Trabalho de Conclusão de Curso

Uma Ferramenta para Interfaces Emergentes em Linhas de Produto de Software

Társis Wanderley Tolêdo
tarsiswt@gmail.br

Orientador:
Márcio de Medeiros Ribeiro

Maceió, fevereiro de 2011

Társis Wanderley Tolêdo

Uma Ferramenta para Interfaces Emergentes em Linhas de Produto de Software

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Márcio de Medeiros Ribeiro

Maceió, fevereiro de 2011

Monografia apresentada pelo aluno Társis Wanderley Toledo como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Márcio de Medeiros Ribeiro - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Patrick Henrique Brito - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Leopoldo Motta Teixeira - Examinador
Centro de Informática
Universidade Federal de Pernambuco

Maceió, fevereiro de 2011

Agradecimentos

Este trabalho é o resultado de uma longa caminhada, na qual recebi incentivo para continuar de inúmeras pessoas em muitos momentos. Agradeço sinceramente a todos aqueles que de alguma forma contribuiram para tal caminhada que ainda está longe de se encerrar.

Em especial à minha família e meus pais, pelo onipresente apoio e compreensão.

Ao professor e orientador Márcio Ribeiro, pelo estímulo a construção deste trabalho e pelas e frequentes sugestões e ideias, sem as quais este trabalho não teria sido concretizado.

A todos os professores do Instituto de Computação pelo esforço considerável na melhoria desta instituição.

Aos amigos com quem tive a oportunidade de dividir tantos bons momentos.

Resumo

Linhas de Produto de Software (LPS) são frequentemente implementadas com compilação condicional, o que inclui o uso de diretivas como `#ifdef` e `#endif`. Estas diretivas poluem e ofuscam o código além de dificultar a separação de interesses. *Virtual Separation of Concerns* (VSoC), ou Separação Virtual de Interesses, é uma abordagem ferramental que tenta reduzir as dificuldades que envolvem implementar LPS com compilação condicional. VSoC permite que o desenvolvedor esconda o código fonte de determinadas *features* para concentrar seu esforço durante tarefas de manutenção em outras. Entretanto, ao esconder o código fonte de algumas *features*, o desenvolvedor pode inconscientemente introduzir erros nestas *features*, um reflexo da falta de modularização. Para atacar este problema, pesquisadores recentemente propuseram o conceito de *Emergent Interfaces*, ou Interfaces Emergentes, uma abordagem também ferramental que permite detectar dependências entre *features* por elementos compartilhados. Este trabalho apresenta uma implementação para interfaces emergentes como um *plug-in* para o IDE Eclipse, bem como uma avaliação desta implementação em cenários de manutenção de sistemas reais.

Conteúdo

Lista de Figuras

Lista de Códigos

Lista de Algoritmos

Lista de Tabelas

Capítulo 1

Introdução

Uma Linha de Produto de Software (LPS) consiste de sistemas de software que compartilham um conjunto gerenciado de funcionalidades que satisfazem necessidades específicas de um determinado segmento de mercado e são desenvolvidos a partir de um núcleo comum de artefatos [6]. É possível, portanto, reutilizar estes artefatos de diferentes maneiras para compor produtos de acordo com diferentes requisitos. O termo *feature* é utilizado para representar variabilidade [16], ou seja, uma *feature* é uma unidade semântica que diferencia e/ou introduz características em um produto. O conjunto de *features* e restrições entre elas é geralmente representado por um *feature model* [10].

Features são frequentemente implementadas usando-se compilação condicional e pré-processadores [11]. As diretivas de pré-processadores, como `#ifdef` e `#endif`, são utilizadas para demarcar trechos de código que pertencem a diferentes *features*. Apesar de amplamente utilizados, pré-processadores possuem desvantagens, como ofuscamento, poluição do código e falta de separação de interesses [23]. A Separação Virtual de Interesses, do inglês *Virtual Separation of Concerns* (VSoC) [11], surgiu como uma proposta para apaziguar as desvantagens inerentes ao uso de pré-processadores. A VSoC permite ao desenvolvedor esconder o código de determinadas *features*, para que possa se concentrar melhor na manutenção e desenvolvimento de outras.

Embora seja útil permitir que o desenvolvedor visualize *features* individualmente, a abordagem não modulariza as *features* de modo a apoiar o desenvolvimento/manutenção [17], já que o desenvolvedor pode, inconscientemente, introduzir um erro em uma *feature* que está escondida ao realizar alguma modificação em elementos que são compartilhados por elas.

Para atacar este problema de modularização de *features* inerente a VSoC, pesquisadores recentemente propuseram o conceito de Interfaces Emergentes (ou *Emergent Interfaces*) [19]. Este trabalho envolve o conceito de modularização emergente de *features* que tem como objetivo estabelecer contratos entre os elementos de código que compõem as *features*. A abordagem é dita ser emergente pois os contratos entre as *features* são calculados sob demanda, e não por uma estrutura rígida pré-definida. Este conceito pode ser usado para manter os benefícios envol-

vidos em esconder códigos de *features*, enquanto as dependências entre as *features* podem ser calculados sob demanda para desenvolvedor, ajudando-o a manter o código e as combinações de *features* seguras.

Este trabalho de conclusão de curso descreve a arquitetura e implementação de um *plug-in* para o IDE Eclipse[1] que concretiza os conceitos de interfaces emergentes, sendo esta a principal contribuição deste trabalho. Como a ferramenta precisa analisar código de vários produtos (e não mais de um único), faz-se necessário estender análises de fluxo de dados, de modo que estas contenham informações acerca de *features*. Assim sendo, para computar as dependências entre *features*, desenvolveu-se uma técnica para realizar análises de fluxo de dados sensível a *features*, bem como uma abordagem para interpretar os resultados de tais análises. A partir de tais resultados, computa-se as interfaces emergentes que, por sua vez, ajudam desenvolvedores durante tarefas de manutenção de LPS.

Para avaliar a ferramenta proposta, adotou-se dois cenários retirados de LPS reais, onde verificou-se a coerência das interfaces geradas e o tempo médio para calculá-las e exibí-las. Tomando por base os cenários aqui utilizados, verifica-se que este tempo é aceitável. Ou seja, em tarefas de manutenção semelhantes às analisadas neste trabalho, a espera da computação das interfaces não interfere na produtividade dos desenvolvedores.

O restante deste trabalho está organizado da seguinte maneira:

- No Capítulo 2, apresenta-se o embasamento teórico necessário ao entendimento deste trabalho;
- A ferramenta aqui proposta é detalhada no Capítulo 3. Primeiramente, aborda-se a sua arquitetura; a implementação de análises de fluxo de dados sensíveis a *features* é apresentada em seguida; e, por fim, o processo de computação e exibição das interfaces emergentes é descrito;
- No Capítulo 4, ilustra-se a avaliação em termos de cenários de uso da ferramenta bem como uma simples análise considerando o desempenho da mesma;
- Os trabalhos relacionados à ferramenta proposta são discutidos no Capítulo 5;
- Por fim, as considerações finais bem como os trabalhos futuros são abordados no Capítulo 6.

[1] http://www.eclipse.org/

Capítulo 2

Embasamento Teórico

Este capítulo tem como objetivo enunciar os temas necessários para a compreensão deste trabalho.

2.1 Linhas de Produto de Software

Uma Linha de Produto de Software (LPS) consiste de sistemas de software que compartilham um conjunto gerenciado de funcionalidades que satisfazem necessidades específicas de um determinado segmento de mercado e são desenvolvidos a partir de um núcleo comum de artefatos [6]. Desta forma, os produtos que podem ser derivados de uma LPS possuem o mesmo cerne e as diferenças entre eles são geridas como elementos de variabilidade.

Em resumo, são 3 os principais benefícios adquiridos ao adotar uma abordagem de LPS [18]:

- *Redução do custo de desenvolvimento*: quando artefatos são reutilizados em diversos sistemas, isto implica uma redução no custo de cada sistema, já que não é necessário implementar os artefatos do início;
- *Melhoria de qualidade*: os artefatos de uma LPS são utilizados em muitos produtos e por isso são testados e revisados muitas vezes, o que por sua vez aumenta a chance de detectar falhas, consequentemente aumentando a qualidade individual dos recursos e da LPS de uma maneira geral;
- *Diminuição do time-to-market*: inicialmente, o *time-to-market* de uma LPS é alto, isto porque os recursos que irão compor os produtos precisam ser desenvolvidos primeiro. Mais tarde, o *time-to-market* é reduzido pois os recursos passam a ser reutilizados para diversos produtos.

Estas vantagens, em sua grande parte oriundas do reuso, no entanto, não são gratuitas: há um esforço considerável envolvido em construir uma infra-estrutura comum a uma família

de produtos e gerenciar todas as suas variações e combinações. Todavia, este investimento adiantado torna o esforço aceitável usualmente a partir do 3º produto da LPS [16].

Na maioria das abordagens de LPS, o termo *feature* é utilizado como um conceito básico de variabilidade [16], ou seja, uma *feature* é uma unidade semântica que diferencia e/ou introduz características no produto.

Uma *configuração*, a seu turno, é um conjunto de *features* habilitadas para um determinado produto. Assim sendo, dado um conjunto de *features* $\mathbb{F} = \{A, B, C, D\}$, então um conjunto, $\mathbb{C} \subseteq \mathbb{F} = \{A, C\}$ corresponde a uma configuração onde as *features* A e C estão habilitadas. Neste caso, diz-se que um produto possui a configuração $\mathbb{C}$.

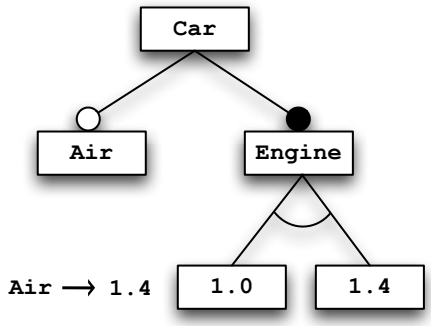


Figura 2.1: *Feature model* de uma linha de produtos de carros.

Configurações usualmente respeitam e são restringidas por um *feature model* [10]. Conceitualmente, um *feature model* é composto por um conjunto de definições de *features* e outro de proposições lógicas que podem efetivamente diminuir o conjunto final de produtos válidos. A Figura 2.1 ilustra um *feature model* de uma LPS de carros. O elemento `Car` representa a LPS. O nó `Engine` é uma funcionalidade *obrigatória* do produto, que é denotada pelo círculo preenchido. O `Engine` e pode ser composto por apenas uma das *features*: `1.4` ou `1.0`. Diz-se que estas são *alternativas*, e por sua vez, são denotadas pelo arco entre elas. A *feature* `Air` é opcional, sua notação é o círculo não preenchido. Note-se ainda que este *feature model* é restringido pela regra $\texttt{Air} \rightarrow \texttt{1.4}$.

Deste modo, o *feature model* da Figura 2.1 pode ser expresso da seguinte maneira:

$$\texttt{Car} \wedge \texttt{Engine} \wedge (\texttt{1.0} \leftrightarrow \neg \texttt{1.4}) \wedge (\texttt{Air} \rightarrow \texttt{1.4})$$

Em outras palavras, a presença da *feature* `Air` deve *implicar* que a *feature* `1.4` também esteja presente, ao passo que a *feature* `1.0` deve excluir a presença da `1.4`, e vice-versa.

