Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Programa de Pós-Graduação em Ciência da Computação

Habilitando a Checagem Estática de Conformidade Arquitetural de Software em Evolução

Roberto Almeida Bittencourt

Tese submetida à Coordenação do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande, Campus I, como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dalton Dario Serey Guerrero
Gail C. Murphy
(Orientadores)

Campina Grande, Paraíba, Brasil

Federal University of Campina Grande
Center for Electrical Engineering and Informatics
Computer Science Graduate Program

Enabling Static Architecture Conformance Checking of Evolving Software

Roberto Almeida Bittencourt

Dissertation submitted to the Coordination of the Computer Science Graduate Program of the Federal University of Campina Grande, Campus I, in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Research Area: Computer Science
Research Line: Software Engineering

Dalton Dario Serey Guerrero
Gail C. Murphy
(Advisors)

Campina Grande, Paraíba, Brazil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

B624h Bittencourt, Roberto Almeida.

Habilitando a checagem estática de conformidade arquitetural de software em evolução / Roberto Almeida Bittencourt. – Campina Grande, 2012.

208 f. : il.

Tese (Doutorado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Orientadores: Prof. Dr. Dalton Dario Serey Guerrero, Prof. Dr. Gail C. Murphy.

Referências.

1. Arquitetura de Software. 2. Engenharia de Software. 3. Evolução de Software. 4. Checagem de Conformidade Arquitetural. 5. Modelos de Reflexão. I. Título.

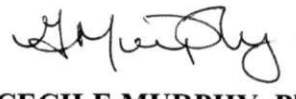
CDU 004.2(043)

"HABILITANDO A CHECAGEM ESTÁTICA DE CONFORMIDADE ARQUITETURAL DE SOFTWARE EM EVOLUÇÃO"

ROBERTO ALMEIDA BITTENCOURT

TESE APROVADA EM 29/02/2012

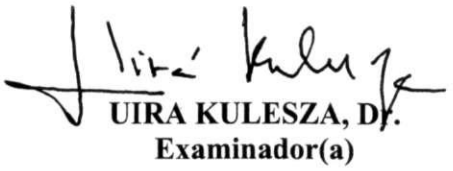
DALTON DARIO SEREY GUERRERO, D.Sc
Orientador(a)



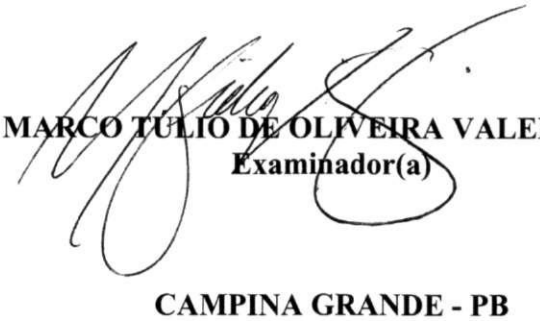
GAIL CECILE MURPHY, Ph.D
Orientador(a)

JACQUES PHILIPPE SAUVÉ, Ph.D
Examinador(a)

TIAGO LIMA MASSONI, Dr.
Examinador(a)



UIRA KULESZA, Dr.
Examinador(a)



MARCO TÚLIO DE OLIVEIRA VALENTE, Dr.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

A técnica dos modelos de reflexão é um processo de checagem de conformidade entre visões arquiteturais modulares e implementação que permite prevenir e remediar o envelhecimento de software através do combate à deterioração arquitetural. Contudo, o esforço manual necessário para aplicar a técnica pode terminar evitando seu uso na prática, especialmente no contexto de evolução de software em processos de desenvolvimento leves. Em termos mais específicos, a técnica é custosa para: *i*) produzir um modelo de alto nível e o mapeamento entre as entidades do código-fonte e este modelo; *ii*) manter tanto o modelo como o mapeamento atualizados à medida que o software evolui; e *iii*) analisar a normalmente longa lista de violações arquiteturais no código fonte.

Este trabalho procura habilitar a checagem de conformidade estática de software em evolução através da automação parcial do esforço manual para aplicar a técnica de modelos de reflexão. Para fazê-lo, primeiramente é avaliado o potencial de técnicas de agrupamento para a geração e manutenção de modelos de alto nível. Também é proposta e avaliada uma técnica de mapeamento incremental entre entidades do código-fonte e modelos de alto nível baseada na combinação da recuperação de informação de vocabulário de software com dependências estruturais. Por fim, uma técnica de priorização baseada na história do software para recomendar as violações arquiteturais no código-fonte mais provavelmente relevantes do ponto de vista dos desenvolvedores de software é relatada e avaliada.

Técnicas de agrupamento são avaliadas através de medidas de acurácia e estabilidade. Os resultados para quatro diferentes algoritmos de agrupamento mostram que nenhum deles consegue o melhor desempenho para todas as medidas, e que todos eles apresentam limitações para prover a geração automática de modelos de alto nível. Por outro lado, a avaliação sugere que a etapa de mapeamento da técnica de modelos de reflexão pode ser habilitada pela técnica proposta de mapeamento incremental automático que combina estrutura e vocabulário. Em dois estudos de caso, esta técnica obteve os maiores valores de medida-F em mudanças de código-fonte unitárias, pequenas ou grandes. Finalmente, a avaliação da técnica de priorização de violações mostra que, de cinco fatores estudados, a duração da violação e a co-locação da violação correlacionam bem com a relevância das violações. Os resultados sugerem que estes fatores podem ser usados para ordenar as violações mais

provavelmente relevantes, com uma melhoria de pelo menos 57% em relação a uma linha-base de violações selecionadas aleatoriamente.

A análise dos resultados sugere que a produção de modelos de alto nível para checagem estática de conformidade arquitetural pode ser auxiliada por um processo semi-automático de recuperação arquitetural, e, à medida que o software evolui, por técnicas incrementais de agrupamento/mapeamento. Por outro lado, a análise dos resultados para a técnica de priorização sugere a eficácia de uma abordagem automatizada para a recomendação de violações arquiteturais a serem analisadas pelos desenvolvedores do software.

Abstract

The reflexion model technique is a static conformance checking technique to keep architecture module views and implementation conformant. It can either prevent or remedy software aging by combating architecture deterioration. However, the amount of manual effort to apply the technique may prevent its use in practice, especially in the context of software evolution in lightweight development methods. More specifically, it can be time-consuming and costly to: *i*) produce a high-level model and the mapping between source code entities and this model; *ii*) keep both model and mapping up-to-date as software evolves; and *iii*) analyze the usual large number of architectural violations in the source code reported by the technique.

This work tries to enable static conformance checking of evolving software by partially automating the manual effort to apply the reflexion model technique. To do so, the potential of clustering techniques to generate high-level models and keep them up-to-date is evaluated. It is also proposed and evaluated an incremental mapping approach between source code entities and high-level models based on the combination of information retrieval of software vocabulary and structural dependencies. Last, a prioritizing technique based on software history to recommend architectural violations in the source code most likely to be relevant to software developers is reported and evaluated.

Clustering techniques are evaluated by measures of accuracy and stability, and results for four different clustering algorithms show that none of them performs best for all measures, and that they are limited to provide fully automated generation of high-level models. On the other hand, evaluation suggests that the mapping step in the reflexion model technique can be enabled by the proposed incremental automated mapping technique that combines structure and vocabulary. In two case studies, the combined technique showed the highest F-measure values for both singleton, small and large source code changes. Finally, evaluation for the prioritizing technique shows that, from five studied factors, violation duration and violation co-location correlate well with violation relevance. Results suggest that these factors can be used to rank the violations most likely to be relevant, with an improvement of at least 57% against a baseline of randomly selected violations.

Analysis of the results suggests that the production of high-level models for static conformance checking can be aided by a semi-automated architecture recovery process, and, as software evolves, by incremental clustering/mapping techniques. On the other hand, analysis of the results for the prioritizing technique suggests the effectiveness of an automated approach to recommend architectural violations to be analyzed by software developers.

Acknowledgments

First, thanks to that amazing power that makes everything exist, brings order to chaos, produces beauty everywhere, and grants us the free will to choose what we want to make of our own lives, for better or for worse, whatever way they name you.

My life would have no meaning without those two special persons that conceived, fed and raised that stubborn kid with love and affection. Thanks, Painho and Mainha, for that, for supporting my education and dreams, and for helping forge my character. To my brothers Márcio and Danilo, and my sister Renacélia, for taking part in that journey with me.

To Valéria, the most beautiful encounter in my life, one that can only be described as *magic*. Val, thanks for the love, support and companionship during all these years.

A dissertation is like a journey through knowledge. But a journey without guidance is undoubtedly harder. To Dalton Serey, my advisor in this work, for believing in my work from the beginning. Dalton, thanks for introducing me to the research world of software engineering, guiding me in my first steps, ideas, drafts and papers, helping develop my critical sense, for the fruitful discussions, and, above all, for being a great friend.

But the journey is long, and one has to travel, to see how different things can be. Thanks to Gail Murphy, my advisor during my research internship at UBC, and also co-advisor in this work. Thanks, Gail, for receiving me at UBC, sharing your experience, teaching me wonderful research and writing skills, for the most effective ever discussion meetings, for supporting me in the hardest moments, for the encouragement to travel unknown paths, and simply for being the great human being you are.

