages 243–254, 2009.

[2] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *26th International Conference on Software Engineering (ICSE)*, pages 702–703, 2004.

[3] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions Software Engineering*, 32(9):698–717, 2006.

[4] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[5] Irum Godil and Hans-Arno Jacobsen. Horizontal decomposition of prevayler. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 83–100, 2005.

[6] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *30th International Conference on Software Engineering (ICSE)*, pages 311–320, 2008.

[7] Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing software product line variabilities in source code. In *2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 303–313, 2008.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, 1988.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[10] Charles W. Krueger. Easing the transition to software mass customization. In *4th International Workshop on Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2001.

[11] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *11th International Software Product Line Conference (SPLC)*, pages 223–232, 2007.

[12] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *28th International Conference on Software Engineering (ICSE)*, pages 112–121, 2006.

[13] Roberto Lopez-Herrejon, Don Batory, and William R. Cook. Evaluating support for features in advanced modularization technologies. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 169–194. Springer-Verlag, 2005.

[14] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: an exploratory study. In *23rd International Conference on Software Engineering (ICSE)*, pages 275–284, 2001.

[15] Marcelo Nassau and Marco Tulio Valente. Object-oriented transformations for extracting aspects. *Information and Software Technology*, 51(1):138–149, 2009.

[16] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[17] Henry Spencer. #ifdef considered harmful, or portability experience with C News. In *USENIX Conference*, pages 185–197, 1992.

[18] Vijayan Sugumaran, Sooyong Park, and Kyo C. Kang. Introduction to the special issue on software product line engineering. *Communications ACM*, 49(12):28–32, 2006.

[19] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *2nd IEEE/IFIP Working Conference on Software Architecture (WICSA)*, pages 45–54, 2001.