2.1.1 Abordagens Para Implementação de LPS

É comum classificar implementações de linhas de produto como sendo *composicionais* ou *anotativas*. Na primeira, as unidades que compõem a LPS são construídas em módulos separados e então são compostas para formar um produto. São exemplos a orientação a aspectos

[12], que recentemente tem recebido uma grande atenção de pesquisidores, AHEAD [4] um paradigma para desenvolvimento incremental de *features*, separação multi-dimensional de interesses [25], *mixin layers* [22] para refinamentos em larga escala ou ainda por componentes de software [24].

Por sua vez, a abordagem anotativa consiste em demarcar porções de interesse para a linha de produto afim de estratificar sua estrutura. Em geral, as porções que foram demarcadas são selecionadas ou não para a construção de um produto. O exemplo mais comum de técnica anotativa é o uso de diretivas similares ao pré-processador C/C++ para delimitar o código fonte, como o `#ifdef` (linha 3) e o `#endif` (linha 5) mostrados no Código 2.1. Trata-se de um trecho do código extraído do *Lampiro*[1], uma LPS que implementa o protocolo XMPP de troca de mensagens em J2ME. Neste caso, a linha 4 está habilitada pois o pré-processor encontrou a definição da variável de pré-processador[2] `BT_PLAIN_SOCKET`. Caso a variável `BT_PLAIN_SOCKET` não estivesse sido definida, então o código da linha 4 teria sido comentado ou removido.

Existem ainda outras propostas como a programação explícita [5], que permite a introdução de novos vocabulários no código fonte a fim de controlar o espalhamento de código de *features*. *Gears* [14] provê modelos para a adoção de customização em massa de software e XVCL [9], uma linguagem para configuração de variantes.

```
1 xmlStream = new SocketStream();
2 ...
3 // #ifdef BT_PLAIN_SOCKET
4 connection = new SocketChannel("socket://" + cfg.getProperty(
    Config.CONNECTING_SERVER), xmlStream);
5 // #endif
6 ...
7 ((SocketChannel) connection).KEEP_ALIVE = Long.parseLong(cfg.
    getProperty(Config.KEEP_ALIVE));
```

Código 2.1: Trecho de código da LPS Lampiro implementado com pré-processadores.

2.1.2 Níveis de Granularidade

De maneira geral, as técnicas composicionais permitem apenas níveis de granularidade mais grossos quando comparado com as técnicas anotativas. Em algumas das técnicas composicionais citadas na Seção 2.1.1, são definidos pontos de extensão no código, como classes, métodos, *traits* e campos que são utilizados para estender ou introduzir pontos extensíveis através

[1] http://www.lampiro.bluendo.com

[2] O local onde a variável de pré-processor é definida pode variar entre os pré-processadores. Em geral elas estão presentes em algum arquivo de configuração.

da sobrescrita ou sobrecarga de métodos. Isto se torna um problema grave quando *features* que devem ser introduzidas na LPS exigem um fino nível de granularidade para ser implementada de maneira simples e direta. As principais dificuldades de extensão inerentes aos métodos composicionais são [11, p.313]:

1. *De instrução*: na sua maioria, as abordagens não permitem que sejam inseridos instruções no meio de um método. Por exemplo, sincronizar uma parte de um código Java;

2. *De expressão*: Modificar o conteúdo de uma expressão também é bastante problemático quando se utiliza a composição, *e.g.* modificar uma expressão booleana dentro de uma instrução condicional;

3. *De assinatura*: Alterar a assinatura de um método ou procedimento pode ser uma necessidade comum ao construir/manter uma linha de produto. A exemplo, um dos parâmetros a ser passado para um método pertence apenas a uma feature, e portanto a sua existência ou não em um produto deve ser controlada.

Em contraste, técnicas anotativas costumam permitir uma mais fina granularidade ao delimitar o código fonte. Pré-processadores, por exemplo, não sofrem de nenhuma das dificuldades listadas acima, de modo que seu nível de granularidade é praticamente arbitrário. Entretanto, técnicas anotativas explícitas são conhecidas por ofuscar o código, reduzindo assim sua legibilidade [23]. Outros pesquisadores [11] mostram que o uso indiscriminado de diretivas podem inserir erros sutis que por sua vez podem aumentar o custo de manutenção/desenvolvimento da LPS.

```
1  static in __rep_queue_filedone(dbenc, rep, rfp)
2    DB_ENC *dbenv;
3    REP *env
4    __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6    COMPQUIET(rep, NULL);
7    COMPQUIET(rfp, NULL);
8    return (__db_no_queue_am(dbenv));
9  #else
10   db_pgno_t first, last;
11   u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
```

```
16    ...
17    }
18 #endif
```

Código 2.2: Erro latente de sintaxe com pré-processador.

O Código 2.2 mostra um erro sutil que, embora introduzido deliberadamente, acontece com uma certa frequência [2]. A chaves aberta na linha 4 só é fechada na linha 17 quando a *feature* `HAVE_QUEUE` está habilitada. O erro se torna mais difícil ainda de ser detectado já que ele não necessariamente estará presente em um produto.

Vale acrescer que é difícil conseguir modularidade anotativamente. Motivados pelos problemas decorrentes de ambas as metodologias, anotativas e composicionais, pesquisadores [11] propuseram uma nova abordagem que possui a semântica de pré-processadores, e portanto pode ser considerada anotativa, mas evita a poluição de código através de anotações bem-comportadas e cores no lugar de diretivas, conforme ilustrado na próxima seção.

2.2 CIDE e Separação Virtual de Interesses

Proveniente da língua inglesa, *Colored Integrated Development Environment* ou Ambiente Colorido Integrado de Desenvolvimento – CIDE[3], é uma abordagem anotativa ferramental para implementação de LPS que tenta mitigar os problemas de granularidade inerentes às abordagens composicionais, ao passo em que soluciona em parte aqueles relacionados às outras abordagens anotativas, como explicitado na Seção 2.1.2. Implementada como um *plug-in* para o IDE Eclipse, o CIDE faz uso editores de código fonte disponíveis para permitir que trechos de tal código possam ser associados a *features* através da sua cor de fundo. Cada *feature* recebe uma cor única, entretanto, é permitido que um trecho de código seja associado a mais de uma *feature*. Nestes casos, as cores se misturam para formar uma nova coloração, de modo que, *e.g.*, a combinação entre vermelho e azul é mostrada como roxo. A Figura 2.2 retirada do site do CIDE ilustra a ferramenta em ação.

Internamente, as informações de cores são associadas apenas aos nós da AST, *Abstract Syntax Tree* ou Árvore Sintática Abstrata. Note-se que vírgulas, chaves etc. em geral não estão presentes na AST, não sendo possível associar cores a esses elementos. Em outras palavras, é possível apenas associar features aos elementos estruturais do código. Embora possua uma granularidade mais grossa em relação aos pré-processadores, as informações sobre as cores estão disponíveis e associadas aos nós da AST, o que dá espaço para a implementação de ferramentas que se utilizem desta informação, como é o caso deste trabalho.

A representação e arquitetura da ferramenta permitem ainda outras funcionalidades, como geração automática de variantes, que permite que o código associado a uma configuração seja

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/cide/

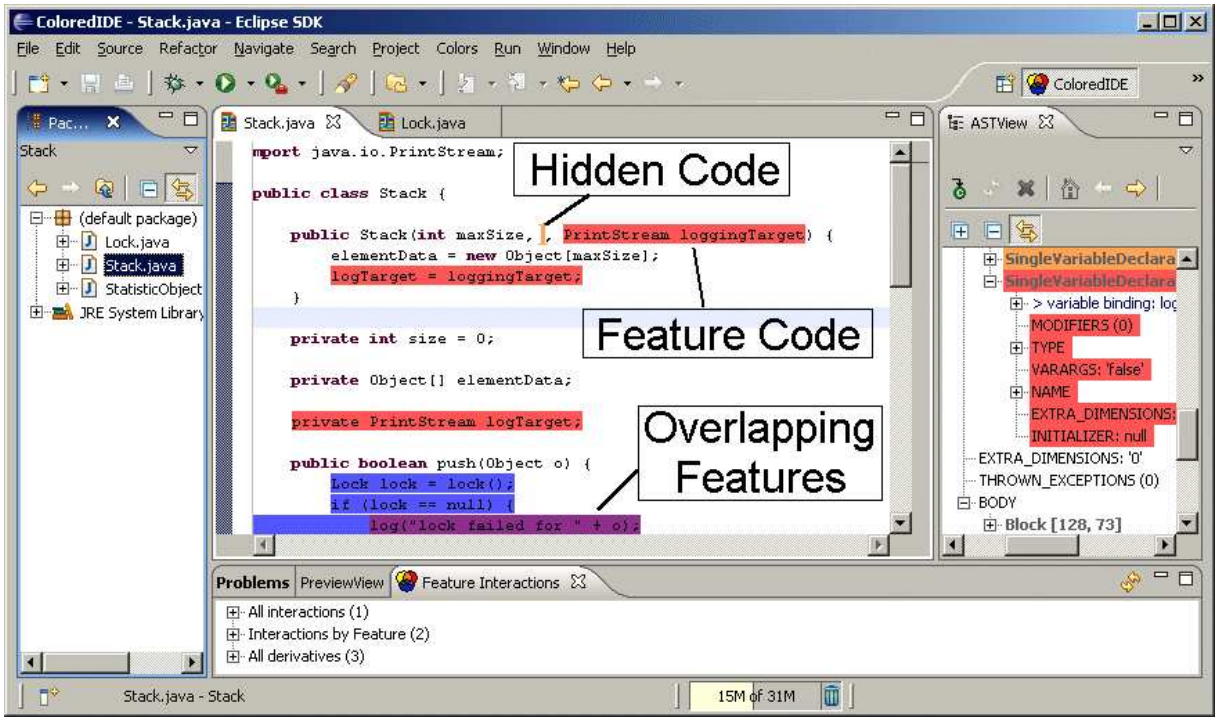


Figura 2.2: CIDE no IDE Eclipse.

exportado como um novo projeto no próprio Eclipse, visualização da AST colorida, e uma representação visual do *feature-model*. É possível ainda esconder o código de determinadas *features*, separando assim virtualmente o código ainda presente, permitindo que o desenvolvedor possa focar na manutenção de uma determinada feature. É daí que surge o termo *Virtual Separation of Concerns* (VSoC), ou separação virtual de interesses [11]. O Código 2.3 ilustra como seria o código fonte mostrado no Código 2.1 só que desta vez utilizando o CIDE.

```
1 xmlStream = new SocketStream();
2 ...
3 
4 ...
5 ((SocketChannel) connection).KEEP_ALIVE = Long.parseLong(cfg.
    getProperty(Config.KEEP_ALIVE));
```

Código 2.3: Trecho de código do Lampiro com o CIDE.