Professors are a strong positive influence in our lives, be sure. To Jacques Sauvé, that taught me a lot about software engineering and research methods. To the professors in our software engineering group at UFCG, Jorge (great prof!), Franklin, Patrícia, Thiago and Rohit during the first three years of my PhD. To the profs from SPL at UBC, Eric, Gregor and Raymond for the thoughtful research meetings. And to the profs that introduced me to the research world, Antonio Ferreira, João Marques and David Carr.

Being in a research group makes the lonely activity of pursuing a PhD much more enjoyable. To the members of our software software engineering group at UFCG for the great discussions, mutual support, and great friendship. I may forget some names, but, let's see... To

Jemerson, Katyusco, Paulo, Waldemar Neto, Mirna, Pablo, Bel, João Arthur, Cássio, Lile, Ana Emília, Amanda, Diego, Gustavo Soares, Gustavo Jansen, Adauto, Zé Filho, Júnior, Emanuela, Makelli, Wilkerson, Francisco Neto, João Felipe and Andressa. Special thanks to the once undergrads Jemerson, Gustavo Jansen, Adauto and Zé Filho, for helping me on the joint effort to develop some of the tools I used in my research work, and to João Arthur, for providing the *Design Wizard* tool. To that group of once PhD students, with their meetings for mutual support and their seminars on research skills, thanks Andréa, Ayla, Rodrigo, Degas, Alexandre, Cheyenne, Raquel and Nazareno. Not in these lists, but very important during these years: Guilherme, for the great discussions on software architecture and development; Nivaldo, for the great chats on academia, and for the support and friendship; and Christina, for the encouragement, trust and friendship.

To the students and post-docs from the Software Practices Lab at UBC, for the great research environment, the cool presentations, great questions both on the whole and on the details, thoughtful comments, and for the fellowship and support that a foreign student needs most. I may forget someone, but, here it goes... To Emerson, Thomas, Sarah, Alex, Nick, Terceiro, James, Dheeraj, Peter, Robin, Nima, Peng, Adrian, Julius, Albert, Rahul and Neil. Also at UBC, thanks to Lauro, Elizeu and Abdullah, for the company for lunch and coffee breaks, friendship and support.

To the UFCG staff, Lilian, Aninha, Vera and Rebeka, for helping me in the lab and at COPIN. To the UBC staff, Hermie Lam and Holly Kwan, for the nice chats, and for helping my way around UBC. To Sara, for the counseling.

To the amazing people I met in Vancouver, both from Canada, Brazil and many other places, people that help make this life experience so unique and unforgettable.

To my university, UEFS - State University of Feira de Santana, for granting me a leave to pursue this PhD, and supporting my research work during these years.

To the Brazilian funding agencies CAPES, CNPq, FINEP and FAPESB, for either directly or indirectly funding this research work.

Last but not least, to the examining committees both of the dissertation proposal and the dissertation itself, for the feedback and contributions to the improvement of this work.

Sumário

Contents

List of Acronyms

ADL - *Architecture Description Language*

API - *Application Programming Interface*

AST - *Abstract Syntax Tree*

DCL - *Dependency Constraint Language*

DSM - *Design Structure Matrix*

ERM - *Evolutionary Reflexion Model*

GUI - *Graphical User Interface*

IDE - *Integrated Development Environment*

IR - *Information Retrieval*

GXL - *Graph eXchange Language*

LOC - *Lines Of Code*

MDA - *Model Driven Architecture*

MoJo - *Moves and Joins*

MVC - *Model-View-Controller*

NED - *Non-Extremity of Cluster Distribution*

RM - *Reflexion Model*

RPA - *Relation Partition Algebra*

SCM - *Software Configuration Management*

SQA - *Software Quality Assurance*

SQL - *Structured Query Language*

SVM - *Support Vector Machine*

UML - *Unified Modeling Language*

XML - *eXtensible Markup Language*

List of Figures

List of Tables

Chapter 1

Introdução

Neste capítulo, uma contextualização desta tese é apresentada ao leitor. O problema central da tese é apresentado, assim como um processo é proposto como solução para o problema. Em seguida, um sumário da tese é descrito, assim como a organização geral deste trabalho.

1.1 Contextualização

Provocadas por novas solicitações de requisitos, mudanças em software são, de modo geral, inevitáveis. Como resultado, a maioria dos sistemas de software cresce com o passar do tempo [Lehman et al. 1997]. Neste contexto de crescimento do software, cada mudança realizada num sistema pode demandar a análise de mais linhas de código à medida que o tempo passa, assim como pode requerer uma maior interação entre os desenvolvedores que mantêm o sistema.

A manutenção de grandes sistemas de software geralmente requer a existência de modelos mentais compartilhados pela equipe de desenvolvimento. Tais modelos ajudam os desenvolvedores a refletir sobre conceitos do software, a discutir estes conceitos com partes interessadas, e a tomar decisões que mudem a estrutura e o comportamento do software. Brooks define *integridade conceitual* como a uniformidade do modelo mental que a equipe de desenvolvimento tem sobre o software [Brooks 1995].

As equipes envolvidas com software em evolução geralmente têm dificuldade em reter a integridade conceitual devido à complexidade e ao tamanho destes sistemas. Brooks argumenta que é melhor ter um conjunto simples e coeso de ideias sobre o design do software do

que muitas boas ideias independentes e não-coordenadas [Brooks 1995].

A manutenção da integridade conceitual pela preservação e modificação apropriadas do design do software é um desafio em sistemas de software em evolução. Frequentemente, quando grandes equipes de software fazem evoluir grandes sistemas de software, os sistemas experimentam uma quebra da modularidade, um maior impacto das mudanças no software e um aumento no potencial de faltas [Eick et al. 2001], sintomas usuais desta perda de integridade. A questão da preservação da integridade conceitual é o tema amplo desta tese, sobre o qual tecerei detalhes neste capítulo e ao longo deste trabalho.

A produção de uma arquitetura de software explícita para um sistema é uma forma de materializar a integridade conceitual deste sistema. Arquitetura de software pode ser definida de diferentes maneiras. Uma definição popular afirma que ela é "a estrutura dos componentes de um programa/sistema, suas interrelações, e os princípios e linhas gerais que governam o seu design e sua evolução ao longo do tempo" [Garlan and Perry 1995, p. 269]. Quando documentada explicitamente, a arquitetura de software pode ter um impacto positivo em várias atividades de desenvolvimento de software (e.g., compreensão, construção, validação e reuso), assim como no gerenciamento de sistemas grandes e complexos (e.g., planejamento do projeto, alocação de equipes de desenvolvimento) [Garlan 2000]. Os arquitetos de software procuram desenvolver arquiteturas reutilizáveis e adaptáveis que se mantenham praticamente intactas quando de mudanças previstas, especialmente em sistemas de domínios voláteis tais como o bancário, o de telecomunicações e o de *e-business*.

Antes de discutir outros aspectos, é necessário esclarecer uma questão de terminologia. Nesta tese, o termo *evolução de software* se restringe à visão de evolução como fenômeno que pode ser *observado*, conforme expresso por Lehman et al. nas leis de evolução de software, e não com a outra visão usual do termo, que lida com métodos, técnicas e atividades para *controlar* a evolução de software [Madhavji et al. 2006].

Perry e Wolf cunharam o termo *deriva arquitetural* para expressar a falta de sensibilidade dos desenvolvedores de software em relação à arquitetura de software [Perry and Wolf 1992], problema diretamente relacionado à falta de integridade conceitual.

Falta de integridade conceitual pode levar a um fenômeno conhecido como envelhecimento de software [Parnas 1994]. De acordo com Parnas, software envelhecido é caracterizado pelo aumento na dificuldade de adaptação a novas funcionalidades demandadas pelos

clientes, pela sua crescente complexidade, pela redução da confiabilidade causada pela introdução de bugs durante a sua manutenção, e pela deterioração gradual da estrutura do software.

Software envelhecido possui alguns sintomas típicos: *i*) a cada nova versão, torna-se mais difícil adicionar novas funcionalidades; *ii*) o desempenho no tempo ou no uso de memória se degrada; *iii*) faltas tendem a ser introduzidas em cada mudança; *iv*) dificuldade em rastrear ou reconstruir decisões de design; *v*) aumento no *churn* de código e na frequência de faltas descobertas [Parnas 1994; Eick et al. 2001; van Gurp and Bosch 2002].

A deterioração estrutural de um sistema de software quando de sua evolução já recebeu outras denominações tais como erosão arquitetural [Perry and Wolf 1992], erosão de design [van Gurp and Bosch 2002], decaimento de código [Eick et al. 2001] ou degeneração arquitetural [Hochstein and Lindvall 2003]. Deterioração estrutural pode ou não redundar em envelhecimento de software, embora isto tipicamente ocorra [Parnas 1994; Eick et al. 2001; van Gurp and Bosch 2002]. Dependerá essencialmente se a estrutura degradada provoca ou não o aumento na dificuldade de adaptação a novas funcionalidades. Neste trabalho, eu estou preocupado com os cenários onde deterioração estrutural resulta em envelhecimento de software, e eu argumentarei que a chave para combater o envelhecimento de software está na manutenção adequada da estrutura do software à medida que ele evolui.

