Predição de Mudanças em Workflows Através do Uso de Dependências Lógicas

Gustavo Ansaldi Oliva, Marco Aurélio Gerosa

Instituto de Matemática e Estatística (IME) – Universidade de São Paulo (USP)
Rua do Matão, 1010 - Cidade Universitária - São Paulo - SP - Brasil - CEP 05508-090

{goliva, gerosa}@ime.usp.br

1. Introdução

Repositórios de larga escala de workflows, que podem compreender milhares de workflows em um cenário do mundo real, são intrinsicamente complexos. Workflows nesses repositórios frequentemente ligam-se uns com os outros, foram uma complexa rede de dependências. Conforme os workflows evoluem, seus números de elementos e de interconexões tendem a aumentar. Ainda, organizações frequentemente apoiam-se em workflows providos *out-of-the-box* por vendors de sistemas de gerenciamento de workflows (*workflow management system*s). Isto significa que modificar ou substituir esses workflows core pode afetar um alto número de outros workflows que dependem deles. Portanto, evoluir repositórios de workflows é uma tarefa desafiadora.

De fato, artefatos de software são produzidos de forma inerentemente incremental através de mudanças contínuas. Isso faz com que mudanças sejam uma parte integral do processo de evolução de software [25]. Desta forma, o processo e metodologia que dão suporte às mudanças são um fator decisivo entre evolução sustentada de alta qualidade e a aposentadoria prematura de um sistema de software [23]. A predição de mudanças é uma das atividades essenciais no suporte à evolução de sistemas.

Dependências lógicas (também conhecidas como dependências de mudança [10], dependências evolucionárias [7] e co-changes [6]) são dependências implícitas que ocorrem entre artefatos de software que evoluem juntos [4, 16]. Essas dependências têm sido utilizadas para uma série de propósitos, incluindo a predição de mudanças em artefatos de software [20, 21, 26, 27, 32, 34, 36, 38]. Neste trabalho, propomos uma abordagem para a predição de mudanças em workflow baseada em dependências lógicas.

Este trabalho é estruturado conforme segue. Na seção 2, apresentamos trabalhos relacionados. Na seção 3, introduzimos nossa abordagem para predição de mudanças em workflows. Na seção 4 apresentamos nossas conclusões e planos de trabalhos futuros.

2. Trabalhos Relacionados

2.1. Dependências Lógicas e Predição de Mudanças

Zimmermann *et al.* [38] desenvolveram uma ferramenta capaz de capturar dependências lógicas a partir do sistema de controle de versão CVS. O objetivo é guiar os programadores acerca de mudanças relacionadas: “programadores que mudaram essas funções também

mudaram...". Como os autores estão lidando um sistema de controle de versão que não tem suporte a commits atômicos, eles realizam três tarefas para agrupar mudanças em *transações*:

- Inferindo transações: os autores utilizam uma clássica janela de tempo deslizante (*sliding time window*). Eles consideram que duas mudanças subsequentes s_i e s_{i+1} feitas pelo mesmo autor e com mesmo comentário são parte de uma mesma transação t se elas estiverem 200 segundos distantes.

- Ignorando merges de repositório: a ferramenta ignora todas as mudanças que afetam mais de 30 entidades.

- Obtendo entidades (de baixo nível): a ferramenta parseia os arquivos mudados, associando entidades sintáticas com faixas de linhas. A Figura 1 ilustra um exemplo.

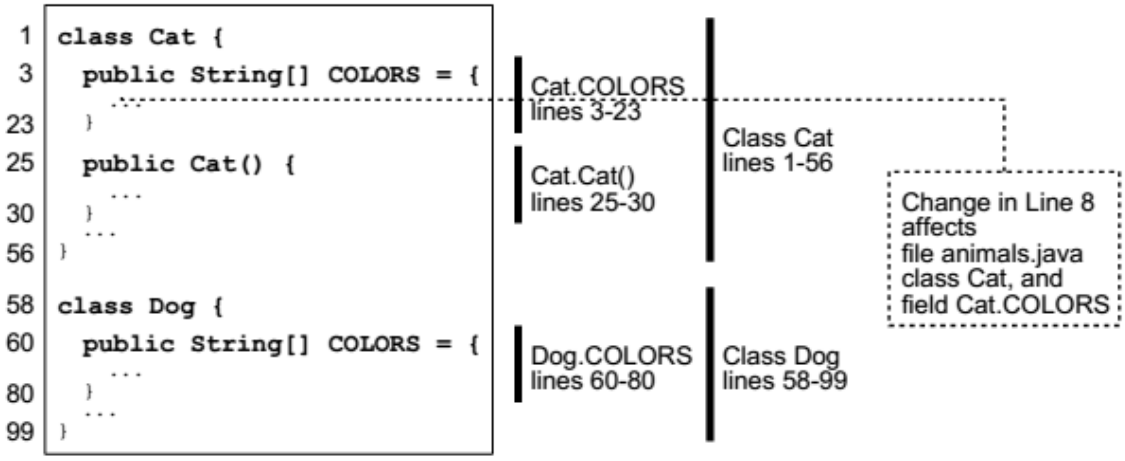

Figura 1. Relacionando mudanças com entidades. Imagem extraída de [38]**.**

De posse das transações, os autores extraem regras de associação (*association rules*), que é a forma como eles formalizam o conceito de dependências lógicas. Uma regra de associação r é um par (x_1, x_2) de dois conjuntos disjuntos x_1 e x_2. Na notação $x_1 \Rightarrow x_2$, x_1 é chamado de antecedente e x_2 é chamado de consequente. Em termos práticos, uma regra de associação $\{x\} \Rightarrow \{y,z\}$ denota que quando o programador muda um artefato x, ele também deve mudar os artefatos y e z. Tais regras não contam uma verdade absoluta, isto é, elas devem ser interpretadas como uma relação probabilística baseada na quantidade de evidência presente nas transações (commits) das quais são derivadas. Essa quantidade de evidência é determinada pelas métricas de suporte e confiança (frequentemente usadas no universo de data mining). A ferramenta desenvolvida pelos autores usa o algoritmo *A Priori* para encontrar as regras de associação. Com as regras em mãos (e os respectivos valores de suporte e confiança calculados), a ferramenta é capaz de (a) sugerir e predizer mudanças futuras, (b) mostrar acoplamento entre itens que não seria detectável por análise estática e (c) prevenir erros oriundos de mudanças incompletas. Um exemplo de uso está ilustrado na Figura 2. Os resultados da avaliação conduzida pelos autores são bastante promissores. Dado um certo arquivo, a ferramenta é capaz de predizer 26% dos arquivos que realmente mudaram na mesma transação. Em 70% de todas transações, as três melhores sugestões da ferramenta contém um local correto (por exemplo, uma função x() na classe A.java).