A intenção da VSoC é permitir que o desenvolvedor tire de seu caminho informação irrelevante à sua tarefa, para que possa melhor se concentrar na *feature* que ele está mantendo no momento. Entretanto, ao esconder trechos de código que de alguma forma estão conectados ou possuem alguma dependência, o desenvolvedor pode, inconscientemente, introduzir erros. Suponha, por exemplo, ainda no contexto do Código 2.3, que o desenvolvedor deseja mudar do tipo `SocketStream` para um outro sintaticamente compatível. Mas se a semântica deste

novo tipo não for coerente com a antiga, o desenvolvedor terá introduzido um erro por não estar vendo que a variável `xmlStream` é utilizada por uma *feature* que está virtualmente separada.

2.3 Interfaces Emergentes

Numa tentativa de alcançar modularidade com VSoC, propôs-se o conceito de Interfaces Emergentes [20] que permitem estabelecer contratos entre trechos de código de diferentes *features* sob demanda, sem uma estrutura rígida pré-definida.

O uso é simples: o usuário seleciona um trecho de código de onde deseja fazer a manutenção e uma ferramenta realiza análises de fluxo de dados afim de identificar os elementos e os contratos existentes entre o código possivelmente escondido e o selecionado. A ideia é abstrata o suficiente de modo é possível adaptá-la para o uso com o CIDE. Exemplificando, suponha que um desenvolvedor deseja alterar o tipo da variável `xmlStream` do Código 2.3. Antes de fazê-lo, ele invocaria a interface emergente sinalizando a linha 1 como a seleção. O resultado seria uma mensagem como a da Figura 2.3 alertando-o de que a variável definida naquela linha é *usada* na linha 3 e está colorida com a *feature* `BT_PLAIN_SOCKET`. Assim sendo, ele avaliaria melhor antes de fazer a modificação, podendo por exemplo analisar a *feature* `BT_PLAIN_SOCKET` com mais profundidade para ter certeza de que a manutenção não causará nenhum problema a *feature*.

Provides `xmlStream = new SocketStream()` to line 3 [`BT_PLAIN_SOCKET`]

Figura 2.3: Mensagem da interface emergente.

No entanto, a simplicidade da ideia esconde uma dificuldade: realizar análises de fluxo de dados sensíveis a *features*. A título de exemplo, suponha duas *features* alternativas: *A* e *B*. Uma análise ordinária poderia mostrar que uma modificação em uma das *features* poderia causar impacto na outra. No entanto, quando vista sob a perspectiva de LPS, este impacto é inexistente, haja vista que o código fonte de ambas as *features* nunca estarão presentes em um mesmo produto. Isto quer dizer que as análises sensíveis a *features* devem ter o fluxo desviado de acordo com o contexto.

Além disso, ao analisar uma LPS, deve-se ter em mente que esta deve ser tratada como uma coleção de produtos, e não uma unidade. Portanto, a quantidade de informação que pode ser obtida através de análises é proporcional ao tamanho da família de produtos.

Afim de explicar como este trabalho eleva análises ordinárias para outras sensíveis a *features*, uma pequena introdução acerca da teoria de Análise de Fluxo de Dados encontra-se na próxima seção.

2.4 Análise de Fluxo de Dados

De modo conceitual, a Análise de Fluxo de Dados (*Data-Flow Analisis* - DFA) [13] é composta por 3 elementos:

1. *Control Flow Graph* (CFG): grafo de fluxo de controle, sob o qual a análise será executada;
2. *Lattice*: representa o conteúdo das análises;
3. Funções de transferência: controlam o conteúdo dos *lattices* simulando a execução.

Um CFG é um grafo direcionado que representa o fluxo de controle de um programa onde uma análise de fluxo pode ser executada (ver Figura 2.4(b)). Os nós representam as instruções, enquanto as arestas representam o fluxo, ambos de acordo com a linguagem de programação. Deste modo, o CFG é construído a partir da estrutura sintática do programa. Além disso, se uma análise leva em conta os nós de chamadas de funções e procedimentos, então a análise é dita interprocedural ou, em caso contrário, será intraprocedural.

(a) Um programa... (b) e seu CFG.

Figura 2.4: Um pequeno programa e seu CFG correspondente.

As informações calculadas são armazenadas em *lattices* para cada ponto do CFG. Isto quer dizer que teremos informação atrelada a cada nó do grafo. Uma maneira conveniente de se representar um *lattice* é através do diagrama de *Hasse*, como mostra a Figura 2.5.

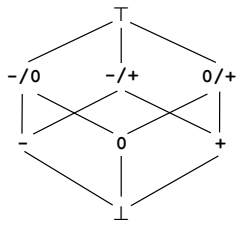


Figura 2.5: *Lattice* para a análise de sinal.

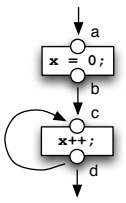
As funções de transferências simulam a execução das instruções presentes no CFG. Na execução de uma função de transferência, um elemento do *lattice*, ℓ, *flui* através da função, $f_S(\ell)$, que irá simular a execução da instrução S, gerando um outro elemento ℓ', como mostra a Figura 2.6. Abaixo encontram-se as funções para analisar o pequeno programa representado pelo CFG da Figura 2.4(b):

$$f_{\texttt{x=0}}(\ell) = \texttt{0} \qquad\qquad f_{\texttt{x++}}(\ell) = \begin{cases} \top & \ell \in \{\texttt{-/+}, \texttt{-/0}, \top\} \\ \texttt{+} & \ell \in \{\texttt{0}, \texttt{+}, \texttt{0/+}\} \\ \texttt{-/0} & \ell = \texttt{-} \\ \bot & \ell = \bot \end{cases}$$

$$\begin{array}{c} \ell \\ \downarrow \\ \boxed{S} \quad \ell' = f_S(\ell) \\ \downarrow \\ \ell' \end{array}$$

Figura 2.6: Efeito da função de transferência f_S.

A função $f_{\texttt{x=0}}$ indica a atribuição da variável à constante `0`, ou seja, a variável sempre será `0` quando a instrução for executada. Já a função $f_{\texttt{x++}}$ deve simular o operador de incremento "++". Deste modo, se a variável continha um valor negativo, então ela poderá conter o valor `0` ou outro valor negativo (`-/0`). A mesma ideia se aplica para os outros componentes desta função.



(a) CFG

$$T\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} \bot \\ f_{\texttt{x=0}}(a) \\ b \sqcup d \\ f_{\texttt{x++}}(c) \end{pmatrix}$$

(b) Funções de transferência agrupadas.

a	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
b	$\bot$	0	0	0	0	0
c	$\bot$	$\bot$	0	0	0/+	0/+
d	$\bot$	$\bot$	$\bot$	+	+	+
	$T^0(\bot)$	$T^1(\bot)$	$T^2(\bot)$	$T^3(\bot)$	$T^4(\bot)$=$T^5(\bot)$	

(c) Iterações de *fixed-point*

Figura 2.7: Um programa é transformado em um CFG e nele é marcado um conjunto de pontos (a, b, c, e d); as funções de transferência são agrupadas e utilizadas para calcular o *least fixed point* computando $T^i(\bot)$ para i crescente até que nada mude em T.

Com esses 3 elementos estabelecidos, para realizar a análise é preciso computar o *least fixed-point* através de um algoritmo, que é a solução das equações e o resultado da análise, como mostra a Figura 2.7. O *least fixed-point*, neste exemplo, é encontrado na 4ª iteração $T^4 = T^5$. O resultado encontrado é:

$$(\bot, 0, 0/+, +)$$

O que significa que para o ponto *a* não se sabe nada sobre o conteúdo da variável x, no ponto *b* x é definitivamente 0, em *c* o valor de x é 0 ou +, ou seja, $x \geq 0$, e por último, para o ponto *d* o valor de x é +.

A fundamentação matemática destes conceitos vai além do escopo deste trabalho. Um estudo mais aprofundado pode ser encontrado em [13].

Existem diversas ferramentas que dão suporte à análise estática em Java, mas uma delas se destaca por sua robustez e extensibilidade, como ilustra a seção seguinte.

2.5 SOOT

O *framework* SOOT foi originalmente produzido pelo *Sable Research Group*[4] na universidade de McGill como uma infra-estrutura de compilador para analisar e transformar *bytecode* Java [26]. Com o passar do tempo, o *framework* foi ganhando novas funcionalidades e melhorias tais como decompilação, visualização e um grande arsenal de análises e otimizações, e pode ser utilizado como uma ferramenta de linha de comando ou programaticamente. Além disto, o *framework* utiliza representações intermediárias do código fonte, que podem ser mais próximas ao *bytecode* ou ao código fonte.

O código-fonte ou *bytecode* precisa ser transformado para a principal representação intermediária, chamada *Jimple*. Trata-se de uma representação de 3 endereços para executar análises. Os autores argumentam que é mais prático executar análises no Jimple em vez do *bytecode*, pois possui tipos, abstrai a pilha de execução e as variáveis implícitas e reduz significativamente o número de instruções.

2.5.1 Análise de Fluxo de Dados com SOOT

As classes que são carregadas para o SOOT são representadas pela classe `SootClass` e são agrupadas na `Scene`, um ambiente onde as análises acontecem. As análises intraprocedurais são executadas método a método, onde cada método de cada classe é representado por um `SootMethod` e um corpo (`Body`). Cada instrução em qualquer uma das representações intermediárias são implementadoras da interface `Unit`.

CFGs são representados pela interface `DirectedGraph` e são facilmente construídos a partir de um `Body`. Já os *lattices* são representados pela interface `FlowSet`. O último elemento constituinte de DFA, funções de transferência, está na classe `ForwardFlowAnalysis` que estende a classe `AbstractFlowAnalysis`. `ForwardFlowAnalysis` é uma implementação padrão que provê um algoritmo de computação de *least fixed-point* bastante eficiente.

[4] http://www.sable.mcgill.ca/

O contrato de implementação de uma análise comum estabelece que 5 métodos sejam implementados, como mostra o Código 2.4. O *framework* apenas transfere o controle —ao executar uma análise— para esses métodos mostrados no contrato em questão.

```
1  protected void copy(FlowSet source, FlowSet dest);
2
3  protected void merge(FlowSet source1, FlowSet source2,
       FlowSet dest);
4
5  protected FlowSet entryInitialFlow();
6
7  protected FlowSet newInitialFlow();
8
9  protected void flowThrough(FlowSet source, Unit unit,
       FlowSet dest);
```

Código 2.4: Contrato para implementar análise.

De uma maneira geral, os `FlowSets` `source` são passados como argumento para que o desenvolvedor implemente o fluxo deste *lattice* e produza um novo `FlowSet` em diversos momentos na execução da análise.

A função `copy` na linha 1 é chamada para a operação de cópia entre os `FlowSets` de diferentes pontos do programa, como por exemplo, entre os pontos *a* e *b* da Figura 2.7(a).

O controle é transferido para a linha 3 para que análise defina a operação de *confluência* entre dois `FlowSets`, como a confluência entre os pontos *d* e *c* também da Figura 2.7(a).

As funções nas linhas 5 e 7 permitem que o desenvolvedor defina quais serão os valores assinalados inicialmente ao ponto de entrada e ao restante dos pontos no CFG, respectivamente.