Segundo Parnas, há duas formas de combater o envelhecimento de software: prevenir e remediar [Parnas 1994]. Várias práticas preventivas podem retardar o envelhecimento do software, mas não são capazes de interrompê-lo, já que a capacidade dos projetistas de prever mudanças é limitada pela impossibilidade de prever adequadamente os desejos futuros dos clientes. Portanto, com o envelhecimento inevitável, remediá-lo torna-se uma questão importante em evolução de software.

Nesse contexto, surgem as seguintes questões. Como o conhecimento da arquitetura do software pode ajudar a prevenir ou remediar o envelhecimento do software? Além disso, dado que as várias metodologias leves de desenvolvimento de software usam pouca ou nenhuma documentação arquitetural, como a integridade conceitual pode ser reforçada em tais processos sem modificar a sua natureza leve?

Ainda que arquiteturas de software sejam documentadas, há uma lacuna entre o que é documentado e o que é, de fato, implementado [Lindvall and Muthig 2008]. Isto é, docu-

mentação sozinha não é capaz de garantir integridade conceitual completa. É comum ocorrer de a documentação arquitetural não ser atualizada frequentemente, mesmo quando novas decisões de design são tomadas. É também frequente que desenvolvedores ignorem regras arquiteturais, mesmo com a documentação disponível. Neste cenário, checagem de conformidade arquitetural pode ajudar a preencher esta lacuna. Com a introdução de checagens frequentes entre arquitetura e implementação, a integridade conceitual é mantida, reduzindo o ritmo de deterioração arquitetural e, consequentemente, retardando o processo de envelhecimento de software.

Checagem de conformidade da arquitetura de software pode ser vista como um caso especial de checagem de conformidade entre especificação e implementação, um tema de pesquisa extensamente explorado pela comunidade de métodos formais. Pode ser vista ainda com uma técnica de avaliação arquitetural tardia, na qual a arquitetura real do software é comparada com a arquitetura planejada e os desvios observados conduzem a medidas corretivas [Tvedt et al. 2002]. A pesquisa em checagem de conformidade arquitetural está focada principalmente na conformidade de visões modulares estáticas. Visões modulares são diagramas arquiteturais estruturais nos quais os módulos são entidades que agregam unidades de código conjuntamente responsáveis pela implementação de um conjunto de responsabilidade, e as relações entre módulos descrevem dependências de código entre módulos [Clements et al. 2002]. Checagem de conformidade de visões modulares, portanto, compara entidades e relações em uma visão modular planejada com a visão modular real recuperada da implementação, e a literatura relacionada é prolífica neste assunto [Murphy et al. 1995; Aldrich et al. 2002; Koschke and Simon 2003; Postma 2003; Sangal et al. 2005; Knodel and Popescu 2007; Bourquin and Keller 2007; Huynh et al. 2008; Terra and Valente 2009; Feilkas et al. 2009].

Murphy e colegas tentaram resolver a questão de conformidade entre visões arquiteturais modulares e implementação através da adoção da *técnica dos modelos de reflexão* [Murphy et al. 1995; Murphy et al. 2001]. Modelos de reflexão transformam as diferenças e similaridades entre uma visão arquitetural modular e a implementação em informação explícita para o desenvolvedor de software. Sucintamente, o engenheiro define um modelo de alto nível de interesse, extrai o modelo real do código-fonte, define um mapeamento entre ambos os modelos e computa uma reflexão para observar onde ambos os

modelos concordam ou discordam. Este trabalho abriu uma nova linha de pesquisa em verificação de software e, desde então, trabalhos prolíficos em checagem de conformidade de arquitetura de software foram publicados. Não só a técnica dos modelos de reflexão (RM) é um trabalho seminal em checagem de conformidade arquitetural, mas é também uma técnica popular. Várias aplicações práticas da técnica RM na indústria já foram registradas na literatura. Aplicações variam de reengenharia [Murphy and Notkin 1997; Knodel et al. 2006], recuperação arquitetural e checagem de conformidade [Murphy et al. 1995; Murphy et al. 2001; Knodel et al. 2006; Knodel et al. 2008a; Rosik et al. 2008; Passos et al. 2010], análise da complexidade arquitetural [Lilienthal 2009], bem como de compreensão de sistemas, redocumentação e reuso de software [Knodel et al. 2006]. Adicionalmente, a técnica RM já foi estendida para lidar com aspectos adicionais tais como modelos hierárquicos [Koschke and Simon 2003], recuperação arquitetural *bottom-up* [Le Gear et al. 2005], mapeamentos automáticos [Christl et al. 2005; Christl et al. 2007], e linhas de produto de software [Frenzel et al. 2007].

Em comparação com outras técnicas de checagem de conformidade, os autores da técnica RM argumentam que ela é vantajosa por sua natureza *leve*, *aproximada* e *escalável* [Murphy et al. 1995]. Simples diagramas de caixas-e-setas capturam regras arquiteturais, sem a necessidade de linguagens de descrição arquitetural complexas, ou o uso de portas, conectores e interfaces. Estes atributos tornam a técnica particularmente útil em processos de desenvolvimento leves, onde é considerado importante evitar linguagens complexas de especificação formal e diagramas detalhados em linguagens de modelagem gráficas.

As aplicações de checagem de conformidade citadas acima mostram os benefícios da introdução desta etapa no processo do software. Ilustrados com exemplos acadêmicos e estudos de caso industriais, estes artigos revelam vários cenários de mundo real onde desenvolvedores aumentam sua integridade conceitual sobre os sistemas de software que estão desenvolvendo, após analisar violações tornadas explícitas pelas checagens de conformidade. Argumento aqui que checagem de conformidade contribui para a manutenção da integridade conceitual bem como para a redução do ritmo de deterioração arquitetural e, consequentemente, do envelhecimento do software. Argumento ainda que a técnica dos modelos de reflexão é uma técnica de checagem de conformidade bem adaptada ao contexto de processos de desenvolvimento leves. Estes argumentos serão discutidos em detalhes ao longo deste

trabalho.

Por outro lado, a técnica original de modelos de reflexão apresenta algumas limitações quando aplicada a um contexto de evolução contínua de software em processos de desenvolvimento leves. Ela requer a produção de um modelo arquitetural inicial, o que pode ser um desafio, já que, em tais processos, há pouca ou nenhuma documentação arquitetural disponível. Um outra tarefa desafiadora na técnica RM é o mapeamento entre entidades de código de baixo nível e módulos arquiteturais de alto nível, o que normalmente demanda conhecimento do domínio ou de convenções de codificação. Além disso, o mapeamento deve ser mantido atualizado quando o software evolui o que impõe uma carga adicional sobre os desenvolvedores de software. Técnicas automatizadas de mapeamento baseadas em dependências estruturais já foram desenvolvidas, mas são relativamente limitadas em capturar a semântica do mapeamento. Finalmente, a técnica RM pode produzir listas detalhadas de violações arquiteturais, geralmente em torno de centenas de violações, sobrecarregando os desenvolvedores de software a cada checagem. Algumas das violações arquiteturais identificadas podem nunca ser resolvidas pois tratam-se de exceções a uma regra geral. Outras podem precisar ser resolvidas num curto período de tempo para evitar a erosão de decisões arquiteturais importantes. Assim, a identificação de violações arquiteturais relevantes a partir das listas de violações é um aspecto desta técnica que requer melhorias. Estas limitaçõess são essencialmente relacionadas ao esforço manual gasto pelos desenvolvedores de software ao aplicar a técnica. E este esforço envolve custos que podem evitar a adoção de checagens frequentes de conformidade.

1.2 Um Processo de Checagem

Para enfrentar as limitações acima, concebi um processo de checagem de conformidade para reduzir o esforço manual durante a aplicação da técnica RM. Este processo permite automatizar parcialmente algumas das etapas para computar os modelos de reflexão. A seguir, apresento uma descrição deste processo, o qual foi denominado processo dos *modelos de reflexão evolucionários* (ERM), por manter algumas etapas da técnica RM original bem como a sua natureza leve, embora tenha sido adaptado para levar em conta a evolução do software.

O processo ERM é um processo de checagem de conformidade de visões arquiteturais modulares. A intenção é usá-lo no contexto de processos de desenvolvimento leves tais como processos ágeis ou processos de desenvolvimento de software de código aberto. Neste contexto, pode não haver uma visão arquitural modular explícita ou regras que definam a forma de interação dos módulos. Portanto, antes de checar conformidade, é necessário recuperar uma visão modular e suas regras associadas. Modelos de reflexão evolucionários são então produzidos como resultado de um subprocesso menos frequente de recuperação arquitetural e um subprocesso mais frequente de checagem de conformidade.