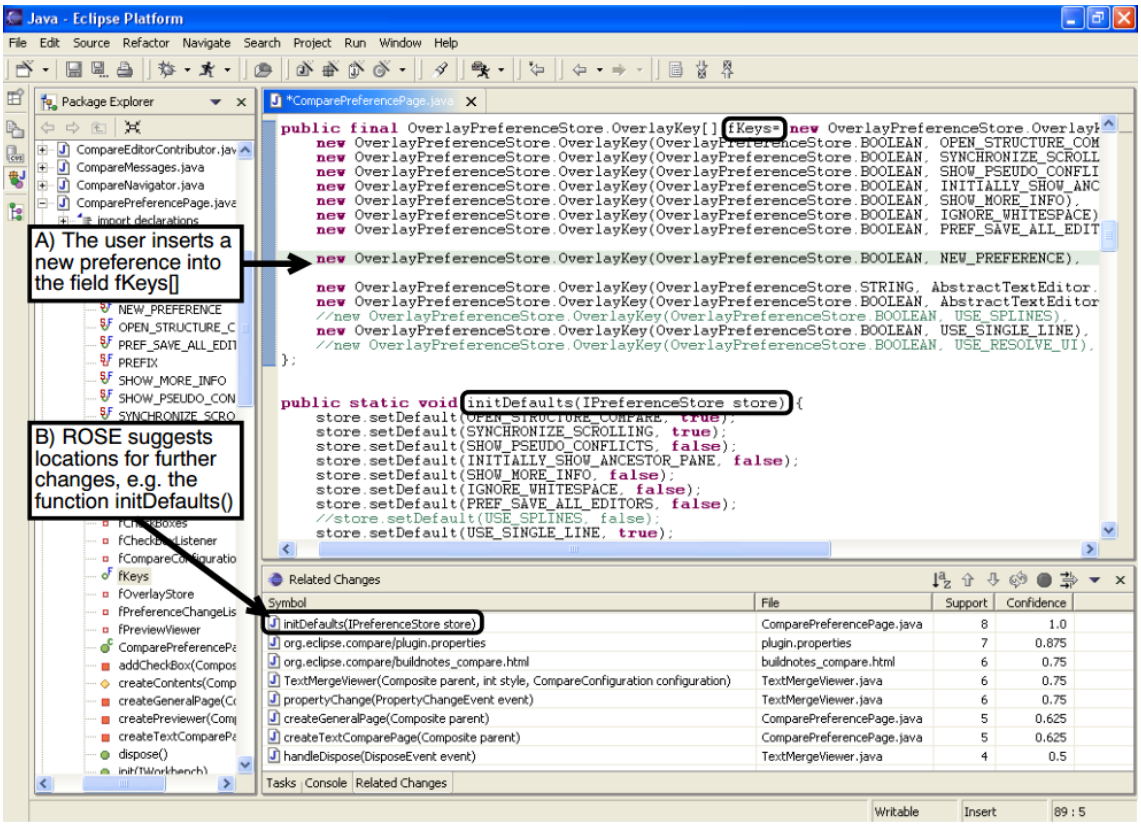

Figura 2. Depois que o programador faz algumas no código (acima), a ferramenta sugere localidades (abaixo) onde, em transações similares do passado, outras mudanças foram realizadas. Imagem extraída de [38]**.**

Wang *et al.* [34] propõem uma abordagem similar à de Zimmermann *et al.* [38] para a detecção de dependências lógicas. A abordagem consiste de três passos básicos (ilustrados na Figura 3):

1) Pré-processamento de dados: Os metadados dos commits são extraídos e inseridos em um banco de dados. Em seguida, assim como no trabalho de Zimmermann *et al.* [38], eles utilizam uma janela de tempo para agrupar transações. Contudo, os critérios são um pouco diferentes: as mudanças devem ser submetidas pelo mesmo autor, o intervalo de tempo entre duas mudanças adjacentes não pode ser maior do que um limiar predefinido e o intervalo de tempo de toda a transação não deve exceder um outro limiar também predefinido. Em seguida, é realizada uma limpeza dos dados: transações que envolvem um número muito alto de arquivos são ignoradas, transações cujas mensagem de log estiverem vazias são ignoradas e arquivos adicionados e em seguida removidos dentro de uma mesma transação de mudança são ignorados.

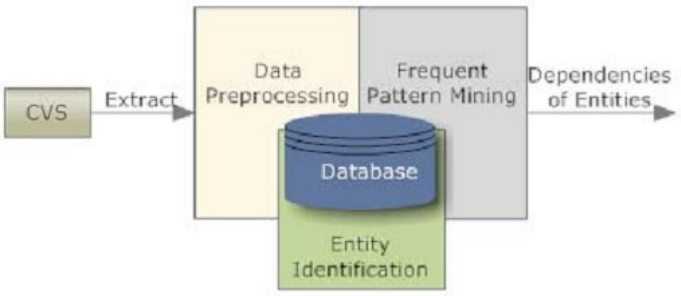

Figura 3. Esquema da Abordagem para Detecção de Dependências Lógicas. Imagem extraída de [34]**.**

2) Extração de entidades (de baixo nível): Um algoritmo fuzzy de matching de entidades é utilizado para precisamente extrair entidades relacionadas ao programa.

3) Por fim, uma versão reduzida do algoritmo *A Priori* é aplicada para encontrar os conjuntos de itens frequentes (*frequent itemsets*) escondidos nas transações de mudanças mapeadas para as entidades de software mudadas. Em seguida, os autores utilizar regras de associação geradas pelos conjuntos de itens frequentes para predizer entidades que mudarão juntas.

Os autores avaliam a abordagem proposta no repositório CVS do projeto Eclipse e comparam os resultados com os obtidos por Zimmermann *et al.* [38] e afirmam obter melhores valores de *precision* e *recall* para diferentes valores mínimos de suporte.

Zhou *et al*. [36] propõem uma abordagem baseada em rede bayesianas para predição de acoplamento lógico. O objetivo é o mesmo daquele dos dois trabalhos anteriores. Mais formalmente, dado um conjunto de entidades $S_e=\{e_1,e_2,...,e_n\}$ que pertencem a um sistema de software e, dado que uma mudança foi aplicada a alguma entidade $e_i \in S_e$, a abordagem dos autores predizem um outro conjunto de entidades $S_c = \{S_c \subset S_e \mid e_i \notin S_c\}$ que também são propensos a mudar. Em vez de usar regras de associação, a abordagem de Zhou *et al*. utiliza redes bayesianas que se baseiam em *features* extraídas dos elementos que mudaram juntos (frequência passada de co-change, idade da mudança, dependência estruturais, etc). Os autores conduzem um estudo de caso em dois sistemas de software livres de médio porte (Azureus e ArgoUML) e demonstram a viabilidade e efetividade da proposta em comparação a trabalhos anteriores. Em particular, os autores mostram valores bastante promissores de *precision* e *recall*.