Por último, na linha 9, tem-se a chamada para a função que representa a função de transferência. Nela, o desenvolvedor deve implementar uma rotina que tome uma decisão de manipular os *lattices* de destino `dest` de acordo com o tipo e conteúdo da `unit` que está sendo processada no momento.

Capítulo 3

A Ferramenta CIDE EI

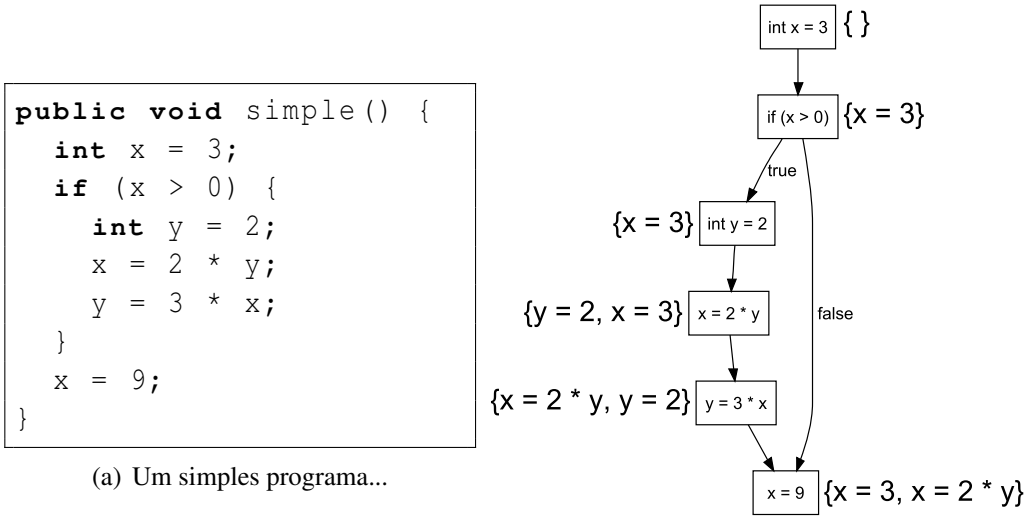
Este capítulo descreve a arquitetura e implementação da ferramenta **CIDE EI**, principal contribuição deste trabalho. A ferramenta é capaz de capturar dependências entre elementos de *features* diferentes de um mesmo método de código fonte Java e exibir estas dependências através de interfaces emergentes, graças a uma implementação de análise de fluxo de dados sensível a *features*.

As análises de fluxo de dados são comumente utilizadas para capturar erros e dependências em código fonte, simulando a execução estaticamente e coletando informações afim de auxiliar os desenvolvedores [27]. Um exemplo clássico é a análise de *reaching definitions* (*vide* Figura 3.1), expressão em inglês que significa *definições alcançantes*, que computa, para cada nó de um CFG, quais definições de variáveis podem alcançar aquele ponto. A Figura 3.1(b) ilustra que no CFG do programa da Figura 3.1(a) o nó `y = 3 * x` é alcançado somente pelas definições `{x = 2 * y, y = 2}`, isto porque a definição `x = 3` foi substituída pela `x = 2 * y`.

Entretanto, o código fonte de uma LPS não consiste de apenas um produto, mas sim de uma família. Isto quer dizer que trechos de código fonte podem ou não estar presentes em um dos possíveis produtos. Adicionalmente, o número de possíveis produtos em uma LPS com *n features* cujo *feature model* não possui nenhuma restrição é de 2^n produtos. Esta complexidade combinatória torna o problema humana e computacionalmente intratáveis [15] para um n suficientemente grande.

Para justificar a necessidade de análises sensíveis a *features*, suponha o código fonte da Figura 3.2 que é o mesmo da Figura 3.1(a), só que com *features* definidas utilizando o CIDE. É fácil perceber que a definição `x = 3` não alcança a instrução `y = 3 * x` quando a *feature* amarela (*A*) está presente, pois a variável `x` será redefinida pela instrução `x = 2 * y`. Uma consequência disso é que a definição `x = 3` não necessariamente alcança a instrução `y = 3 * x`. Ou seja, isto depende da configuração e do produto que está sendo analisado. É preciso, portanto, estender as análises para que possam ser capazes de capturar informações e dependências dentro de uma família de sistemas, e não mais sobre um único produto.

As informações coletadas com as análises sensíveis as *features* são utilizadas para imple-



(a) Um simples programa...

(b) e seu resultado para análise de *reaching definitions*.

Figura 3.1: Resultado da análise *reaching definitions* anotada em um CFG.

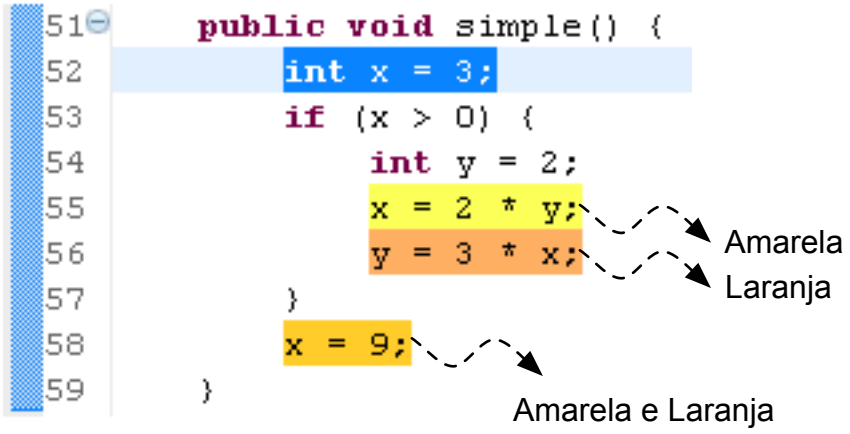


Figura 3.2: Trecho de código colorido com o CIDE.

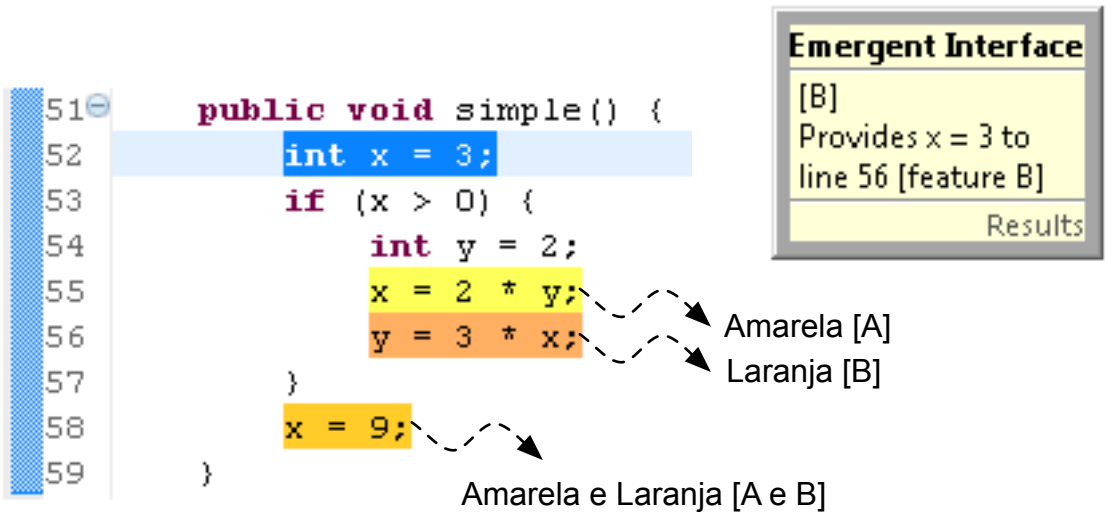


Figura 3.3: Interface emergente gerada pelo CIDE EI.

mentar o conceito de interface emergente, exibindo para o usuário, que neste caso é o desenvolvedor da LPS, informações relevantes relacionadas às outras *features*. O resultado final para o exemplo colorido pode ser visto na Figura 3.3. Assim, se o desenvolvedor estiver realizando uma manutenção que envolve a declaração `int x = 3;`, a ferramenta pode alertá-lo (de maneira sensível a *features*) sobre outros pontos do código, como quebras de fluxo, uso de definições etc., fazendo com que ele tenha uma melhor perspectiva do impacto que as mudanças que ele deseja fazer terão sobre a LPS. A mensagem que aparece na figura em questão como um *pop-up* é construída a partir da análise de *reaching definitions* e contém as seguintes informações:

- A primeira linha do *pop-up*, [B], identifica a configuração que possivelmente seria afetada pela manutenção;
- Logo abaixo desta linha, encontra-se a mensagem propriamente dita, *Provides x = 3 to line 56*, que significa que a atribuição `int x = 3;` alcança a linha 56;
- E a continuação desta última linha, *[feature B]*, significa que o uso da variável ocorre na *feature B*.

Embora trate-se de um exemplo pequeno, os desenvolvedores poderão se beneficiar da informação proveniente da interface ao inspecionar códigos bem mais complexos ou com um maior número de dependências entre as *features*. Além disso, os desenvolvedores também não precisarão "quebrar" a separação virtual que o CIDE lhes provê para procurar manualmente estas dependências. Ainda é possível implementar com relativa facilidade outros tipos de análises capazes de providenciar mais informações para o desenvolvedor, ponderando sempre sobre a quantidade de informação a ser exibida na interface para evitar que o usuário seja sobrecarregado com ela.

3.1 Arquitetura

De uma maneira sucinta, a ferramenta proposta por este trabalho consiste de dois *plug-ins* para o IDE Eclipse e implementam análises de fluxo dados sensíveis a *features*, combinando um conjunto de técnicas e ferramentas já existentes e validadas para gerar interfaces emergentes de acordo com a seleção do usuário. A Figura 3.4 mostra a relação entre cada um dos elementos que compõem a ferramenta, onde a seta representa a relação "depende de".

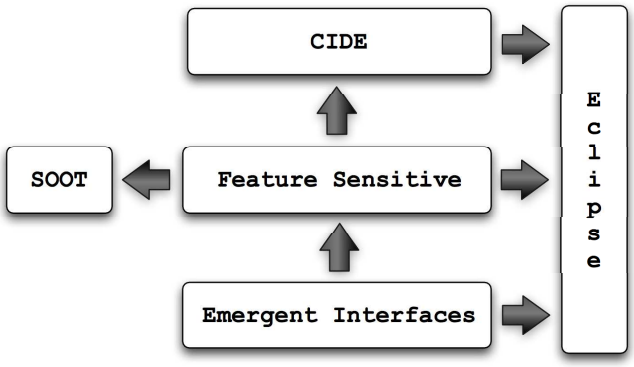


Figura 3.4: Arquitetura do CIDE EI.

O **CIDE**, explicado com mais detalhes na Seção 2.2, é um *plug-in* para o IDE **Eclipse** que implementa a abordagem VSoC. As informações de cores estão atreladas a AST de acordo com uma representação interna ao *plug-in* e disponíveis através de uma interface.

Denominou-se **Feature Sensitive** o componente que provê a implementação de análises sensíveis a *features* estendendo a estrutura de classes do *framework* **SOOT** (*vide* Seção 2.5). Este componente foi construído como *plug-in* para que fosse possível executar análises sensíveis a *features* em toda uma LPS através da interface gráfica do IDE. Adicionalmente é possível executar individualmente as análises em métodos arbitrários. Além disso, serve também como uma camada de separação entre o **CIDE** e o componente **Emergent Interfaces**.