Recuperação arquitetural é um campo de estudo extensamente estudado. Assim sendo, não é intenção desta tese contribuir com novas técnicas de recuperação arquitetural. Ao invés disto, pretende-se avaliar algumas técnicas pré-existentes no contexto de evolução de software. No processo ERM, a recuperação arquitetural é realizada através da série de etapas descritas abaixo:

Extração de design: entidades de design de baixo nível e suas relações são extraídas do código-fonte;

Levantamento de design: entidades de design e suas relações são levantadas ao nível de tipo, onde entidades são classes ou interfaces e relações são as suas dependências;

Agrupamento de design: tipos são agrupados em módulos através de técnicas de agrupamento automáticas;

Reagrupamento semi-automático: resultados do agrupamento de design são melhorados através de técnicas semi-automáticas de reagrupamento;

Reagrupamento manual: desenvolvedores de software alteram o agrupamento resultante para melhorar os resultados automáticos e impor um ponto de vista particular sobre o agrupamento;

Definição da visão modular e das regras arquiteturais: desenvolvedores de software nomeiam os módulos e estabelecem relações entre eles de acordo com regras arquiteturais estruturais.

Depois que uma visão arquitetural modular é definida através de recuperação arquitetural, o subprocesso de checagem de conformidade pode ser aplicado durante a evolução do

sistema de software. Por exemplo, ele pode ser aplicado antes de enviar o código-fonte para o repositório de software. Deste modo, violações podem ser reveladas antes de tornar as mudanças mais definitivas. As etapas do processo de checagem de conformidade são descritas abaixo:

Mudanças arquiteturais: embora não tão frequentes como mudanças no código-fonte, mudanças arquiteturais podem ocorrer também. Nesta etapa, os desenvolvedores podem modificar a visão arquitetural modular e suas regras associadas;

Reextração e relevantamento de design: o design é reextraído para atualizar entidades e relações que podem ter sido modificadas tanto no design de baixo nível como no design no nível de tipos;

Mapeamento semi-automático: Com o código-fonte modificado, o mapeamento entre entidades no nível de tipos e os módulos arquiteturais deve ser atualizado;

(Re)mapeamento manual: O mapeamento semi-automático pode não ser suficiente para contemplar todas as entidades modificadas, e os desenvolvedores podem necessitar atualizar manualmente o mapeamento anterior;

Checagem e registro de violações: a etapa mais importante no processo, e a única obrigatória, é realizada, computando-se um modelo de reflexão. Os resultados da checagem podem ser melhorados a partir do conhecimento do histórico do software, por priorização ou por filtragem;

Resolução de violações: os desenvolvedores resolvem as violações arquiteturais através da modificação do código-fonte ou do modelo arquitetural do software.

Vale a pena ainda mencionar como o processo ERM afeta o desenvolvimento de software. Como uma técnica semi-formal leve, o processo impõe uma carga extra sobre os desenvolvedores. Entretanto, trata-se uma sobrecarga leve, já que a fase de recuperação arquitetural é menos frequente, nem todos as etapas precisam ser realizadas cada vez que a checagem é executada, e o processo não retira a atenção dos desenvolvedores de suas atividades no momento da codificação. Além do mais, com o proceso ERM, desenvolvedores não precisam aprender notações formais para checar conformidade. Modelos de reflexão podem ser gerados e checados frequentemente, sejam antes de enviar o código-fonte para o repositório,

sejam durante os *builds* noturnos ou semanais, ao mesmo tempo em que testes de unidade e de integração são executados.

O Apêndice A descreve as etapas do processo ERM em mais detalhe. Durante este trabalho, foi desenvolvida uma suíte de ferramentas protótipo, chamada de *Design Suite*, para facilitar a automação do processo ERM. O apêndice também descreve detalhes sobre a suíte de ferramentas, e como ela se encaixa no processo.

1.3 Sumário

Nesta tese, tento resolver as limitações da técnica dos modelos de reflexão quando esta é aplicada no contexto de evolução de software. Mais especificamente, eu proponho técnicas para reduzir a quantidade de esforço manual em etapas específicas da técnica: *i*) na geração dos modelos de alto nível; *ii*) no mapeamento entre o código-fonte e os modelos de alto nível; e *iii*) na análise dos resultados das checagens de conformidade.

Os resultados deste trabalho fazem parte do escopo de um processo leve de checagem de conformidade, baseado na técnica RM, que é adaptado ao contexto de evolução de software. A avaliação deste processo como um todo é complexa, pois envolve desenvolvedores de software, um conjunto de ferramentas e um horizonte de tempo mais longo. Por conseguinte, o foco deste trabalho é reduzido às três etapas do processo explicitadas acima, aquelas que demandariam um grande esforço manual caso a técnica RM original fosse aplicada. As melhorias obtidas para cada etapa são independentes entre si e podem ser usadas isoladamente por desenvolvedores de ferramentas e por pesquisadores de engenharia de software. As contribuições desta tese são, portanto, no design e na avaliação de técnicas para dar suporte a estas etapas, e são descritas a seguir:

1. Uma avaliação experimental de algoritmos de agrupamento aplicados na produção de visões modulares de alto nível e no contexto de software em evolução;

2. Design de uma técnica de mapeamento incremental para mapear entidades de código-fonte em módulos de alto nível baseada na combinação da recuperação de informação de vocabulário do software e de dependências estruturais, e uma avaliação quali-quantitativa de técnicas de mapeamento incremental;

3. Investigação de fatores de provável influência na relevância de violações arquiteturais estáticas descobertas em checagens de conformidade com a técnica RM;

4. Design e avaliação de um sistema de recomendação e de um filtro para reduzir a sobrecarga dos desenvolvedores de software quando da análise das violações apontadas nos resultados das checagens de conformidade;

5. Desenvolvimento de uma suíte de ferramentas protótipo para habilitar a automação parcial do processo proposto de checagem de conformidade.

1.4 Organização

Este documento é organizado da seguinte maneira. Neste capítulo e no próximo são introduzidos o contexto, o problema e um processo proposto como solução para o problema. O Capítulo 3 revisa o estado-da-arte em recuperação arquitetural e checagem de conformidade de visões arquiteturais modulares. Uma descrição sucinta desta tese é dada no Capítulo 4, com a caracterização do problema, questões de pesquisa, critérios de sucesso, hipóteses, objetivos de pesquisa e metodologia. Um estudo detalhado de três etapas do processo de checagem de conformidade e sua avaliação experimental são apresentados nos próximos três capítulos: a produção de modelos de alto nível a partir de técnicas de agrupamento de software, no Capítulo 5; o mapeamento semi-automático da implementação para modelos de alto nível, no Capítulo 6; e a priorização de violações resultantes das checagens de conformidade arquitetural, no Capítulo 7. Uma discussão sobre os resultados obtidos e seus desdobramentos segue no Capítulo 8. Finalmente, conclusões são apresentadas no Capítulo 9.

Chapter 2

Preliminaries

PRELIMINARES

Neste capítulo, o contexto e a motivação desta tese são apresentados ao leitor. O problema central da tese é introduzido e um processo é proposto como solução para o problema. Em seguida, um sumário da tese é apresentado, assim como uma descrição da organização do trabalho.

In this chapter, I present the context and motivation of this dissertation to the reader. The main problem is then presented, and a process is proposed as the solution to the problem. A summary of the dissertation follows, as well as a description of the organization of this work.

2.1 Context and Motivation

Driven by new feature requests, software changes are inevitable. As a result, most software systems grow over time [Lehman et al. 1997]. Each system change performed as time passes may end up requiring more lines of code to be analyzed, as well as requiring increased interaction with the typically growing number of developers working on the system. These facts lead to a progressive growth of maintenance costs along time.

Maintenance of large software systems usually requires shared mental models. Such models help developers to reason and communicate about software concepts, and to perform decisions that change software structure and behavior. Brooks defines *conceptual integrity* as the uniformity of a mental model that the software team has about the software [Brooks

1995].

Teams involved with evolving software usually have difficulty retaining conceptual integrity due to its complexity and size. Brooks argues that it is better to have a simple and cohesive set of ideas about the software design than lots of good but independent and uncoordinated ideas [Brooks 1995].

Maintaining conceptual integrity by appropriately preserving and modifying software design is a challenge in evolving software systems. Frequently, when large software teams evolve large software systems, the systems experience a breakdown of modularity, an increased span of software changes, and an increased fault potential [Eick et al. 2001], which are usual symptoms of this loss of integrity. The issue of preserving conceptual integrity is the broad theme of this dissertation, and I elaborate on it in this chapter and throughout this work.

Producing an explicit software architecture for a system is one way to help materialize the conceptual integrity of a system. Software architecture may be defined in many different ways. One popular definition states that it is "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" [Garlan and Perry 1995, p. 269]. When explictly documented, software architecture may have a positive impact on various software development activities (e.g., comprehension, construction, validation and reuse) as well as on software management (e.g., project planning, allocation of development teams) of large and complex systems [Garlan 2000]. Software architects aim to develop reusable and adaptable software architectures that remain almost intact in face of anticipated changes, especially in systems in volatile domains such as banking, telecommunications and e-business.

Before going any further, it is important to clarify some terminology. In this dissertation, the term *software evolution* is restricted to the view of evolution as a phenomenon that can be *observed*, as it was expressed by Lehman et al. in the laws of software evolution, and not with the other usual view that deals with methods, tools and activities to *control* software evolution [Madhavji et al. 2006].