Malik *et al.* [26] defendem que comentários atualizados são críticos para evolução bem sucedida de sistemas de software. Neste trabalho, os autores investigam quais os fatores que determinam a atualização de comentários em funções (procedimentos, rotinas). Em particular, eles recuperam o histórico de mudanças de código de quatro grandes projetos de código aberto (GCC, FreeBSD, PostgreSQL e GCluster) e, utilizando o algoritmo *Random Forests*, eles investigam a lógica para atualizar comentários segundo três dimensões: (i) características da função modificada, (ii) características da própria mudança e (iii) aspectos de tempo e de *ownership*. A avaliação empírica mostra que os autores conseguem predizer com 80% de acurácia a chance de atualização de um comentário associado a uma função modificada. A análise conduzida mostrou que o número de funções co-mudadas que

dependem da função em questão é um dos fatores mais importantes na determinação das chances de atualização de comentário.

Dois anos após o trabalho anterior, Malik e Shakshuki [27] desenvolvem uma abordagem para predizer mudanças em funções e guiar programadores na tarefa de propagação de mudanças. A abordagem consiste de três fases. Na primeira fase, os dados são extraídos do sistema de controle de versão e então pré-processados para ser tornarem adequados para data mining. Essa primeira fase é bastante similar ao que é feito nos trabalhos de Zimmermann *et al.* [38] e Wang *et al.* [34], portanto omitimos os detalhes. Na segunda fase, três heurísticas são aplicadas nos dados pré-processados para determinar a propagação de mudança de função. Dentre as heurísticas aplicadas, os autores calculam dependências lógicas usando regras de associação para determinar conjuntos de funções que tendem a mudar juntas frequentemente. Na última fase, são recomendadas as entidades relevantes necessitando mudanças baseado na heurística que produz o melhor resultado. Os autores realizam uma avaliação com o projeto PostgreSQL com dados cobrindo um período de 9 anos e descobrem que em 52% dos casos, a heurística baseada em co-mudança de funções foi escolhida como o melhor *predictor* pelo gerenciador de heurísticas, tendo ainda os melhores resultados em *precision*, *recall*, e nas *F-measures* F-1, F-2 e F-0.5.

Em um *short paper*, Kagdi e Maletic [21] defendem a necessidade de se desenvolver um método para predição de mudança que combina os conjuntos de mudanças estimados a partir de análise dependências estruturais (*single version analysis*) e a partir da análise de dependências lógicas (*multiple version analysis*), isto é, mudanças reais encontradas em repositório de código. A hipótese dos autores é que a combinação dessas duas técnicas levaria a um melhor resultado na predição de mudanças. Os autores discutem como isso poderia ser feito e implementado. Os autores também descrevem um arcabouço de avaliação (*evaluation framework*) centrado nos tradicionais conceitos de *precision* e *recall* para ser utilizado como base para pesquisas com esse objetivo.

Kagdi *et al.* [20] apresentam uma abordagem para análise de impacto de mudanças baseada na combinação de duas técnicas: acoplamento conceitual (*conceptual coupling*) e dependências lógicas. Técnicas de recuperação de informação (*information retrieval*) são utilizadas para derivar acoplamentos conceituais a partir de código fonte de uma versão específica (e.g., release) de um sistema. Como de praxe, os autores identificam dependências lógicas minerando os logs do sistema de controle de versão e descobrindo regras de associação. Os autores conduzem um estudo empírico com os dados históricos de quatro projetos de código aberto: Apache httpd, ArgoUML, iBatis e KOffice. Os resultados mostraram que a combinação das duas técnicas fornece melhorias estatisticamente significativas na acurácia quando comparado ao uso de cada uma das técnicas de forma independente. A saber, os autores obtiveram melhorias no recall de até 20% sobre a técnica de acoplamento conceitual no projeto KOffice e de até 45% sobre a técnica de dependências lógicas no projeto iBatis.

Pirklbauer *et al.* [32] propõem um processo e ferramenta para gerência de dependências, dando foco para o aspecto de análise de impacto de mudanças. Um destaque da ferramenta proposta consiste no agrupamento de dependências estruturais e lógicas em uma única base de dados, que depois serve de entrada para mecanismos de visualizações desenvolvidos pelos autores (Figura 4).

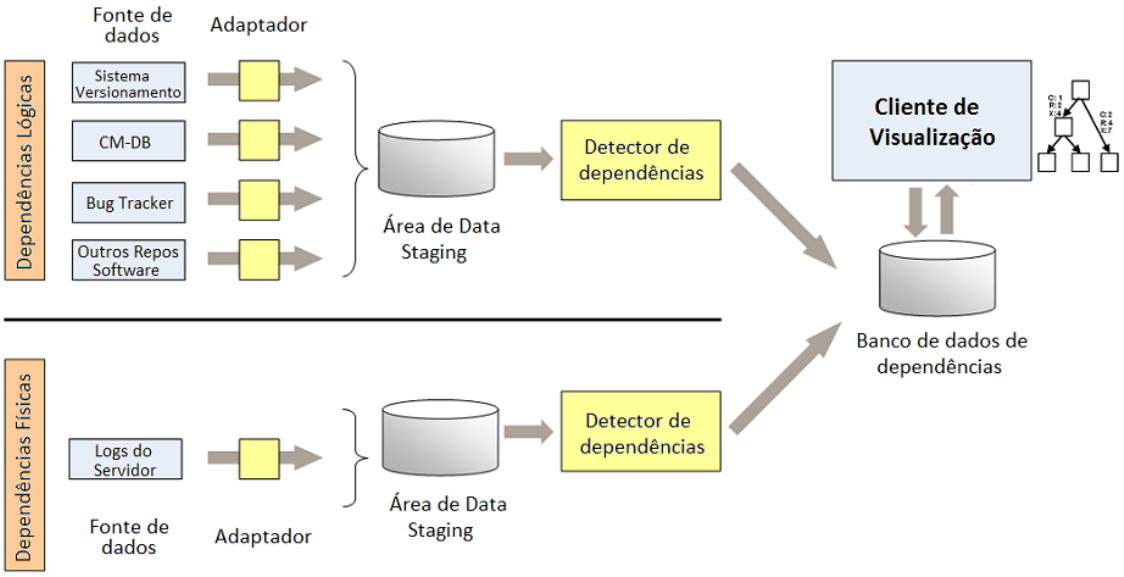

Figura 4. Arcabouço para gerência de dependências proposto por Pirklbauer *et al.* Imagem extraída e traduzida de [32].