A extensível estrutura de classes do **SOOT** permitiu uma acomodação suave das mudanças necessárias para tornar as análises sensíveis a *features*.

Por último, mas não menos importante, temos o componente **Emergent Interfaces**, que é responsável pelas seguintes tarefas:

- *Lidar com a seleção do usuário*: diferentemente do componente **Feature Sensitive**, são executadas as análises de acordo com o conteúdo da seleção e onde ela ocorreu;
- *Executar as análises*: as análises disponíveis no componente **Feature Sensitive** são executadas numa tentativa de capturar diversos tipos de dependências no método onde ocorreu a seleção;
- *Interpretar os resultados*: uma grande quantidade de informação é gerada pelas análises. Não seria prático exibir toda a informação para o usuário, pois este seria facilmente so-

brecarregado. É preciso, portanto, iterar sobre os resultados das análises afim de gerar um conteúdo concisa e amigável;

- *Exibir a interface*: a interface é exibida como um *pop-up* casual escondendo a complexidade dos resultados da análise do usuário.

Os componentes **Feature Sensitive** e **Emergent Interfaces** são detalhados nas Seções 3.2 e 3.3, respectivamente.

3.2 Análise de Fluxo de Dados Sensível a *Features*

Nesta seção será explicada a técnica e implementação desenvolvidas. Em primeiro lugar será explicada o conceito abstrato da técnica que permite a análise sensível a *features*, e em seguida é preciso aprofundar um pouco mais o entendimento sobre o *framework* SOOT para compreender a implementação de fato.

3.2.1 Teoria e Abstração

Para evitar que seja preciso construir cada um dos produtos de uma LPS e, assim, executar análises de fluxo de dados para cada um dos produtos, modificações em alguns dos elementos que compõem a análise de fluxo de dados (*vide* Seção 2.4) são feitas, de modo que as informações são calculadas para todos os produtos da LPS de uma só vez.

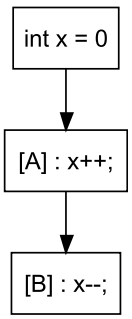


Figura 3.5: Um CFG instrumentado.

Com esta técnica, as análises são executadas para todas as possíveis configurações simultaneamente. Isto quer dizer que a função de transferência deve simular a execução daquela instrução para todas as configurações as quais a instrução pertence. Todavia, as informações que tornam possível decidir a simulação precisam ser tornadas explícitas. O lugar mais apropriado para atrelar estas informações é o CFG. A este processo dá-se o nome de *instrumentação do CFG*, que consiste em associar a cada nó o conjunto de *features* às quais o nó (instrução) pertence. A Figura 3.5 ilustra o CFG instrumentado do Código 3.1.

```
int x = 0;
x++; [A]
x--; [B]
```

Código 3.1: Uma pequena LPS anotada com duas *features*.

Desta forma, a função de transferência trabalha em cima de múltiplos *lattices*, e não mais somente em um. Definiu-se, então, um novo *lattice* denominado *lifted lattice* de forma que este contenha um conjunto de configurações, onde cada uma destas é associada a um *lattice*. Analogamente, denominou-se *lifted transfer function* as funções de transferência que operam sobre *lifted lattices*. Para exemplificar estes conceitos, suponha o trecho de código de uma LPS anotada com duas *features*, A e B, como mostra o Código 3.1.

Suponha ainda que a análise a ser executada neste programa é análise de sinal, mostrada como exemplo na Seção 2.4. São 3 os possíveis produtos desta linha:

$\mathbb{C} = \{A\}$:

```
int x = 0;
x++;
```

$\mathbb{C} = \{B\}$:

```
int x = 0;
x--;
```

$\mathbb{C} = \{A,B\}$:

```
int x = 0;
x++;
x--;
```

Então um exemplo de um *lifted lattice* neste contexto é:

$$(\{A\} \mapsto +, \{B\} \mapsto -, \{A,B\} \mapsto 0/+)$$

que corresponde dizer, segundo o *lattice*, que para a configuração $c = \{A\}$ a variável x é *positiva*, ao passo que para a configuração $\mathbb{C} = \{B\}$, é *negativa* e finalmente, para a configuração $\mathbb{C} = \{A,B\}$, x é *zero ou positiva*.

Ainda neste mesmo contexto, o efeito de uma *lifted transfer function* sobre uma instrução é:

$$(\{A\} \mapsto 0, \{B\} \mapsto 0, \{A,B\} \mapsto 0)$$
$$\downarrow$$
$$[\![A]\!]\text{: x++;}$$
$$\downarrow$$
$$(\{A\} \mapsto +, \{B\} \mapsto 0, \{A,B\} \mapsto +)$$

Ambos os *lattices* das configurações $\{A\}$ e $\{A,B\}$ foram afetados pela função de transferência $f_{\text{x++}}$ pois as duas contêm $[\![\text{A}]\!]$ em sua respectiva configuração de *features* habilitadas, enquanto o da configuração $\{B\}$ permanece inalterado, como era de esperar, já que a configuração $\{B\}$ não contém a *feature A* habilitada e, portanto, não possui a instrução x++.

Em poucas palavras, para realizar análises de fluxo de dados sensíveis a *features* é preciso

3 (três) elementos:

1. *CFG instrumentado*: um CFG onde cada um dos nós é associado a um conjunto de *features* aos quais o nó pertence;
2. *Lifted lattice*: encapsula um *lattice* para cada possível configuração;
3. *Lifed transfer functions*: funções de transferências que operam em *lifted lattices*.

3.2.2 Prática e Implementação

Na Seção 2.5 foi mostrado como o SOOT dispõe internamente, através de representações intermediárias e abstrações, os elementos necessários para transformar e analisar código fonte e *bytecode* Java. O SOOT provê uma API para construir CFGs de métodos, representados pela interface `DirectedGraph`. Um `DirectedGraph` é em geral composto de `Unit`, interface que representa uma instrução nas representações intermediários do *framework*, e permite iterar sobre estas `Unit`s como em uma `Collection`.

De posse de mecanismos que permitem instanciar e iterar sobre um CFG, é preciso agora atrelar as informações a ele, em outras palavras, instrumentá-lo, de modo que cada `Unit` contenha informações sobre a quais *features* elas pertencem. Felizmente, a cada instrução no SOOT (`Unit`), é possível adicionar `Tag`s, classes que servem de contêiner para informações.

O instrumentador de CFG itera sobre as `Unit`s, e associa uma `Tag`, chamada `FeatureTag`, a cada `Unit`. A `FeatureTag` contém uma `Collection` utilizada para armazenar os nomes das *features* das quais a `Unit` faz parte e alguns métodos de conveniência. O Código 3.2 mostra parte da implementação da `FeatureTag`. O pseudocódigo do processo de instrumentação é exibido no Algoritmo 3.1. No contexto do CFG da Figura 3.5, cada um dos nós do CFG é associado a uma `FeatureTag`. Uma instância de uma `FeatureTag` pode conter um número variável de objetos do tipo `E`. Neste caso, optou-se por representar as *features* como `String`s. O primeiro nó, `int x = 0;`, recebe uma instância de uma `FeatureTag` vazia, já que este não está associado a nenhuma *feature*, o segundo nó, `x++;` recebe uma `FeatureTag` contendo a `String` "*A*" e o último nó, `x--;`, uma `FeatureTag` com a `String` "*B*".

Algoritmo 3.1 Algoritmo em pseudocódigo do processo de instrumentação.

```
for all Unit u in DirectedGraph do
    if u contains any COLORS then
        Add new FeatureTag(COLORS) to u
    end if
end for
```

```
1 public class FeatureTag<E> extends AbstractSet<E> implements
     Tag {
2   ...
3   private Set<E> features = new HashSet<E>();
4   ...
5 }
```

Código 3.2: Parte do código da classe `FeatureTag`.

É importante ainda notar que o CIDE, de onde as informações das cores são extraídas, tem a sua própria AST, assim como o SOOT, mas ambas são muito diferentes, haja vista que a AST do CIDE é de, neste caso, linguagem Java, enquanto que a do SOOT é da representação intermediária Jimple. Foi preciso, portanto, elaborar um mecanismo que permita mapear entre os nós da AST do CIDE e as `Units` do SOOT. Afortunadamente, o SOOT armazena como `Tags` a linha e coluna do código fonte de onde a `Unit` se originou do código fonte e, desta maneira, é possível utilizar-se desta informação para comparar com os nós da AST do CIDE.

O próximo passo consiste em estender a estrutura de dados que o *framework* usa para os *lattices*. Para o SOOT, *lattices* são objetos que implementam a interface `FlowSet`. É a responsabilidade de um `FlowSet` armazenar o conteúdo das análises (geralmente em uma `Collection` membro), bem como encapsular operações entre *lattices*, como união, diferença, intersecção etc. Há uma implementação padrão para estes métodos na classe abstrata `AbstractFlowSet`, ficando a cargo do desenvolvedor a implementação do armazenamento de informações.

Como o objetivo é que o *lifted lattice* seja capaz de conter vários *lattices* dentro de si associados às configurações, faz sentido que a estrutura de dados que os armazene seja um `Map` com configurações como chaves e *lattices* como seus valores, como pode ser visto na linha 3 do Código 3.3. Ele deve representar uma estrutura do tipo:

$$(\{A\} \mapsto +, \{B\} \mapsto -, \{A,B\} \mapsto 0/+)$$

como já mostrado na Seção 3.2.1. Neste caso, a configuração $\{A,B\}$, por exeplo, é um `Set<String>` contendo as `Strings` "*A*" e "*B*", e `0/+` é um `FlowSet` que representa a informação de *zero ou positivo*, onde o primeiro seria uma chave para o `Map` e o segundo seu valor.

A implementação das operações entre `FlowSets` para o *lifted lattice* resume-se a delegá-las para os respectivos *lattices* internos ao *lifted lattice*, como mostrado na linha 25 do Código 3.3, e portanto não é necessário entrar em detalhes sobre elas. A esta implementação de `FlowSet` deu-se o nome de `LiftedFlowSet`, e parte do seu código pode ser visto no Código 3.3. Faça-se notar que esta independe da implementação dos `FlowSets` que a compõe, um benefício adquirido ao implementar o `LiftedFlowSet` com agregação em vez de unicamente com herança.

```
1  public class LiftedFlowSet extends AbstractFlowSet {
2
3    private Map<Set<String>, FlowSet> map = new HashMap<Set<
       String>,FlowSet>();
4
5    public LiftedFlowSet(Collection<Set<String>> configurations
       , FlowSet clonee) {
6      for (Set<String>configuration : configurations) {
7        map.put(configuration, clonee.clone());
8      }
9    }
10
11   @Override
12   public void union(FlowSet other, FlowSet dest) {
13     LiftedFlowSet otherLifted = (LiftedFlowSet) other;
14     LiftedFlowSet destLifted = (LiftedFlowSet) dest;
15
16     Iterator<Entry<Set<String>, FlowSet>> iterator = this.map
         .entrySet().iterator();
17     while (iterator.hasNext()) {
18       Entry<Set<String>, FlowSet> entry = (Entry<Set<String>,
           FlowSet>) iterator.next();
19       Set<String> configuration = entry.getKey();
20
21       FlowSet thisFlowSet = entry.getValue();
22       FlowSet destFlowSet = destLifted.map.get(configuration)
           ;
23       FlowSet otherFlowSet = otherLifted.map.get(
           configuration);
24
25       thisFlowSet.union(otherFlowSet,destFlowSet);
26     }
27   }
28   ...
29 }
```

Código 3.3: Implementação de um *lifted lattice*.