Perry and Wolf coined the term *architectural drift*, to express the lack of sensitivity of software developers to the software architecture [Perry and Wolf 1992], which is related to the lack of conceptual integrity. Figure 2.1 shows a scenario of conceptual integrity, with uni-

form ideas about the software that can be expressed in a software architecture diagram. On the other hand, Figure 2.2 shows a scenario of lack of conceptual integrity, where software developers have different ideas about the design of a software system.

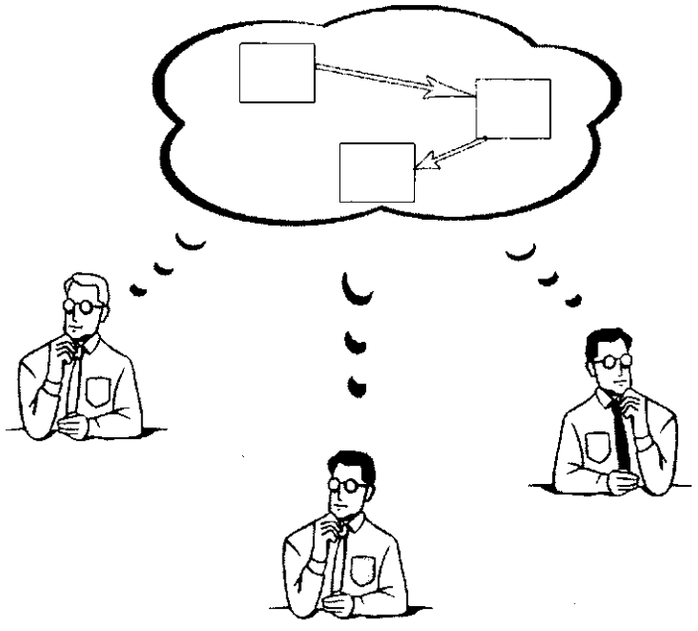


Figure 2.1: Conceptual integrity

Lack of conceptual integrity can lead to a phenomenon known as software aging [Parnas 1994]. According to Parnas, aged software is characterized by the increased difficulty to adapt to new features required by customers due to its growing complexity, by reduced reliability due to introduction of bugs during software maintenance, and by gradual deterioration of software structure.

Aged software has typical symptoms: *i*) at each version, it becomes difficult to add new features; *ii*) time or memory performance degrades; *iii*) faults tend to be introduced together with each change; *iv*) difficulty to track or reconstruct design decisions; *v*) increase in code churn and in the frequency of discovered faults [Parnas 1994; Eick et al. 2001; van Gurp and Bosch 2002].

The structural deterioration of a software system as it evolves has also been named as architectural erosion [Perry and Wolf 1992], design erosion [van Gurp and Bosch 2002], code decay [Eick et al. 2001], or architectural degeneration [Hochstein and Lindvall 2003]. Structural deterioration may end up or not in software aging, although it typically does [Par-

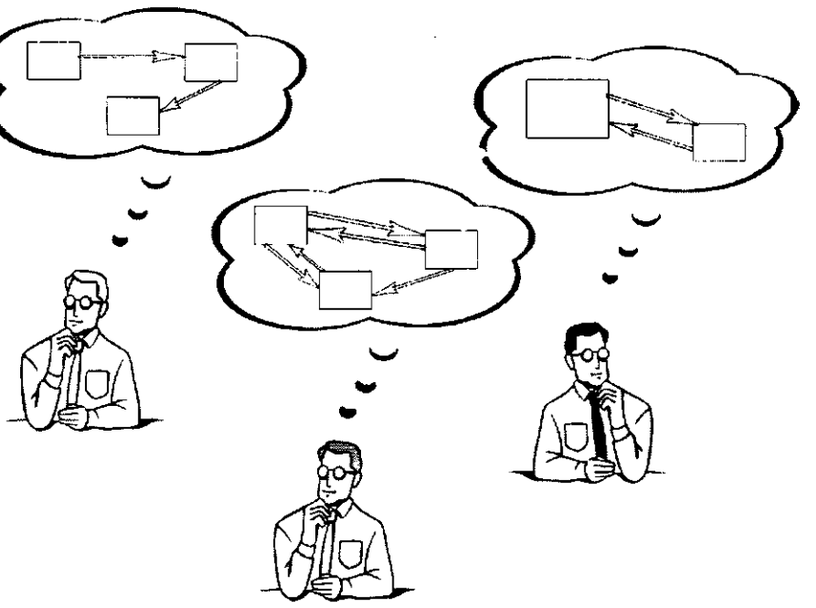


Figure 2.2: Lack of conceptual integrity and architectural drift

nas 1994; Eick et al. 2001; van Gurp and Bosch 2002]. It depends on whether or not the degraded structure increases the difficulty to adapt to new features. In this work, I am concerned about the scenarios where structural deterioration results in software aging and I will argue that the key to fighting software aging lies in maintaining adequate software structure as it evolves.

According to Parnas, there are two ways to combat software aging: prevention and remedy [Parnas 1994]. Various preventive practices can retard software aging, but are not able to stop it, since the designers' capacity of predicting changes is limited by the impossibility of adequately predicting customers' future wishes. Thus, with inevitable aging, remedy becomes an important issue in software evolution.

In this context, the following questions arise. How can the knowledge of software architecture help to prevent or remedy software aging? How can architectural drift be reduced in the process of software evolution in order to slow down software aging? Furthermore, given that various lightweight development methods use few or no architectural documentation, how can conceptual integrity be reinforced in such processes without changing their lightweight nature?

Even with documented software architectures, there is a gap between what is documented

and what is, in fact, implemented [Lindvall and Muthig 2008]. That is, documentation alone is not able to fully guarantee conceptual integrity. It usually happens that architecture documentation is not frequently updated, even when new design decisions are taken. It is also frequent that developers do not follow architectural rules, even when documentation is available. In this scenario, architecture conformance checking may help bridge this gap. With the introduction of frequent checks between architecture and implementation, conceptual integrity is sustained, which reduces structural deterioration, and, as a consequence, the process of software aging is retarded.

Conformance checking of software architecture can be seen as a special case of checking conformity between software specification and implementation, which is a research theme thoroughly explored by the formal methods community. It can also be seen as a late software architecture evaluation technique, in which actual software architecture is compared to planned software architecture and observed deviations drive corrective measures [Tvedt et al. 2002]. Architecture conformance checking research is mainly focused on conformance of static module views. Module views are structural architecture diagrams in which modules are entities that aggregate code units that together implement a set of responsibilities, and module relations describe code dependencies between modules [Clements et al. 2002]. Conformance checking of module views, thus, compares entities and relations in a planned module view with the actual module view recovered from implementation, and the literature is prolific on this subject [Murphy et al. 1995; Aldrich et al. 2002; Koschke and Simon 2003; Postma 2003; Sangal et al. 2005; Knodel and Popescu 2007; Bourquin and Keller 2007; Huynh et al. 2008; Terra and Valente 2009; Feilkas et al. 2009].

Murphy and colleagues have tried to solve the issue of conformance between architecture module views and implementation with the adoption of the *reflexion model technique* [Murphy et al. 1995; Murphy et al. 2001]. Reflexion models turn the differences and similarities between an architecture module view and implementation into explicit information to the software developer. In short, the engineer defines a high-level model of interest, extracts the actual model from source code, defines a mapping between those models and computes a reflexion to see where both models agree or disagree. This work has opened up a new research path in software verification and prolific work on conformance checking of software architectures has since been published. Not only is the reflexion

model (RM) technique a seminal work on architecture conformance checking, but also a very popular technique. Practical applications of the RM technique in industry have been recorded in the literature. Applications range from reengineering [Murphy and Notkin 1997; Knodel et al. 2006], architecture recovery and conformance checking [Murphy et al. 1995; Murphy et al. 2001; Knodel et al. 2006; Knodel et al. 2008a; Rosik et al. 2008; Passos et al. 2010], architectural complexity analysis [Lilienthal 2009], and system understanding, redocumentation and software reuse [Knodel et al. 2006]. Furthermore, the RM technique has been extended to deal with additional issues such as hierarchical models [Koschke and Simon 2003], bottom-up architecture recovery [Le Gear et al. 2005], automated mappings [Christl et al. 2005; Christl et al. 2007], and software product lines [Frenzel et al. 2007].

Compared to other conformance checking solutions, the RM technique presents advantages for its *lightweight*, *approximate* and *scalable* nature [Murphy et al. 1995]. Simple box-and-arrows diagrams capture architectural rules, without either the need for complex architecture description languages or the use of ports, connectors and interfaces. These features make them particularly useful in lightweight development processes, where avoidance of complex formal specification languages and sparing of detailed diagrams in graphical modeling languages are deemed important.

The applications of conformance checking cited above show the benefits of introducing this step in the software process. Illustrated with academic examples and industrial case studies, those papers unveil various real-world scenarios where developers increase their conceptual integrity about the software systems they are developing after analyzing violations made explicit by conformance checks. I argue that conformance checking contributes to maintaining conceptual integrity, and to reducing the pace of architecture deterioration and, as a consequence, of software aging as well. I also argue that reflexion models are a conformance checking technique well-adapted to the context of lightweight development processes. These arguments will be discussed in detail along this work.