2.2. Sobre o Fenômeno de Dependências Lógicas

Investigação da relação entre dependências estruturais e lógicas. Gall *et al.* [16] usam informações do histórico de lançamentos (*release history*) de um sistema de telecomunicações para descobrir dependências lógicas e padrões de mudança entre módulos e subsistemas. A técnica é chamada de *Change Sequence Analysis* (CSA). Nesse caso, os autores afirmam que "acoplamento lógico se refere a um comportamento de mudança idêntico observado em dois elementos durante a evolução do software". Mais precisamente, dada a estrutura do sistema de telecomunicações (Figure 5), são listadas as sequências de mudança (*change sequences*) para todos os programas. Cada sequência corresponde aos lançamentos (*releases*) nos quais um programa foi modificado. Em seguida, as diferentes subsequências são comparadas contra todas as sequências. Se duas ou mais sequências contém a mesma subsequência, então os autores consideram que uma dependência lógica é formada entre essas sequências e entre os programas e subsistemas associados.

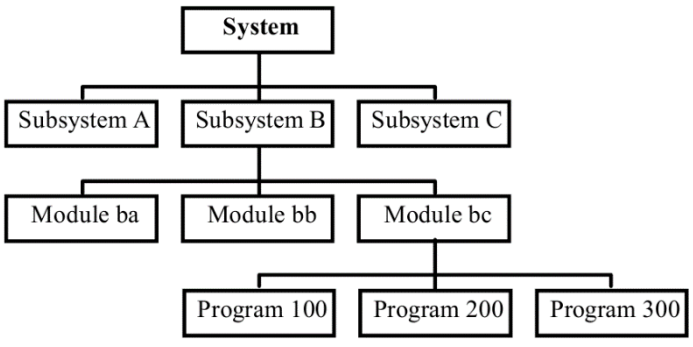

Figure 5. Estrutura lógica do sistema de telecomunicações. Imagem extraída de [16].

Gall *et al.* [17], neste novo estudo, propõem a técnica de *Relation Analysis* (RA) para identificação de dependências lógicas. A técnica investiga classes que mudaram juntas com

frequência. Mais precisamente, os autores mineram os logs do CVS e verificam classes que mudaram juntas e pelo mesmo autor. As datas (timestamps) são comparadas por igualdade, mas uma janela de tempo fixa de quatro minutos é considerada, já que commits grandes demoram algum tempo para serem finalizados. Os autores afirmam ainda que o número de mudanças conjuntas define a "força" da conexão lógica. A técnica de RA complementa a CSA, no sentido que o autor do commit é levado em consideração e são investigados elementos de mais alta granularidade (classes). CSA e RA, juntamente com *Quantitative Analysis* (QA), formam a base da metodologia QCR, que foi proposta por Krajewski e que tem por objetivo analisar a evolução de software por meio de dependências lógicas entre módulos do sistema.

Zimmermann *et al.* [37] usam a mesma ideia de Gall, porém considerando uma janela fixa de tempo de 3 minutos. Além disso, os autores introduzem a ideia de suporte (*support*) e confiança (*confidence*) como filtros para dependências lógicas. Ainda, o trabalho foi o primeiro a considerar acoplamento lógico em baixo grau de granularidade, relacionando as mudanças a funções, métodos e atributos. Por exemplo, os autores encontraram um acoplamento com suporte alto entre um struct definido no arquivo `AppData.h` e o campo `ddd_resources[ ]` definido no arquivo `resources.c`. Por fim, os autores apresentam a ferramenta eRose, que implementa essa técnica de identificação de dependências lógicas e ainda provê recursos de visualização. Anos mais tarde, os autores refinam essa ferramenta em um outro trabalho [38].

Antoniol *et al.* [1] aplicam um algoritmo chamado Dynamic Time Warping (DTW) para identificar grupos de arquivos logicamente dependentes. Esse algoritmo foi desenvolvido na área de reconhecimento de padrões e foi adotado em um sistema de reconhecimento de fala. Os autores reportam e discutem os resultados de uma aplicação preliminar da abordagem no repositório CVS do Mozilla.

Fluri e Gall [14] argumentam que algumas mudanças, como inclusão de comentários, não tem significância relevante. Assim, os autores propõem um mecanismo (e uma ferramenta) para classificar as mudanças e então determinar sua significância. Os autores elaboram, por fim, uma taxonomia de mudanças que informa o nível de significância para cada tipo de mudança. Com base em tal taxonomia, os autores desenvolvem um filtro para dependências lógicas, de modo que dependências insignificantes são ignoradas.

Canfora *et al.* [8] observam a mudança conjunta de linhas de código (*line co-change*) a fim de identificar interesses transversais em código hospedado no CVS. Os autores definem *line co-change* como um enriquecimento das transações de modificação (MTs) por meio de informação histórica sobre linhas. Em particular, uma MT no CVS é dada por:

$MT = \{r_1, r_2, \ldots, r_k\}\ \forall\ i, j \in \{1, \ldots, k\}$, the following conditions holds:

$file(r_i) = file(r_j): j = i$
$author(r_i) = author(r_j)$
$branch(r_i) = branch(r_j)$
$notes(r_i) = notes(r_j)$
$time(r_i) - time(r_i - 1) \leq 200s : i \geq 2$

A MT enriquecida com informações histórica sobre linhas é dada por: $MT = \{L_{r1}, L_{r2}, \ldots, L_{rk}\}$, onde L_{ri} é o conjunto de linhas na release atual de file(r_i) que foram mudadas ou adicionadas na revisão r_i. Por exemplo, a sequência de MTs executadas no arquivo texto da

Figure 6 é: $MT_1 = \{\{2,3,4\}\}$; $MT_2 = \{\{4\}\}$; $MT_3 = \{\{1,2,3\}\}$; $MT_4 = \{1,2,3\}\}$. A **Tabela 1** ilustra essa situação.

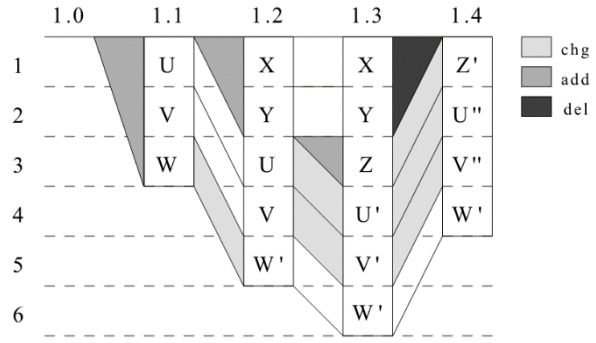

Figure 6. Exemplo de 4 MTs (cada uma contendo uma única revisão) modificando um arquivo texto

1.1	1.2	1.3	1.4	line #
		a	c	1
a		c	c	2
a		c	c	3
a	c			4

Tabela 1: Tabela de histórico de linha para revisão 1.4

Wenzel *et al.* [35] usam um algoritmo de diferenciação chamado *SiDiff* para identificar elementos similares em diferentes versões de um modelo. Deste modo, os autores conseguem detectar acoplamento lógico entre elementos de um modelo. A Figura 7 ilustra esse cenário. Assim, numa revisão específica (1.3.2.1), o círculo branco denota o elemento do modelo a ser observado. Os elementos que dependem logicamente dele são coloridos de verde, passando por amarelo e chegando em vermelho de acordo com a intensidade. A intensidade é calculada com base na medida de confiança da dependência lógica.