O idioma visível nas linhas 13–26 mostra como as operações em questão são delegadas a medida que itera-se sobre os `Map`s dos operandos `LiftedFlowSet`s. A semântica desta operação

é que `this` é o primeiro operando (de onde o método é chamado), `FlowSet other` é o segundo operando e o resultado deve ser depositado em `FlowSet dest`.

Por último, é preciso certificar-se de que as funções de transferências utilizam o *lifted lattice*. Como já dito na Seção 2.5.1, as funções de transferências são implementadas ao criar uma análise estendendo a classe `AbstractFlowAnalysis`. Além dos métodos abstratos desta classe, deve-se ainda implementar um algoritmo de *fixed point*. O *framework*, entretanto, já provê uma implementação destes métodos, chamada `ForwardFlowAnalysis`, bem como o algoritmo, restando apenas implementar a lógica específica de cada análise através do contrato também já descrito na Seção 2.5.1, como um padrão de projeto *template method* [7].

A Figura 3.6 mostra cada um dos métodos do contrato necessários para implementar uma análise. Note-se ainda que para o *framework*, há 2 (dois) `FlowSets` por nó no CFG: um antes e outro logo depois dele, representados como círculos preenchindos na Figura. O círculo imediatamente anterior ao nó é chamado de *before flow*, e o logo abaixo dele de *after flow*. Eles representam, respectivamente, o *lattice* antes e depois do nó. Abaixo está um sumário destes métodos:

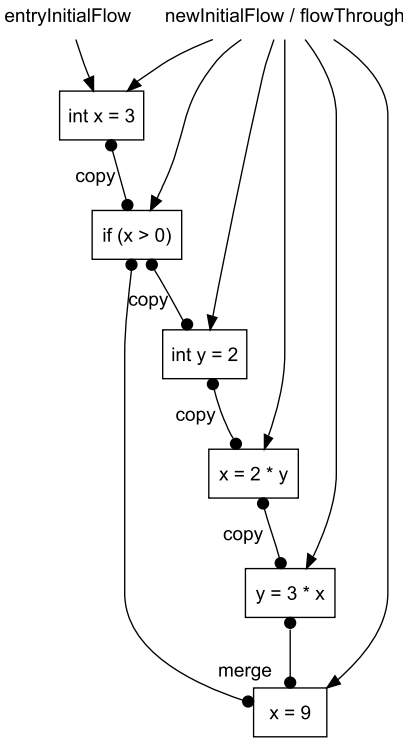


Figura 3.6: Como os métodos do contrato de análise do SOOT afetam os nós de um CFG.

- `newInitialFlow`: utilizado para associar um `FlowSet` inicial a cada um dos nós do CFG *antes* de simular a execução.
- `entryInitialFlow`: idem, mas apenas para o nó inicial (*head*) do CFG;
- `flowThrough`: função de transferência; é chamada para todas as `Unit` do CFG, uma de cada vez, e determina como os `FlowSets` são afetados pela simulação;

- `copy`: permite ao desenvolvedor tratar como um *after flow* é "copiado" para o *before flow*;
- `merge`: define como tratar quando há mais de um fluxo "entrando" em um nó do CFG. Também chamado de confluência.

Para estar em conformidade com a *lifted transfer function*, o método `flowThrough` deve afetar apenas os *lattices* das configurações às quais esta instrução pertence. Um idioma minimalista para se obter este comportamento é mostrado no Código 3.4. Novamente no contexto do CFG da Figura 3.5, se `x++;` é uma `Unit` com uma `FeatureTag<String>` que contém {"*A*"}, então a *lifted transfer function* deve somente simular a execução desta `Unit` paras as configurações que contenham todos os elementos da `FeatureTag`, isto é $\{\{A\},\{A,B\}\}$.

```
1  protected void flowThrough(LiftedFlowSet source, Unit unit,
       LiftedFlowSet dest) {
2    FeatureTag<String> tag = (FeatureTag<String>) unit.getTag("
       FeatureTag");
3    Collection<String> features = tag.getFeatures();
4
5    Collection<Set<String>> configurations = source.
       getConfigurations();
6
7    for(Set<String> configuration : configurations) {
8      FlowSet sourceFlowSet = source.getFlowSet(configuration);
9      FlowSet destFlowSet = dest.getFlowSet(configuration);
10
11     if (configuration.containsAll(features)) {
12       ...
13     } else {
14       sourceFlowSet.copy(destFlowSet);
15     }
16   }
17 }
```

Código 3.4: Uma implementação para a *lifted transfer function*.

A variável `Collection<String> features` na linha 3 contém a lista de *features* as quais a `Unit unit` está associada, de acordo com a instrumentação. Na linha 7 está o *foreach* que irá iterar sobre todas as configurações que o `LiftedFlowSet source` possui. Já na linha 11 está o mais importante detalhe deste método: dentro deste `if` só devem ser alterados os *lattices* cujas configurações contém todas as *features* que foram associadas a `Unit`. Caso contrário (linha 14), o `FlowSet` é copiado para o *after flow*, e portanto nada muda no *lattice*.

Com essas modificações e extensões, é possível finalmente se obter análise de fluxo de dados sensíveis a *features* com o *framework* SOOT. Estes 3 elementos (CFG instrumentado, *lifted lattice* e *lifted transfer function*) formam o núcleo do componente *Feature Sensitive*.

Na seção seguinte, será explicado como as informações coletadas aqui são utilizadas para detectar dependências entre *features* e exibí-las para o usuário.

3.3 Interfaces Emergentes: Implementação

Para alcançar os objetivos propostos pelos autores do conceito de Interfaces Emergentes [20] é preciso interpretar as informações que análises sensíveis a *features* agregam. Neste trabalho, focou-se na análise *reaching definitions* como fonte das informações utilizadas para gerar interfaces para uma LPS.

LPS são instrínsicamente combinatoriais: o código fonte representa uma família de produtos. Embora a ideia e implementação descritas na seção anterior permita executar análises de fluxo de dados para todos os produtos de uma LPS simultaneamente, a quantidade de informação coletada é também combinatorial. Assim sendo, faz parte do desafio de implementar uma ferramenta para interfaces emergentes exibir as informações de forma concisa e amigável para o usuário.

No contexto do SOOT, as informações relativas a uma análise ficam guardadas na própria classe que estende a `FlowAnalysis`, e estão disponíveis através dos métodos `getFlowAfter(Unit u)` da `FlowAnalysis` e `getFlowBefore(Unit u)` da classe pai desta última, `AbstractFlowAnalysis`, que representam o *flow after* e o *flow before*, respectivamente.

A Figura 3.7 mostra os `LiftedFlowSet`s da análise *reaching definitions* para cada instrução de um simples programa. Para obter o `LiftedFlowSet` da `Unit u` da instrução `x = 9`, por exemplo, basta chamar a função `#getFlowAfter(u)` ou `#getFlowBefore(u)` da classe da análise. Este `LiftedFlowSet` representará a seguinte informação:

$$(\{A\} \mapsto \{\texttt{x = 2 * y}\}, \{B\} \mapsto \{\texttt{x = 3}\}, \{A,B\} \mapsto \{\texttt{x = 2 * y}\})$$

que equivale aos *lattices* presentas na linha da instrução `x = 9` da Figura 3.7. Estes `LiftedFlowSet`s são usados como fonte de informação para identificar dependências entre *features* e em seguida gerar uma interface informando-as ao desenvolvedor. No contexto da Figura 3.7, por exemplo, se o desenvolvedor está interessado na instrução `int x = 3`, então deve-se procurar *usos*[1] dessa definição. Neste caso, a atribuição `int x = 3` é usada apenas na instrução `y = 3 * x`, que por sua vez só está presente para a configuração $\{B\}$. Em outras palavras, `int x = 3` alcança um uso somente para a configuração $\{B\}$. Isto indica que uma **dependência** foi encontrada entre a atribuição de interesse do desenvolvedor e a configuração

[1] Caracteriza-se uso, neste contexto, a presença da variável em uma instrução qualquer, desde que a instrução não seja uma atribuição a esta mesma variável.

$\{A\}$	$\{B\}$	$\{A,B\}$	
{}	{}	{}	`int x = 3;`
{x = 3}	{x = 3}	{x = 3}	`if (x > 0){`
{x = 3}	{x = 3}	{x = 3}	`  int y = 2;`
{x = 3, y = 2}	{x = 3, y = 2}	{x = 3, y = 2}	$[A]$: `x = 2 * y;`
{x = 2 * y, y = 2}	{x = 3, y = 2;}	{x = 2 * y, y = 2}	$[B]$: `y = 3 * x;`
			`}`
{x = 2 * y}	{x = 3}	{x = 2 * y}	$[A,B]$: `x = 9;`

Figura 3.7: Um pequeno programa e o conteúdo dos `LiftedFlowSet` p/ a análise *reaching definitions*.

$\{B\}$, e portanto pode trata-se de informação relevante ao desenvolvedor.

Contudo, o conteúdo da mensagem a ser exibida na interface depende diretamente da seleção do usuário. É essencial, portanto, que ao iterar sobre sobre as informações calculadas tenha-se sempre isto como um filtro e ponto de partida.

Agora de posse das informações coletadas pelas análises é preciso compor a interface. Dividiu-se a construção da interface em três subfases distintas para facilitar a compreensão: (i) construir uma estrutura intermediária que organiza as informações já filtradas, (ii) iterar sobre esta estrutura interpretando os resultados e produzindo o conteúdo da interface e (iii) exibir a interface. O Algoritmo 3.2 mostra, para análise *reaching definitions*, a construção da dita estrutura intermediária, aqui chamada de `REACHES DATA`. Seu principal objetivo é estruturar como os diferentes elementos de código e configuração se inter-relacionam. Na fase (ii) itera-se sobre as informações, filtradas em (i), produzindo assim o conteúdo da interface, neste caso mensagens de texto dizendo sob qual configuração uma determinada definição é utilizada em uma *feature* diferente daquela que contém a definição (por exemplo "[B] Definição `x = 3` alcança a linha 56 (*feature B*)."). Por último, em (iii), exibir as mensagens de textos construídas em (ii).

Embora [19] dê como exemplos interfaces emergentes compostas apenas por mensagens de texto, em teoria ela pode ser composta por vários tipos de informação: textual (*e.g.* "Definição `x = 3` alcança a *feature B* na linha 42."), gráfica (*e.g.* um grafo ou figura) ou ainda por um conteúdo interativo.