On the other hand, the original reflexion model technique presents some limitations when applied to a context of continuous software evolution in lightweight development processes. It requires the production of an initial architecture model, which can be challenging, since, in such processes, usually there is either no architecture diagrams or very few architectural

documentation is available. Another challenging task in the RM technique is the mapping between low-level code entities and high-level architectural modules, which usually demands domain knowledge or knowledge of coding conventions. Moreover, the mapping has to be kept up-to-date when software evolves and that imposes an extra burden on software developers. Automated mapping techniques based on structural dependencies exist, but are rather limited to capture the semantics of mapping. Finally, the RM technique can produce detailed lists of architectural violations, usually amounting to hundreds of violations, overloading software developers with information at each conformance check. Some of the identified architectural violations may never be solved because they may be exceptions to a general rule. Others may need to be solved in a short period of time to prevent erosion of important architectural features. Thus, identifying relevant architectural violations from violation lists is an issue in the technique that requires improvement. These limitations are mainly related to the manual effort spent by software developers when applying the technique. And this effort involves costs that can prevent adoption of frequent conformance checks.

2.2 A Process for Conformance Checking of Evolving Systems

To deal with the limitations above, I envisioned a conformance checking process to reduce the manual effort when applying the RM technique. To do so, the process tries to partially automate some of the steps to compute reflexion models. In the following, I present a description of this process, which I named *evolutionary reflexion model* (ERM) process, as it keeps some steps of the original RM technique, as well as its lightweight nature, but is adapted to take software evolution into account.

The ERM process is a process for the conformance checking of architecture module views. It is intended to be used in the context of lightweight development processes such as agile processes or open source development processes. In this context, there might not be an explicit architecture module view or rules that define the interaction of modules. Thus, before checking conformance, it is necessary to recover a module view and its associated rules. Evolutionary reflexion models are, then, produced as the result of a less frequent subprocess of architecture recovery and a more frequent subprocess of conformance checking,

as illustrated in Figure 2.3.

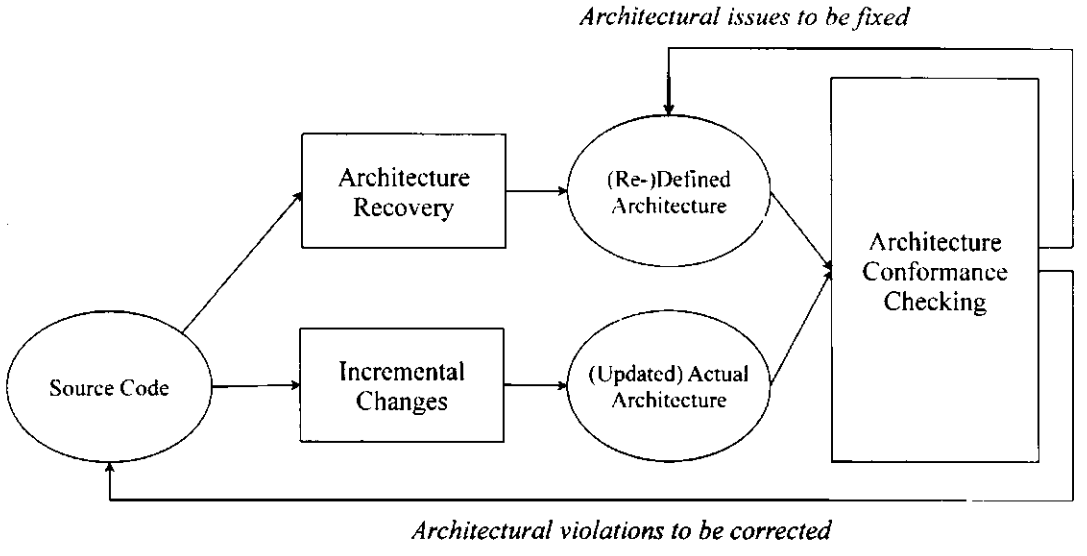


Figure 2.3: Evolutionary Reflexion Model (ERM) process

Architecture recovery is a thoroughly explored field of study. As such, it is not the intention of this dissertation to contribute on architecture recovery techniques, but, instead, to choose from a range of proven useful techniques. In the ERM process, architecture recovery is accomplished through a series of steps described below.

Design extraction: low-level design entities and relations are extracted from source code. Typical entities are methods, fields and classes, while typical relations are method calls, field accesses and class inheritances.

Design lifting: design entities and relations are lifted to the type level, where entities are classes or interfaces and relations are their dependencies. For example, each methods or field is lifted to its container class, while each method call or field access is lifted to its container dependency.

Design clustering: types are clustered into modules through automated software clustering techniques. Examples of actual modules are layers. For instance, an information system can be split into the graphical user interface layer, the business model layer, and the data persistence layer; a clustering technique is successful if it can recover such kinds of abstractions.

Semi-automated re-clustering: results from design clustering are improved through semi-automated re-clustering techniques. For instance, a graphical form class that has been incorrectly placed in the business model layer can be replaced into the graphical user interface layer by suggestions of an automated re-clustering technique.

Manual re-clustering: software developers change the resulting clustering to improve automated results and impose a particular point of view on the clustering. As an example, developers may wish to split a recovered abstraction of a graphical user interface layer into two sublayers: one based on direct manipulation and another based on web forms and servlets.

Definition of a module view and architectural rules: software developers name modules and establish relations between modules according to structural architectural rules. For example, developers may establish that the business model cannot depend on the graphical user interface.

The architecture recovery subprocess is illustrated in Figure 2.4.

After an architecture module view is defined through architecture recovery, the subprocess of conformance checking may be applied during the evolution of the software system. For instance, it may be applied before committing source code to the software repository. In this way, violations may be revealed before making changes more definitive. The steps of the conformance checking subprocess follow below.

Architecture changes: although not so frequent as source code changes, architecture can change as well. In this step, software developers may change the architecture module view and its associated rules. For example, a new module that performs network communication can be added to an existing information system, and relations may be established that only the business model layer can depend on the network communication module.

Design re-extraction and re-lifting: design is re-extracted to update entities and relations that might have changed both in low-level and in type-level designs. It follows the same rationale as the two first steps in the architecture recovery subprocess, but with an updated system.

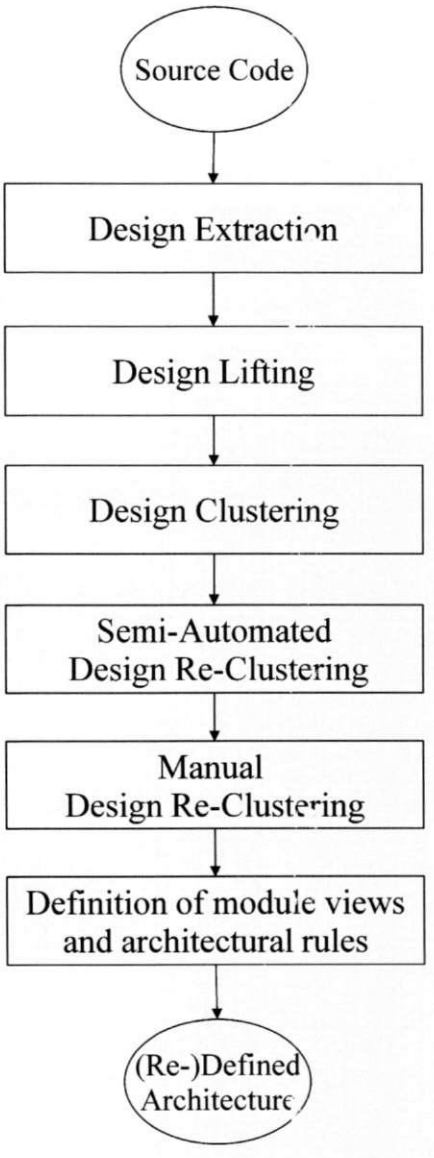


Figure 2.4: Architecture recovery subprocess

Semi-automated mapping: With the source code changed, mapping between type-level entities and modules may have to be updated. For instance, a newly added sockets class has to be mapped onto the network commmunication module, while a newly added business concept class has to be mapped onto the business model layer.

Manual (re-)mapping: semi-automated mapping may not be enough to cope with all changed entities and software developers might have to manually update the previous mapping. For instance, it may happen that a graphical user interface form class has various dependencies to business concepts and is incorrectly mapped onto the business model layer. The manual re-mapping solves these issues.

Checking and violation logging: the most important step in the process and the only compulsory one is accomplished by computing reflexion models. A reflexion model can show violations both as a graph and as a list of textual warnings. Results from checking can be improved from knowledge of software history, by either prioritization or filtering. For example, exceptions to a general rule may be filtered out from the violation list.

Violation resolution: software developers solve architecture violations by means of either changing source code or updating software architecture.

The conformance checking subprocess is illustrated in Figure 2.5.