Figura 7. Acoplamento lógico entre elementos de um modelo [35]

Robbes *et al.* [33] argumentam que a identificação de acoplamento lógico pode ser aperfeiçoada ao se utilizar informações de mais baixo nível obtidas em tempo de desenvolvimento. Os autores apresentam uma ferramenta (desenvolvida em um estudo anterior) que grava todas as mudanças feitas em um sistema enquanto ele está sendo desenvolvido e as armazena em um repositório de mudanças (*change repository*). Nesse cenário, as informações sobre mudanças retêm a informação de tempo precisa a respeito de quando uma mudança foi efetivamente realizada e consegue ainda distinguir entre classes e métodos (e não somente arquivos).

Wang *et al.* [34] propõem um método para identificação de dependências lógicas entre funções/métodos (*software entities*) no CVS. Primeiro, as transações de mudança são reconstruídas usando a técnica de janela fixa de tempo, conforme [37]. Em seguida, transações que envolvem muitos arquivos e ou que contém mensagens de log vazias são descartadas. Após isso, uma estratégia *fuzzy* é utilizada para extrair informações de mais baixo nível, isto é, as transações de mudança são mapeadas para as funções/métodos. Por

fim, um mecanismo é proposto para classificar as dependências lógicas entre funções/métodos em relações estruturadas (*strutured relations*) e relações não estruturadas (*unstructured relations*).

Investigação da relação entre dependências estruturais e lógicas. Fluri *et al.* [15] investigam o grau em que dependências lógicas são causadas por acoplamentos estruturais (*source code coupling*) ou por modificações textuais (como atualização de licenças). A estratégia aplicada pelos autores é sumarizada na Figura 8. A primeira avaliação dos autores em um software livre de médio porte mostrou que uma quantidade razoável de dependências lógicas (*change couplings*) não são causadas por mudanças em código fonte.

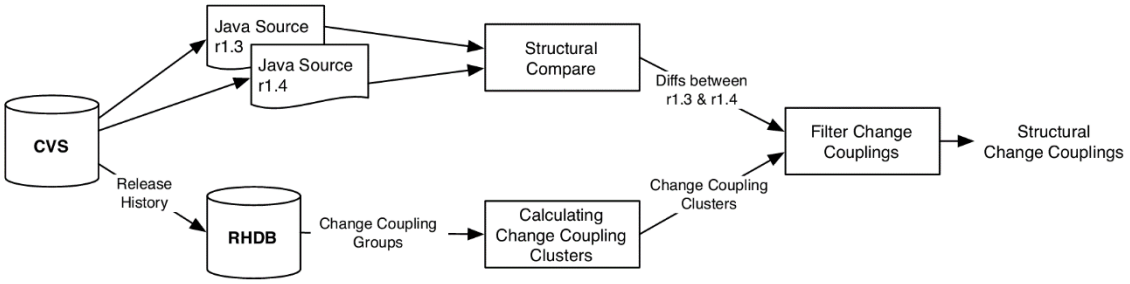

Figura 8. Fluxo das informações e etapas de processamento [15]

Hanakawa [18] estudou a relação entre os conjuntos de elementos altamente acoplados estruturalmente (M) e de elementos altamente acoplados logicamente (L). A hipótese colocada pela autora é de que a média da interseção entre M e L tende a diminuir ao longo do tempo. Isso ocorre por conta de (i) um acréscimo de ações de "copiar e colar" código (resultando em acoplamento lógico apenas) e (ii) desenvolvedores esquecendo-se de fazer o commit de classes estruturalmente relacionadas de uma vez só (resultando em acoplamento estrutural apenas). Hanakawa propõe uma métrica de complexidade baseada no tamanho de tal interseção (Figura 9). Embora os resultados obtidos confirmem que a interseção entre M e L de fato diminui (isto é, a complexidade aumenta) na maior parte dos casos estudados, há pouca evidência de que tais resultados derivam das suposições da autora. Isso é particularmente evidente na análise de complexidade realizada envolvendo a ferramenta JUnit.

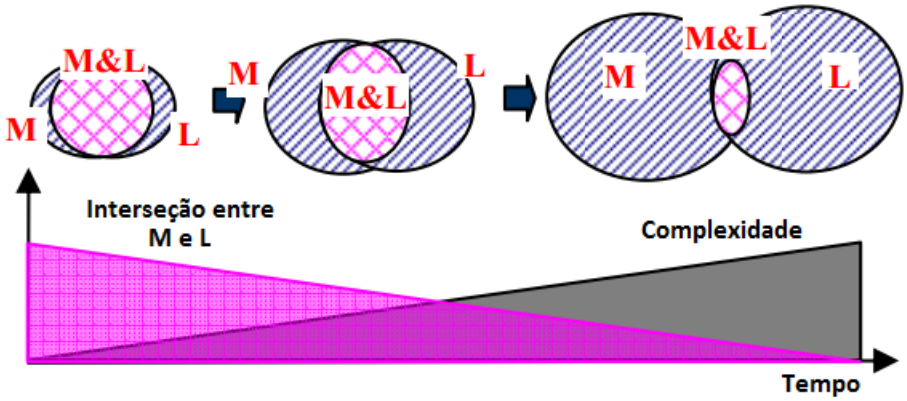

Figura 9. Relação entre interseção (de M e L) e complexidade [18]

Origens das dependências lógicas. Conforme anteriormente comentado, Fluri e Gall [14] desenvolvem uma taxonomia para tipos de mudança e suas respectivas significâncias. A Figura 10 mostra um resumo da taxonomia.