Com a exibição de mensagens, o desenvolvedor poderá manter a abstração do código possivelmente escondido e saberá com mais precisão como mudanças podem afetar diferentes pontos do código e consequentemente diferentes produtos. Isto porque as dependências entre as *features* podem ser calculadas computacionalmente, e não mais mentalmente. Terá também um melhor conhecimento de quais produtos precisarão ser testados depois da manutenção.

Algoritmo 3.2 Algoritmo em pseudocódigo da criação da interface para a *reaching definitions*.

```
USER SELECTION ← set of all instructions within the user selection
ASSIGNMENTS ← set of assignments in USER SELECTION
STATEMENTS ← set of all instructions within a method
CONFIGURATIONS ← set of all possible configurations
METHOD ← method where the user selection occurred
REACHES DATA ← set of entries in the form of (CONFIGURATION, ASSIGNMENT, STATEMENT,
DIFFERENCE)

for all ASSIGNMENTS asgn in USER SELECTION do
  for all CONFIGURATIONS conf do
    for all STATEMENTS stmt in METHOD do
      if DIFFERENCE ← (FEATURES(asgn) ∩ FEATURES(stmt)) ≠ {∅} then
        if stmt uses asgn then
          Store (conf,asgn,stmt,DIFFERENCE) in REACHES DATA
        end if
      end if
    end for
  end for
end for
```

Capítulo 4

Avaliação

Este capítulo avalia a ferramenta descrita neste trabalho sob dois aspectos: (i) como ela pode ajudar na detecção de dependências entre *features* diferentes e exibir informações para o usuário; e (ii) quão rápido ela é capaz de computá-la. Esta avaliação será feita em cima de 2 cenários de uso derivados de situações reais de manutenção de código.

4.1 Cenários de Uso

Neste seção serão mostrados alguns cenários onde a ferramenta é utilizada. Discute-se ainda o conteúdo da interface gerada como fonte de informação para a investigação do desenvolvedor. As Figuras 4.1 e 4.2 que mostram o código e a interface para o cenário 1 e 2 respectivamente não escondem o código fonte das *features* em questão para que se compreenda melhor o cenário e como as *features* estão ligadas por elementos em comum.

4.1.1 Cenário 1

Este cenário é baseado no jogo *Best Lap* [1] implementado em J2ME, e é distribuído para 65 dispositivos diferentes pela companhia *Meantime Mobile Creations*[1]. O código fonte para este cenário, colorido com o CIDE, pode ser visto na Figura 4.1. No trecho de código, a variável `totalScore`, definida na linha 30, é utilizada na linha 37 colorida com a *feature* `Arena`. O desenvolvedor deve tomar cuidado ao alterar a atribuição desta variável, pois ela é utilizada em uma *feature* diferente daquela que é o foco do desenvolvedor no momento. Neste cenário a ferramenta pode ser útil pois ela pode identificar automaticamente a dependência existente, sem que o desenvolvedor precise perder tempo investigando os usos da variável em diversas outras *features*.

O desenvolvedor poderá selecionar o código fonte no qual tem interesse em investigar e invocar o comando que irá calcular a interface emergente. A interface exibida para seleção

[1] `http://www.meantime.com.br/`

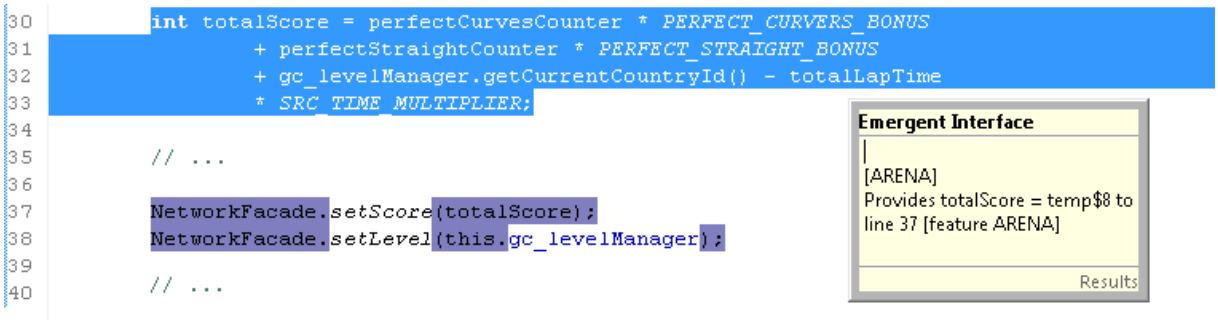


Figura 4.1: Código fonte e interface para o cenário 1 de manutenção.

"`totalScore = ...`" neste cenário pode ser vista também na Figura 4.1. A ferramenta identifica, através da interpretação da análise *reaching definitions* sensível a *features*, que a atribuição desta variável alcança a *feature* `Arena` na linha 37, e compõe a interface com tais informações como pode ser visto no *pop-up* da Figura 4.1. O desenvolvedor terá então consciência de que a linha de código que ele deseja manter pode afetar uma determinada configuração, isso de maneira automática, o que pode implicar aumento de produtividade.

Como uma limitação deste trabalho, há, no entanto, um elemento que o desenvolvedor deve ficar atento, pois seu significado pode não ser imediato: o termo `$temp8`. A representação Jimple quebra expressões complexas em simples, o que envolve criar variáveis temporárias com esta mostrada. Desta forma, o termo `$temp8` representa uma das partes da quebra da expressão a qual a variável `totalScore` é atribuida. Embora torne a expressão diferente da original, o entendimento da interface pode ser completo desde que o desenvolvedor compreenda o significado destas variáveis temporárias.

4.1.2 Cenário 2

Neste cenário a ferramenta é aplicada a um trecho de código da biblioteca *GLib*[2]. A biblioteca é originalmente implementada em C com o uso de pré-processadores. Para este cenário, o código foi portado para a linguagem Java. Ademais, substitui-se as diretivas de pré-processador por *features* com o CIDE do trecho de código em questão, como pode ser visto na Figura 4.2. Neste cenário, há 3 *features* opcionais que fazem uso da variável `status`. Sem suporte ferramental, se o desenvolvedor deseja realizar alguma mudança que envolva esta variável, então pode ser necessário investigar $2^3 = 8$ produtos.

O resultado da interface para a seleção `GFileAttributeStatus status = null;` pode ser visto também na Figura 4.2. Embora a figura não seja grande o suficiente para contê-la, a interface mostra que a variável afeta na realidade apenas 6 produtos, informação que o desenvolvedor deveria, caso não tenha nenhuma ferramenta que o auxilie neste processo, ter que descobrir por si só. Note-se ainda que a quantidade de informação compilada é bem maior

[2] http://www.gtk.org/

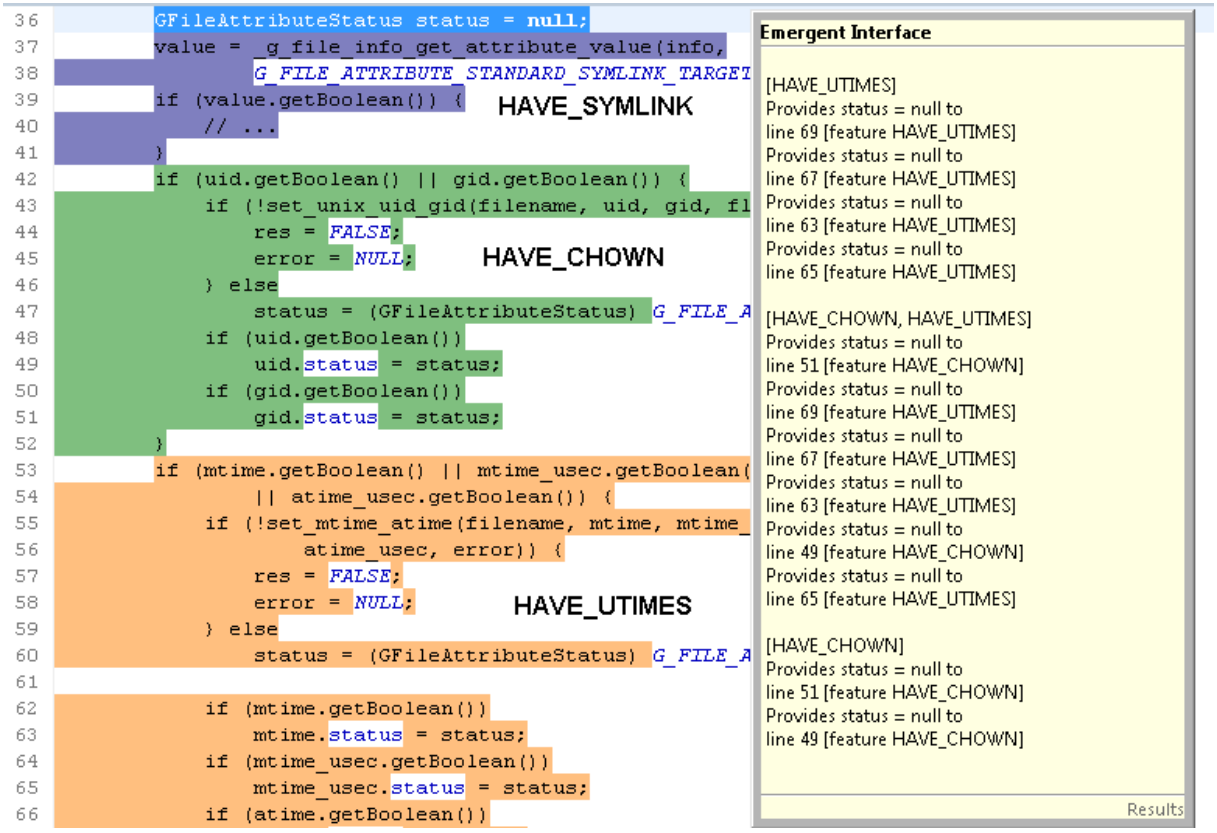


Figura 4.2: Código fonte e interface para o cenário 2 de manutenção.

que a do cenário anterior, mesmo tendo apenas 2 *features* a mais, o que é, novamente, uma amostra de que as LPS são instrísecamente combinatoriais, e por isso mesmo as informações extraídas delas também o são.

Para casos onde a quantidade total de configurações possíveis é maior, utilizar mensagens de texto como a mostrada neste cenário pode se tornar inviável. Estudar como dispor uma maior quantidade de informação em outros formatos ou mais refinadas faz parte do conjunto de trabalhos futuros.

4.2 Desempenho

Para medir o tempo necessário para computar a análise, intepretar os resultados e exibir a interface, invocou-se a construção da interface emergente pela interface 10 vezes para uma mesma seleção nos dois cenários, e então calculou-se a média. O objetivo de sucessivas medições consiste em remover eventuais ruídos na análise, ou seja, números discrepantes.

O computador onde executou-se este experimento possui as seguintes características:

- *Processador:* Intel®Core™2 Duo P8600 2.4GHz;
- *Memória:* 3GB;

- *Sistema operacional:* Windows Vista™ 32bits
- *Eclipse:* Galileo 3.5.2 SR2
- *JVM:* Java HotSpot™ Client VM (build 14.0-b16, mixed mode)

O tempo necessário para realizar estas tarefas cada uma das vezes para o cenário 1 pode ser visto na Tabela 4.1. Embora a primeira execução seja um pouco mais lenta, as subsequentes são rápidas, especialmente devido as otimizações da JVM. A média é de pouco mais de meio segundo, um tempo razoavelmente baixo.