It is worth mentioning how the ERM process affects software development. As a semi-formal lightweight technique, it imposes some extra burden to the developers. Nonetheless, it is still a lightweight burden, since the architecture recovery phase is less frequent, not all steps need to be accomplished each time the conformance checking is run and the process does not divert developers from their activities when they are coding. Furthermore, with the ERM process, developers do not need to learn formal notations to check conformance. Reflexion models can be generated and checked either before committing source code to the repository or during nightly/weekly builds, at the same time unit and integration tests are run.

Appendix A describes the steps of the ERM process in more detail. During this work, I developed a prototype toolset named *Design Suite* to facilitate the automation of the ERM process. The appendix also describes details about the toolset, and how it fits into the process.

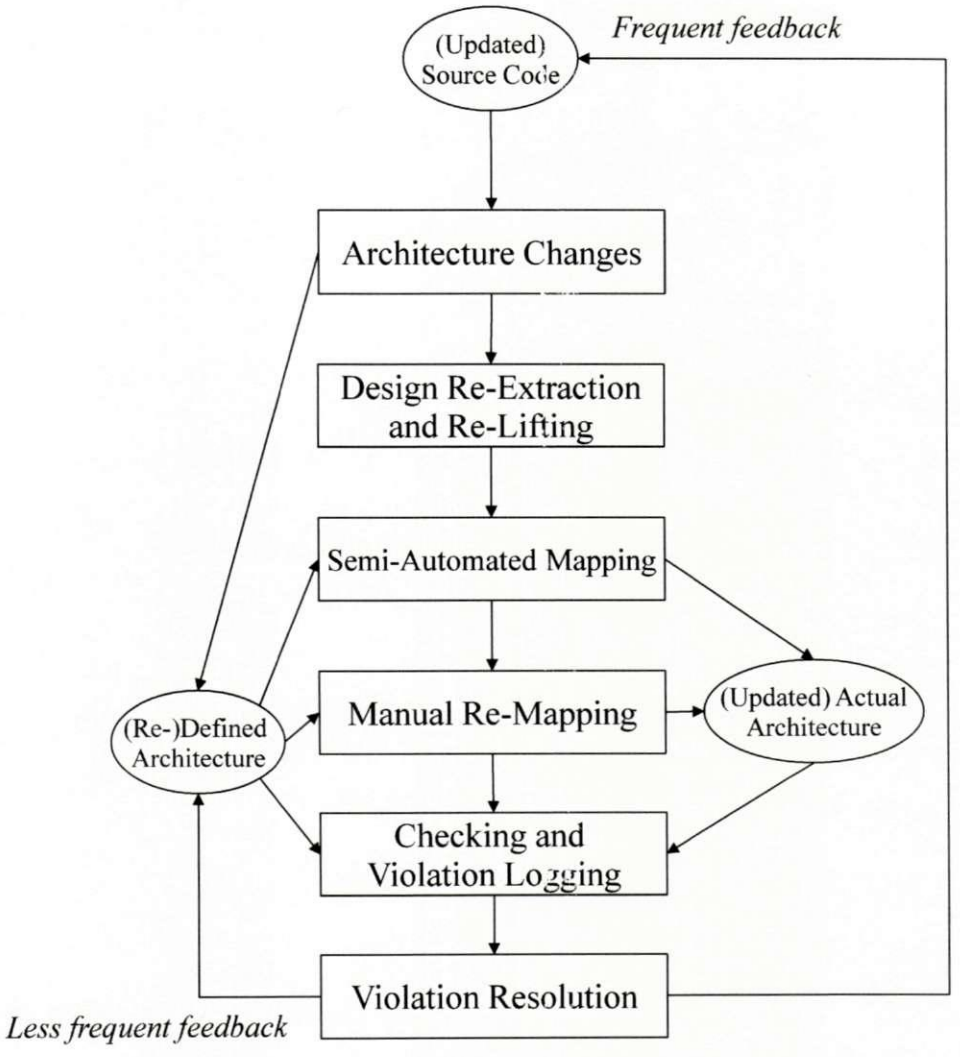


Figure 2.5: Conformance checking subprocess

2.3 Summary of the Dissertation

In this dissertation, I try to solve the limitations of the reflexion model technique when it is applied in the context of software evolution. More specifically, I propose techniques to reduce the amount of manual effort in specific steps of the technique: *i*) in the generation of high-level models; *ii*) in the mapping between source code and high-level models; and *iii*) in the analysis of results of conformance checks.

The results of this work are in the scope of a lightweight conformance checking process, based on the RM technique, that is adapted to software evolution. Evaluation of this process is complex, since it involves software developers, a set of software tools, and a longer time horizon. Hence, I focus instead on the three steps of the process explained above, those that would require a large amount of manual effort had the original RM technique been applied. Improvements achieved in this work for each step are independent of each other and can be used by tool developers and software engineering researchers one at a time. The contributions of this dissertation are, thus, in the detailed design and evaluation of techniques to support these steps, and are described below:

1. An empirical evaluation of clustering algorithms that produce high-level module views in the context of software evolution;

2. Design of an incremental mapping technique to map source code entities onto high-level modules based on a combination of information retrieval of software vocabulary and structural dependencies, and a quali-quantitative evaluation of incremental mapping techniques;

3. Investigation of factors that likely influence the relevance of static architectural violations found by conformance checks with the RM technique;

4. Design and evaluation of a recommender and a filter to reduce the overload of software developers when analyzing violations in the results of conformance checks;

5. Development of a prototype toolset to enable partial automation of the proposed conformance checking process.

2.4 Organization of this Work

This document is organized as follows. This chapter introduces the context, problem, and a conformance checking process as the solution to the problem. Chapter 3 reviews the state-of-the-art of architecture recovery and conformance checking of architecture module views. An outline of the dissertation is given in Chapter 4, with problem characterization, research questions, success criteria, hypotheses, research goals and methodology. A detailed study of three steps of the conformance checking process and their empirical evaluation is presented in the next three chapters: the production of high-level models from software clustering, in Chapter 5; the semi-automated mapping of implementation onto high-level models, in Chapter 6; and the prioritization of violations from architecture checks, in Chapter 7. A discussion on the results and its unfoldings follows in Chapter 8. Finally, conclusions are drawn in Chapter 9.

Chapter 3

Background

FUNDAMENTOS

Neste capítulo, alguns conceitos importantes para a compreensão da tese são apresentados ao leitor. O estado-da-arte em recuperação arquitetural e checagem de conformidade de visões arquiteturais modulares é revisado e uma avaliação crítica dos trabalhos relacionados é realizada.

In this chapter, some important concepts for this dissertation are introduced to the reader. The state-of-the-art of architecture recovery and conformance checking of architecture module views is reviewed and a critical appraisal of related work is performed.

3.1 Architecture Recovery

Architecture recovery is a subset of reverse engineering that tries to recover high-level architectural information through analysis of a software system [Hochstein and Lindvall 2005]. As a research theme, it has received considerable attention from the software engineering community, since the early 1990s. It can be used for a variety of purposes such as improving software comprehension, documenting legacy systems, serving as a starting point for reengineering processes, identifying components for reuse, migrating systems to software product lines, co-evolving architecture and implementation, checking compliance between architecture, low-level design and implementation, analyzing legacy systems and achieving graceful software evolution [Kazman et al. 2001;

Pollet et al. 2007].

A recent survey recovers prolific work in this area [Pollet et al. 2007]. An older study surveys the area of software architecture with focus on combating architectural degeneration, giving emphasis to architecture and design recovery [Hochstein and Lindvall 2005]. Before reviewing the area of architecture recovery, some basic information on software architecture is shortly presented in the following.

3.1.1 Software Architecture

Most definitions of software architecture take a structural perspective. They consider that the architecture is composed of elements and the connections among them. Some other aspects are also considered: configuration, constraints, properties, rationale, and requirements [Clements et al. 2002]. A popular definition states that "software architecture is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [Bass et al. 2003, p. 3]. In the context of software evolution, a simple and useful definition asserts that "software architecture is the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" [Garlan and Perry 1995, p. 269].

Architecture Documentation

Software architectures are documented for a variety of purposes. Documentation serves as a basis for system analysis, but also as a vehicle of communication among stakeholders, besides helping in system understanding. Different aspects need to be emphasized when designing an architecture. It would be hard to place all these aspects under the same representation. Thus, different views may be used to describe an architecture, depending on the analyzed viewpoint. A variety of views has been proposed by researchers and practitioners, depending on the aspect of concern. Clements et al. describe three viewtypes (module, component-and-connector and allocation), from which every other view may be derived [Clements et al. 2002]. Module views include elements that are code units implementing a set of responsibilities. Component-and-connector views have runtime entities as

components and their pathways of interaction as connectors. Allocation views map elements of either module or component-and-connector views onto the environment the software is in contact with.

Software architecture is usually expressed as different architecture views and design rules. A combination of module, component-and-connector and allocation viewtypes are able to express most architectural concerns in a set of architecture views in terms of software entities and relations [Clements et al. 2002]. Textual design rules complement these views with design decisions or constraints that may not fit well in graphical diagrams.