Change Type	*Operation*	*Significance Level*
Body-Part		
Additional Functionality*	$\langle$ INS$((l(M), n(M)), C, k)$, INS$((l(P(M)), \{\}), M, 4)\rangle$	low
Additional Object State*	$\langle$ INS$((l(A), n(A)), C, k)$ + INS$((l(T(A)), v(T(A))), A, 1)\rangle$	low
Condition Expression Change	UPD$(L, v(CE_{new}(L)))$; UPD$(CS, v(CE_{new}(CS)))$	medium
Decreasing Statement Delete	DEL$(s), \delta_{old}(M) > \delta_{new}(M)$	high
Decreasing Statement Parent Change	MOV$(s, y, k), p_{old}(s) \neq p_{new}(s)$, $\delta_{old}(M) > \delta_{new}(M)$	high
Else-Part Insert	INS$((l(EP), \{\}), CS, 1)$	medium
Else-Part Delete	DEL(EP)	medium
Increasing Statement Insert	INS$((l(s), v(s)), y, k), \delta_{old}(M) < \delta_{new}(M)$	high
Increasing Statement Parent Change	MOV$(s, y, k), p_{old}(s) \neq p_{new}(s)$, $\delta_{old}(M) < \delta_{new}(M)$	high
Removed Functionality	DEL(M)	crucial
Removed Object State	DEL(A)	crucial
Statement Delete	DEL(s)	medium
Statement Insert	INS$((l(s), v(s)), y, k)$	medium
Statement Ordering Change	MOV$(s, p(s), k)$	low
Statement Parent Change	MOV$(s, y, k), p_{old}(s) \neq p_{new}(s)$	medium
Statement Update*	UPD(s, val)	low
Declaration-Part		
Class Renaming*	UPD$(C, v(n_{new}(C)))$	high
Decreasing Accessibility Change[1]	INS$((l(\mu_A), v(\mu_A)), y, 1)$; DEL$(\mu_A)$; UPD$(\mu_A, val)$	crucial
Attribute Type Change	UPD$(T(A), v(T_{new}(A)))$	crucial
Attribute Renaming*	UPD$(n(A), v(n_{new}(A)))$	high
Final Modifier Insert	INS$((l(\mu_F), v(\mu_F)), y, 2), y \in \{C, M, A\}$	crucial
Final Modifier Delete	DEL$(\mu_F(x)), x \in \{C, M, A\}$	low
Increasing Accessibility Change[1]	INS$((l(\mu_A), v(\mu_A)), y, 1)$; DEL$(\mu_A)$; UPD$(\mu_A, val)$	medium
Method Renaming*	UPD$(M, v(n_{new}(M)))$	high
Parameter Delete	DEL(ρ)	crucial
Parameter Insert	$\langle$ INS$((l(\rho), v(n(\rho))), P(M), k)$, INS$((l(T(\rho)), v(T(\rho))), \rho, 1)\rangle$	crucial crucial
Parameter Ordering Change	MOV$(\rho, P(M), k)$	crucial
Parameter Type Change	UPD$(T(\rho), v(T_{new}(\rho)))$	crucial
Parameter Renaming*	UPD$(\rho, v(n_{new}(\rho)))$	medium
Parent Class Delete	DEL$(T), T \in p_{Cold}(C)$	crucial
Parent Class Insert	INS$((l(T), v(T)), p_C(C), k)$	crucial
Parent Class Update	UPD$(T, v(T_{new})), T \in p_{Cold}(C)$	crucial
Return Type Delete	DEL$(T(M))$	crucial
Return Type Insert	INS$((l(T(M)), v(T(M))), M, 3)$, $T_{old}(M) = \{\}$	crucial
Return Type Update	UPD$(T(M), v(T_{new}(M)))$	crucial

*Functionality-Preserving, all others -Modifying

[1]possible values for $v(\mu_A)$, val, and $v_{old}(\mu_A)$ see Section 3.2

Figura 10. Classificação de tipos de mudança (extraído de [14]**)**

Aplicabilidade na indústria. Laar [24] apresenta um estudo de caso na Philips Healthcare MRI, onde o foco foi na transferência da tecnologia de identificação de dependências lógicas para a indústria. Mais especificamente, o autor desenvolveu uma ferramenta chamada CouplingViewer para identificar dependências entre módulos e a aplicou para o software da Philips. O autor argumenta que, de acordo com experts da indústria, a taxa de sinal-ruído era

muito baixa e isto inviabilizou a incorporação do CouplingViewer no conjunto de ferramentas da Philips.

2.3. Outras Aplicações de Dependências Lógicas

Assunto	Autores	Título	Ano	Veículo	Ref.
Predição de bugs, defeitos e falhas	D'Ambros, M., Lanza, Michele, Robbes, Romain	On the Relationship Between Change Coupling and Software Defects	2009	WCRE 09	[13]
	Cataldo, M., Mockus, A., Roberts, J., Herbsleb, J.	Software Dependencies, Work Dependencies, and Their Impact on Failures	2009	IEEE Transac. Soft. Engin.	[9]
Manutenção de clones	Aversano, L., Cerulo, L., Di Penta, M.	How Clones are Maintained: An Empirical Study	2007	CSMR 07	[2]
Interesses Transversais	Aversano, L., Cerulo, L., Di Penta, M.	Relating the Evolution of Design Patterns and Crosscutting Concerns	2007	SCAM 07	[3]
	Canfora, G., Cerulo, L., Di Penta, M.	On the Use of Line Co-change for Identifying Crosscutting Concern Code	2006	ICSM 06	[8]
Rastreamento de mudanças de elementos de modelos	Wenzel, S., Hutter, H., Kelter, U.	Tracing Model Elements	2007	ICSM 07	[35]
- Descoberta de design flaws e oportunidades para refatoração, reestruturação e reengenharia	Gall, H., Hajek, K., Jazayeri, M.	Detection of logical coupling based on product release history	1998	ICSM 98	[16]
	Gall, H., Jazayeri, M., Krajewski, J.	CVS release history data for detecting logical couplings	2003	IWPSE 03	[17]
	Itkonen, J., Hillebrand, M., Lappalainen, V.	Application of relation analysis to a small Java software	2004	CSMR 04	[19]
	Beyer, D., Hassan, A.	Evolution Storyboards: Visualization of Software Structure Dynamics	2006	ICPC 06	[5]
	D'Ambros, M., Lanza, M.	Reverse engineering with logical coupling	2006	WCRE 06	[11]
	D'Ambros, M., Lanza, M., Lungu, M.	Visualizing co-change information with the evolution radar	2009	IEEE Transac. Soft. Engin.	[12]
- Avaliação Arquitetural	Zimmermann, T., Diehl, S., Zeller, A.	How history justifies system architecture (or not)	2003	IWPSE 03	[37]
	Beyer, D., Noack, A.	Clustering Software Artifacts Based on Frequent Common Changes	2005	IWPC 05	[6]

Avaliação da complexidade do software	Hanakawa, N.	Visualization for Software Evolution Based on Logical Coupling and Module Coupling	2007	APSEC 07	[18]
Manter documentação, Internacionalização	Kagdi, H., Maletic, Jonathan	Mining for Co-Changes in the Context of Web Localization	2006	WSE 06	[22]

3. Predição de Mudanças em Workflows usando Dependências Lógicas

Com base nos trabalhos relacionados e na experiência dos autores desse trabalho no tópico de dependências lógicas [29–31], concebemos uma abordagem para predição de mudanças em workflows baseada em dependências lógicas. Dependências lógicas (também conhecidas como dependências de mudança [28], dependências evolucionárias [7] e co-changes [6]) são dependências implícitas que ocorrem entre artefatos de software que evoluem juntos [4, 16]. Esses artefatos não necessariamente precisar ser estruturalmente relacionados, já que estão conectados a partir de um ponto de vista evolucionário, ou seja, esses artefatos mudaram juntos com frequência no passado e, portanto, estão inclinados a mudarem juntos no futuro. Desse modo, dependências lógicas são definidas com base em um intervalo de tempo.