Execução 1	2	3	4	5	6	7	8	9	10	Média
2073	289	439	372	269	445	512	351	545	422	570,7ms

Tabela 4.1: Tempo em milisegundos necessário para calcular a interface emergente para o cenário 1.

Executou-se a mesma análise também 10 vezes para o cenário 2, e o resultado pode ser visto na Tabela 4.2. Novamente o tempo para a primeira execução é bem mais alto que os subsequentes. A média para este cenário foi de aproximadamente 0,6 segundos, um número também razoavelmente baixo.

Execução 1	2	3	4	5	6	7	8	9	10	Média
2082	475	339	310	407	473	441	397	487	584	599,5ms

Tabela 4.2: Tempo em milisegundos para calcular a interface emergente para o cenário 2.

Capítulo 5

Trabalhos Relacionados

A abordagem *Conceptual Module* [3] permite aos desenvolvedores definir módulos conceituais, conjunto de linhas de código fonte tratadas como unidades de lógica, e realizar consultas sobre como as unidades lógicas interagem com o restante do código fonte ou ainda com outros módulos conceituais. Esta ferramenta também usa análise de fluxo de dados e de controle para computar a interação entre os módulos. Como benefícios, os autores afirmam que o fardo de sumarizar e correlacionar as unidades lógicas com o código fonte e entre outras unidades é movido para a ferramenta, aumentando assim a produtividade do desenvolvedor. O CIDE EI é conceitualmente bastante similar a abordagem do *Conceptual Module*, uma vez que este também utiliza análises de fluxo para computar a interações entre trechos de código. Contudo, o CIDE EI vai mais além, pois é sensível a *features*, e mais específico, já que é voltado para LPS.

Em um outro trabalho relacionado, pesquisadores desenvolveram uma ferramenta chamada *RTalk* [27] que permite gerar representações intermediárias de um programa de modo a facilitar a execução de análises. Adicionalmente, a ferramenta é capaz de mapear discretamente entre os elementos da representação e os do código fonte, permitindo que o vocabulário das análises sejam "traduzidos" para um outro mais adequado para o desenvolvedor que desconhece em profundidade a análise executada. A ferramenta CIDE EI também utiliza representações intermediárias sobre as quais as análises são implementadas. A principal representação utilizada é o Jimple, presente no *framework* SOOT. O CIDE EI também tenta mapear entre os elementos do Jimple encontrado nas análises e os elementos do código fonte, utilizando as linhas de código como parâmetro. Entretanto, este mapeando não é tão preciso quanto o do *RTalk*, devido a restrições da representação do Jimple, como pode ser visto na Seção 4.1.2. Tal melhoria também faz parte dos trabalhos futuros. Além de capturar dependências, o CIDE EI também é capaz de capturar interação entre *features*, e ainda outros tipos de interações, como instruções de mudança de fluxo, como *break* e *continue*.

O *Senseo* [21] é também um *plug-in* para o IDE Eclipse que é capaz de computar informações de tempo de execução, como número de chamadas feitas de método, tipos de objetos, número de instruções de *bytecode* executadas etc. Estas informações podem ser utilizadas pelo

desenvolvedor para identificar pontos de interesse no código fonte. O *plug-in* exibe essas informações de como grafos, *pop-ups*, *tooltips* etc. Os autores realizaram um experimento controlado, no qual detectaram que desenvolvedores tiveram 33,5% mais respostas corretas e uma redução de até 17,5% no tempo necessário para realizar tarefas de manutenção. Em comparação, o CIDE EI também é capaz de computar informações simulando a execução do código através de análises de fluxo de dados sensíveis a *features*, mas seu foco é nas dependências entre entre *features*. O *Senseo* é de fato uma ferramenta de propósito geral, enquanto o CIDE EI é uma ferramenta de para um nincho específico. Desta forma, o CIDE EI é capaz de captar dependências entre *features* em uma LPS, ao contrário do *Senseo*. Embora não tenha sido feito nenhum experimento ciêntífico para comprovar, espera-se que o CIDE EI seja capaz de aumentar a produtividade do desenvolvedor. Realizar tal experimento faz parte dos trabalhos futuros.

Testar LPS pode ser uma tarefa árdua, pois pode requerer que muitos produtos sejam examinados. Há, entretanto, *features* cuja presença não afeta o resultado de alguns dos testes. Faz sentido, portanto, que exista uma espécie de filtro para que certas combinações não sejam testadas desnecessariamente, reduzindo assim o esforço necessário para analisar a LPS. Esta ideia foi apresentada em um trabalho recente [8] que mostra como análises de fluxo de dados podem ser usadas para descobrir quais *features* um conjunto de casos de teste alcançam. Depois as *features* alcançadas são combinadas para formar as configurações que precisam ser testadas, diminuindo assim o conjunto total de testes que precisam ser executados. Entretanto, as análises executadas neste trabalho não são sensíveis a *features*, no sentido de que não levam em conta as informações do *feature model*.

Há situações, como manutenção em código, onde apenas um trecho de código é o ponto de interesse de um desenvolvedor. O primeiro passo envolvido na tarefa de manutenção neste caso é compreender o comportamento do código em questão. Isto pode forçar o desenvolvedor a investigar uma grande parte do sistema afim de compreender o comportamento como um todo, e só então partir para o trecho onde a manutenção deve ocorrer. *Automatic Program Slicing* [28] pode ajudar a automatizar o processo de identificação das fatias (daí o termo *slicing*, ou fatiamento em inglês) de código que envolvem um determinado comportamento, desde que seja dado como entrada de alguma forma de identificá-lo, como um conjunto de instruções. Posto isso, análises de fluxo de dados podem achar todas as fatias de código que podem ser influenciadas pelo comportamento especificado. Pode-se fazer uma analogia entre o CIDE EI e *program slicing* ao considerar que a seleção do usuário é o comportamento que se deseja fatiar, e a fatia encontrada pelo CIDE EI é na realidade um comportamento identificado de forma sensível a *features*, mais especificamente uma comportamento de dependência e interação entre as *features*. Note-se ainda que o *Conceptual Module* é também uma espécie de *program slicing*.

Capítulo 6

Conclusão

Este trabalho apresentou o CIDE EI, um *plug-in* para o IDE Eclipse capaz de computar interfaces emergentes através de análises de fluxo de dados sensíveis a *features* para LPS implementadas com o CIDE.

Para realizar as análises, a ferramenta conta com o SOOT, um *framework* para análise e otimização de *bytecode* Java. Este framework possui uma infra-estrutura que permite que análises de fluxo de dados sejam executadas sobre a sua representação intermediária do *bytecode*, chamada *Jimple*.

As análises sensíveis a *features* elevam as análises de fluxo de dados comuns de modo que estas se tornam capazes de computar informação para uma família de produtos, e não mais para um único programa. A implementanção dos elementos que compõem a análise de fluxo dados (CFG, *lattices* e funções de tranferência) juntamente com a análise de *reaching definitions* foram implementadas estentendo a estrutura de classes de *framework* SOOT. Este conjunto de classes forma um componente da arquitetura da ferramenta, que por sua vez é utilizado por outro componente que é capaz de interpretar os resultados das análises sensíveis a *features* e identifica dependências entre as *features* que oriundas da seleção do usuário. Depois de computar a análise e interpretar os resultados, as dependências identificadas em diferentes configurações são exibidas para o usuário como uma interface emergente.

Para avaliar a ferramenta, utilizou-se dois diferentes cenários onde acontecem dependências entre as *features* através de elementos compartilhados. Além de aplicar a ferramenta a estes dois cenários, mediu-se ainda o tempo necessário para calcular e exibir as interfaces com o intuito de avaliar o seu desempenho, haja vista que LPS são intrínsecamente combinatoriais, e portanto a quantidade de informações calculadas pode atingir níveis elevados. A ferramenta mostrou-se importante em ambos os cenários, onde foi capaz de identificar dependências entre diferentes *features*, em um tempo que pode ser considerado hábil para o hardware onde foi executado. Para ambos os casos, a interface foi computada em menos de 0.6s. Portanto, a espera em tal computação é baixa em cenários semelhantes aos utilizados neste trabalho, sendo importante para não afetar a produtividade dos desenvolvedores.

Em alguns casos, a quantidade de informação identificada é simplesmente grande demais para ser exibida para o usuário em elementos simples de interface, como *pop-ups* e *tooltips*. Investigar uma maneira de apresentar estas informações para o usuário sem sobrecarregar a interface faz parte dos trabalhos futuros para esta ferramenta. Como os resultados das análises são calculados e interpretados em Jimple, é possível que a interface contenha elementos pertencentes a esta representação intermediária, e não somente a linguagem Java, o que pode inicialmente diminuir a compreensão do desenvolvedor ao ler a mensagem contida na interface emergente. Isto também é um ponto que precisa ser melhorado para este trabalho. Outro ponto que precisa ser trabalhado é a necessidade de adicionar outros tipos de análises para identificar mais dependências, como cadeias de atribuições. Por último, mas não menos importante, é necessário validar a eficiência e eficácia da ferramenta através de experimentos empíricos em grupos controlados para avaliar quantitativa e qualitativamente os benefícios que argumentou-se que a ferramenta traria.

Bibliografia

[1] Vander Alves. Implementing software product line adoption strategies (phd thesis). Technical report, Recife, PE, BRA, 2007.

[2] Sven Apel and Christian Kästner. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, September 2009. (column).

[3] Elisa L. A. Baniassad and Gail C. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 64–73, Washington, DC, USA, 1998. IEEE Computer Society.

[4] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[5] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, AOSD '02, pages 10–18, New York, NY, USA, 2002. ACM.

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

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] Chang Hwan, Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. 2011.

[9] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. Xvcl: Xml-based variant configuration language. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 810–811, Washington, DC, USA, 2003. IEEE Computer Society.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[11] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, chapter 10, pages 220–242. Springer-Verlag, Berlin/Heidelberg, 1997.

[13] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[14] Charles W. Krueger. Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 282–293, London, UK, 2002. Springer-Verlag.

[15] Charles W. Krueger. New methods in software product line practice. *Commun. ACM*, 49:37–40, December 2006.

[16] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[17] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[18] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[19] Márcio Ribeiro and Paulo Borba. Towards feature modularization. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 225–226, New York, NY, USA, 2010. ACM.

[20] Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent feature modularization. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 11–18, New York, NY, USA, 2010. ACM.

[21] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Exploiting dynamic information in ides improves speed

and correctness of software maintenance tasks. *Transactions on Software Engineering*, 2011. To appear.

[22] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11:215–255, April 2002.

[23] Henry Spencer. ifdef considered harmful, or portability experience with c news. In *In Proc. Summer'92 USENIX Conference*, pages 185–197, 1992.

[24] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (ACM Press)*. Addison-Wesley Professional, December 1997.

[25] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[26] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99. IBM Press, 1999.

[27] Daniel von Dincklage and Amer Diwan. Integrating program analyses with programmer productivity tools. *Software: Practice and Experience*, January 2011.

[28] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.