3.1. Premissas

Como premissas desse trabalho, temos:

- O sistema de gerenciamento de workflows (*workflow management system*) gera um commit log para cada vez que (i) um novo workflow é criado, (ii) um workflow é excluído e (iii) um workflow é modificado;
- Por mudança, entendemos qualquer tipo de mudança aplicada no esquema (schema) dos workflows
- Neste log, consta o nome do workflow, a data (*timestamp*) da alteração e o nome do desenvolvedor/designer que efetuou a mudança;

3.2. A Abordagem

As premissas anteriores permitem que dependências lógicas sejam extraídas a partir dos logs do sistema de gerenciamento de workflows. Dependências lógicas são comumente tratadas como regras de associação, que é um conceito oriundo da área de mineração de dados (data mining). Formalmente, uma regra de associação é uma implicação da forma $X_1 \Rightarrow X_2$, que diz que quando X_1 ocorre, X_2 também ocorre. X_1 e X_2 são dois conjuntos disjuntos de itens. X_1 é chamado de antecedente (também conhecido como left-hand-side e LHS) e X_2 é chamado de consequente (também conhecido como right-hand-side e RHS). Por exemplo, a regra $\{A, B\} \Rightarrow \{C\}$ encontrada nos dados das vendas de um supermercado indicaria que se um cliente compra A e B juntos, esse mesmo cliente é propenso a também comprar C. No contexto desse estudo, uma dependência lógica de um workflow f_2 (cliente) para outro workflow f_1

(fornecedor) é denotada pela regra $F_1 \Rightarrow F_2$, na qual o antecedente e o consequente são conjuntos unitários que contém os workflows f_1 e f_2 respectivamente.

Medidas de interesse e significância para regras de associação são usualmente dadas por limiares (*thresholds*) de suporte (*support*) e confiança (*confidence*). Em nosso estudo, a medida de suporte denota o número de vezes que dois artefatos foram mudados juntos. A medida de confiança define o grau em que artefatos estão logicamente conectados, caracterizando assim a força da relação. Zimmerman *et al.* [38] formalizaram esses conceitos para o sistema de controle de versão CVS, isto é, um sistema sem suporte à commit atômico. Nós adaptamos tal formalização de modo que se tornasse apropriada para o nosso contexto. Descrevemos, a seguir, a formalização adaptada:

- Frequência de um conjunto de arquivos F em um conjunto de commits C:

$frq(C, F) = |\{c \mid c \in C \text{ e } c \text{ inclui todos os arquivos em } F\}| = P(F)$

- Suporte de uma dependência lógica $F_1 \Rightarrow F_2$:

$supp(C, F_1 \Rightarrow F_2) = frq(C, F_1 \cup F_2) =$ número de commits que contém ambos f_1 e f_2

- Confiança de uma dependência lógica $F_1 \Rightarrow F_2$:

$conf(C, F_1 \Rightarrow F_2) = supp(C, F_1 \Rightarrow F_2) / frq(C, F_1) =$ valor de suporte dividido pelo número de commits que contém f_1

Deve-se notar que os valores de confiança para $F_1 \Rightarrow F_2$ e $F_2 \Rightarrow F_1$ são diferentes. No primeiro caso, o valor de confiança determina, por definição, o grau em que o workflow f_2 é cliente de outro workflow f_1. Analogamente, no segundo caso, a confiança determina o grau em que o workflow f_1 é cliente de f_2. Para ilustrar essa sutil diferença, considere o exemplo mostrado na Figura 11.

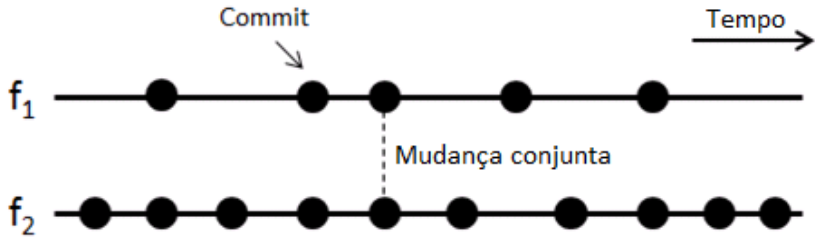

Figura 11. Exemplo de regra de associação

Na maior parte das vezes, quando f_1 é incluído em um commit, f_2 também é. Portanto, a regra $F_1 \Rightarrow F_2$ (que diz que f_2 depende de f_1) tem uma alta confiança: $conf(C, F_1 \Rightarrow F_2) =$ número de commits que contem ambos f_1 e f_2 / número de commits que contém $f_1 = 4/5 = 80\%$. Por outro lado, a regra $F_2 \Rightarrow F_1$ (que diz que f_1 depende de f_2) tem metade da confiança da regra anterior: $conf(C, F_2 \Rightarrow F_1) =$ número de commits que contem ambos f_1 e f_2 / número de commits que contém $f_2 = 4/10 = 40\%$.

Assim, dado um conjunto de logs referente a um período de tempo t, podemos extrair dependências lógicas na forma de regras de associação e predizer novas co-changes a partir do instante t+1. Assim, quando um workflow é modificado, podemos inspecionar o conjunto de regras de associação e fornecer sugestões para o usuário final a respeito de quais outros workflows possivelmente precisam ser modificados em conjunto.

4. Conclusão e Trabalhos Futuros

Nesse trabalho, apresentamos uma abordagem para predição de mudanças em workflows baseada em dependências lógicas. Utilizando a mesma ideia central do trabalho de Zimmermann *et al.* [38], nós analisamos change logs e extraímos dependência lógicas na forma de regras de associação. De posse destas tais regras, podemos inferir com alguma certeza (definida em termos dos valores de suporte e confiança) quais workflows outros precisam ser modificados quando um dado workflow é modificado. Como trabalho futuro, precisamos refinar a abordagem e avalia-la em um sistema real de gerenciamento de workflows.

Referências

[1] Antoniol, G., Rollo, V.F. and Venturi, G. 2005. Detecting groups of co-changing files in CVS repositories. *Principles of Software Evolution, Eighth International Workshop on* (2005), 23–32.

[2] Aversano, L., Cerulo, L. and Di Penta, M. 2007. How Clones are Maintained: An Empirical Study. *Proceedings of the 11th European Conference on Software Maintenance and Reengineering* (Washington, DC, USA, 2007), 81–90.

[3] Aversano, L., Cerulo, L. and Penta, M.D. 2007. Relating the Evolution of Design Patterns and Crosscutting Concerns. *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation* (Washington, DC, USA, 2007), 180–192.

[4] Ball, T., Adam, J.-M.K., Harvey, A.P. and Siy, P. 1997. If Your Version Control System Could Talk... *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering* (Mar. 1997).

[5] Beyer, D. and Hassan, A.E. 2006. Evolution Storyboards: Visualization of Software Structure Dynamics. *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on* (2006), 248–251.

[6] Beyer, D. and Noack, A. 2005. Clustering Software Artifacts Based on Frequent Common Changes. *Proceedings of the 13th International Workshop on Program Comprehension* (Washington, DC, USA, 2005), 259–268.

[7] Burch, M., Diehl, S. and Weissgerber, P. 2005. Visual data mining in software archives. *Proceedings of the 2005 ACM symposium on Software visualization* (St. Louis, Missouri, 2005), 37–46.

[8] Canfora, G., Cerulo, L. and Di Penta, M. 2006. On the Use of Line Co-change for Identifying Crosscutting Concern Code. *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on* (2006), 213–222.

[9] Cataldo, M., Mockus, A., Roberts, J.A. and Herbsleb, J.D. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Trans. Softw. Eng.* 35, 6 (Nov. 2009), 864–878.

[10] D'Ambros, M., Gall, H., Lanza, M. and Pinzger, M. 2008. Analysing Software Repositories to Understand Software Evolution. *Software Evolution*. T. Mens and S. Demeyer, eds. Springer. 37–67.

[11] D'Ambros, M. and Lanza, M. 2006. Reverse Engineering with Logical Coupling. *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on* (2006), 189–198.

[12] D'Ambros, M., Lanza, M. and Lungu, M. 2009. Visualizing Co-Change Information with the Evolution Radar. *IEEE Trans. Software Eng*. 35, (2009), 720–735.

[13] D'Ambros, M., Lanza, M. and Robbes, R. 2009. On the Relationship Between Change Coupling and Software Defects. *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France* (2009), 135–144.

[14] Fluri, B. and Gall, H.C. 2006. Classifying Change Types for Qualifying Change Couplings. *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on* (2006), 35–45.

[15] Fluri, B., Gall, H.C. and Pinzger, M. 2005. Fine-grained analysis of change couplings. *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on* (2005), 66–74.

[16] Gall, H., Hajek, K. and Jazayeri, M. 1998. Detection of Logical Coupling Based on Product Release History. *Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 1998), 190–.

[17] Gall, H., Jazayeri, M. and Krajewski, J. 2003. CVS Release History Data for Detecting Logical Couplings. *Proceedings of the 6th International Workshop on Principles of Software Evolution* (Washington, DC, USA, 2003), 13–.

[18] Hanakawa, N. 2007. Visualization for Software Evolution Based on Logical Coupling and Module Coupling. *Proceedings of the 14th Asia-Pacific Software Engineering Conference* (Washington, DC, USA, 2007), 214–221.

[19] Itkonen, J., Hillebrand, M. and Lappalainen, V. 2004. Application of relation analysis to a small Java software. *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on* (Mar. 2004), 233–239.

[20] Kagdi, H., Gethers, M., Poshyvanyk, D. and Collard, M.L. 2010. Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code. *Reverse Engineering (WCRE), 2010 17th Working Conference on* (2010), 119–128.

[21] Kagdi, H. and Maletic, J.I. 2007. Combining Single-Version and Evolutionary Dependencies for Software-Change Prediction. *Proceedings of the Fourth International Workshop on Mining Software Repositories* (Washington, DC, USA, 2007), 17–.

[22] Kagdi, H. and Maletic, J.I. 2006. Mining for Co-Changes in the Context of Web Localization. *Web Site Evolution, 2006. WSE '06. Eighth IEEE International Symposium on* (2006), 50–57.

[23] Kagdi, H. and Maletic, J.I. 2006. Software-Change Prediction: Estimated+Actual. *Software Evolvability, 2006. SE '06. Second International IEEE Workshop on* (2006), 38–43.

[24] Laar, P. van de 2009. On the transfer of evolutionary couplings to industry. *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on* (May. 2009), 187–190.

[25] Lehman, M., Perry, D., Ramil, J., Turski, W. and Wernick, P. 1997. Metrics and Laws of Software Evolution–The Nineties View. *Proceedings IEEE International Software Metrics Symposium (METRICS'97)* (Los Alamitos CA, 1997), 20–32.

[26] Malik, H., Chowdhury, I., Tsou, H.-M., Jiang, Z.M. and Hassan, A.E. 2008. Understanding the rationale for updating a function's comment. *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on* (2008), 167–176.

[27] Malik, H. and Shakshuki, E. 2010. Predicting Function Changes by Mining Revision History. *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on* (Apr. 2010), 950–955.

[28] Mens, T. and Demeyer, S. 2008. *Software Evolution*. Springer Publishing Company, Incorporated.

[29] Oliva, G.A. and Gerosa, M.A. 2011. On the Interplay between Structural and Logical Dependencies in Open-Source Software. *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering* (Washington, DC, USA, 2011), 144–153.

[30] Oliva, G.A., Santana, F.W.S., Gerosa, M.A. and Souza, C.R.B. de 2012. Preprocessing Change-Sets to Improve Logical Dependencies Identification. *Proceedings of the 6th International Workshop on Software Quality and Maintainability* (Szeged, Hungary, 2012).

[31] Oliva, G.A., Santana, F.W.S., Gerosa, M.A. and Souza, C.R.B. de 2011. Towards a classification of logical dependencies origins: a case study. *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution* (Szeged, Hungary, 2011), 31–40.

[32] Pirklbauer, G., Fasching, C. and Kurschl, W. 2010. Improving Change Impact Analysis with a Tight Integrated Process and Tool. *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations* (Washington, DC, USA, 2010), 956–961.

[33] Robbes, R., Pollet, D. and Lanza, M. 2008. Logical Coupling Based on Fine-Grained Change Information. *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on* (2008), 42–46.

[34] Wang, X., Wang, H. and Liu, C. 2009. Predicting Co-Changed Software Entities in the Context of Software Evolution. *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on* (2009), 1–5.

[35] Wenzel, S., Hutter, H. and Kelter, U. 2007. Tracing Model Elements. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on* (2007), 104–113.

[36] Zhou, Y., Wursch, M., Giger, E., Gall, H. and Lu, J. 2008. A Bayesian Network Based Approach for Change Coupling Prediction. *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on* (2008), 27–36.

[37] Zimmermann, T., Diehl, S. and Zeller, A. 2003. How history justifies system architecture (or not). *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of* (2003), 73–83.

[38] Zimmermann, T., Weissgerber, P., Diehl, S. and Zeller, A. 2005. Mining Version Histories to Guide Software Changes. *IEEE Trans. Softw. Eng.* 31, 6 (Jun. 2005), 429–445.