Module Views

A module is a software unit with a well-defined interface providing a set of services. A module can be any aggregation of software units (e.g.: a class, a collection of classes, a pluggable component, or a layer). Module views can describe a variety of styles, such as decomposition, dependencies and generalization, which give room to the relations *is-part-of*, *depends-on* and *is-a*, respectively. Further relations can be derived from these basic relations. Module views can be used as a blueprint for construction, as a basis for analysis and as a mechanism for communication. Most software architecture documentation describes at least one module view [Clements et al. 2002]. Finally, module views are also the most popular view addressed in research [Hofmeister et al. 2000].

Design Rules

Design rules can be related either to software structure or to the environment where software is immersed. Environmental design rules express design decisions that cannot be captured in architecture views. On the other hand, structural design rules are special design parameters that allow decoupling other parameters by means of stating global rules followed by the entire system or subsystem [Sullivan et al. 2001]. The most well-known design rule in software systems is a module interface. Structural design rules usually express acceptable and unacceptable dependencies between architecture modules [Sangal et al. 2005].

Structural design rules can be formally expressed in a variety of ways. Some examples are interfaces, ports and connectors of architecture description languages (ADLs) [Medvidovic and Taylor 2000], domain constraint languages [Terra and Valente 2009], cell marks

in a design structure matrix [Sangal et al. 2005] or arrows in a box-and-arrows diagram [Murphy et al. 1995].

3.1.2 Taxonomy

A large body of research on architecture recovery has been built for more than fifteen years. A recent work surveys 146 papers which are either considered influential or propose a specific approach to the architecture recovery problem [Pollet et al. 2007]. This survey classifies previous work in terms of inputs, outputs, techniques, processes and goals.

Inputs to architecture recovery are varied. Usually, input information is used to recover relations between software pieces. Entities from source code can be related using structural dependencies, source code vocabulary similarity, dynamic information from software traces, physical organization of software files or historical information from software repositories. Moreover, human organization, human expertise and documentation can also be used as inputs to architecture recovery.

Architecture recovery techniques may be classified into quasi-manual, semi-automatic or quasi-automatic [Pollet et al. 2007]. In the first, the software architect manually recovers abstractions assisted by a tool. The second automates repetitive tasks, instructing the tool to recover abstractions. And the last infers architectural knowledge directly from software artifacts.

In this work, I use another classification, that seems to be more common in the literature: manual, semi-automated and fully automated techniques. Manual techniques may or may not use software tools, and are driven solely by human decisions. In semi-automated techniques, the process is driven by humans, and either repetitive tasks are automated or architecture knowledge is inferred directly from software artifacts. In any case, final decisions are taken by humans, who use the generated knowledge as suggestions. Finally, fully automated techniques generate abstractions with no human intervention.

3.1.3 Architecture Module View Recovery Framework

Architecture recovery typically follows three steps: detailed low-level fact extraction from program static and dynamic analysis, abstraction of low-level facts to compose high-level

components, and architecture visualization of high-level views [Armstrong and Trudeau 1998]. When recovering a module view, source code entities and their dependencies are usually analyzed statically, although dynamic analysis and other types of information may be used as well to enrich fact extraction. Module view recovery benefits, for instance, from explicit dependencies between code entities in control- and data-flow graphs. Abstraction of architectural modules is generally done through clustering and filtering tools, although other techniques are available as well. Finally, a module view is generated through text and graphic visualization algorithms to produce a clear and concise high-level description of the system.

Fact Extraction

The first step to architecture module view recovery is the extraction of the structural and modular organization of the source code. The level of recovery is generally higher than the line-of-code level and depends on the used programming language. Entities that compose the source code and their dependency relations are identified and added to a low-level fact set. Low-level design entities depend on the type of programming language used. In a procedural language, typical entities are global variables, procedures and files, while typical relations are global variable uses, procedure calls and include dependencies. In an object-oriented language, examples of entities are methods, fields and classes, while method calls, field accesses, class inheritances, object parameter passing and interface implementations are examples of relations. This fact set may be organized as either graphs, relational databases or other data models [Armstrong and Trudeau 1998].

Fact extractors recover information from software artifacts such as source code, intermediate bytecodes or UML design diagrams and export their results in a variety of data formats. Holt et al. describe some of these tools and propose a standard for data exchange between software reengineering tools [Holt et al. 2006].

Abstraction

To identify architectural modules, one needs to abstract from the extracted facts. This abstraction usually consists of grouping a set of low-level entities into internally cohesive and externally relatively decoupled clusters. For instance, some procedures in a procedural sys-

tem may be grouped into an abstract data type; in a drawing object-oriented system, some classes that implement geometric figures may be grouped into a geometry module. Furthermore, non-relevant entities may be filtered out from the fact set to produce a clear and concise organization. For instance, uses of `java.util` library classes in a Java system may be ommitted from an architecture recovery process. Abstraction can be either assisted by the software architect or automated by techniques such as clustering, concept analysis and graph dominance [Wiggerts 1997; Pollet et al. 2007].

Visualization

Visualization is essential to architecture understanding. Architecture module views can be graphically described through a variety of visualization layouts. Visualization results must be clear and concise, and are usually displayed as views ordinarily used in architecture documentation, such as graph, UML or matrix views. Hierarchical visualization is desired, since module views may contain decomposition styles.

Research on information and software visualization provides insights for ideas on visualization layouts. Diehl surveys software visualization techniques and classifies them into methods that describe software structure, behavior or evolution [Diehl 2007].

3.1.4 Module View Recovery Techniques

Since module views decompose software into code units implementing a set of responsibilities, it is natural to depart from code units in order to recover architecture module views. Source code is usually the only available artifact in legacy systems, making this input especially relevant for architecture recovery. As a consequence, a large number of architecture recovery techniques is based on source code as input and ends up with some form of module view as output.

Manual and Semi-Automated Recovery Techniques

AT&T have provided some of the first tools to extract designs from source code. They use the framework *CIA/Ciao* to model concepts in programming languages and extract facts from source code, first in C [Chen et al. 1990] and later in C++ and Java [Korn et al. 1999]. And

they use the *GraphViz* tool to visualize graphs [Gansner and North 2000]. These tools have been used as input and output to various other design abstraction tools.

Later developments with the *Rigi* tool allowed to extract facts from source code into graphs that can be queried and edited by a graph editor [Müller et al. 1993]. Quasi-manual abstraction helps to filter, group and navigate through software entities. Visualization was later improved by *SHriMP*, which provides hierarchical fish-eye views [Storey and Muller 1995].

Still in the tool domain, University of Waterloo has produced a pipeline of tools for architecture recovery named *SWAG Kit* [Software Architecture Group 2009]. Source code can be extracted with tools like *cppx* and *javex*, abstraction is done through Tarski relational algebra and scripts with the *grok* tool and software landscapes can be viewed with *lsedit*.

Architecture recovery typically uses static information from the source code in order to recover high-level module views. Armstrong and Trudeau describe and compare three pioneering environments that extract static information to abstract it into high-level module views and show them graphically: *Rigi*, *PBS* and *Dali* [Armstrong and Trudeau 1998]. These tools are similar in their way of organizing facts into tuples, of abstracting facts using scripts that process these tuples in an assisted process, and of visualizing module views represented as box-and-arrow diagrams. These tools also allow manual changes to the views through interaction with the graphical views.

In a different approach, a top-down recovery process known is the *reflexion model* technique recovers the architecture through two steps. First, an initial description of a graph mental model is done by the architect. Then, a regular expression clusterer maps source code entities into this model [Murphy et al. 1995]. This process involves conformance checking as well, and is later detailed in Section 3.2.

Gupro is a tool that represents source code facts as graph files and allows sophisticated queries in a graph query language (*GreQL*) [Ebert et al. 2002]. This representation is especially well-suited to huge software systems. The initial file format has evolved to a standard exchange format for software reengineering known as *GXL* [Holt et al. 2000; Holt et al. 2006].

Bauhaus is a comprehensive research toolset for reverse engineering, with tools ranging from dead code finding, clone detection, architecture recovery and compliance, feature detec-

tion and metrics [Raza et al. 2006]. Facts are extracted from source code and dynamic traces into low- and high-level representations. Abstraction is done through a semi-automated interaction framework that combines twelve automatic approaches and user guidance. Visualization is done with the *Gravis* tool.

Automated Recovery Techniques

Automated techniques aim to recover high-level abstractions with information available in low-level models and very few or no intervention from a reverse engineer. Automated techniques are of special interest, since they promise faster recovery just from analyzing existing knowledge in software artifacts. They usually take advantage of other techniques like formal concept analysis, data clustering and graph dominance in order to discover abstractions. Since they use specific criteria or heuristics for recovery, these techniques end up imposing software architectures that respect such criteria or heuristics [Anquetil et al. 1999]. For instance, if the heuristic of a clustering algorithm demands high cohesion and low coupling, the recovery process ends up producing highly-cohesive and low-coupled decompositions.

Software clustering is a bottom-up architecture recovery technique that groups lower-level software entities that are similar in some way [Pollet et al. 2007]. This type of technique is further described next, in sub-section 3.1.5.

Concept analysis uses lattice theory to identify sensible groupings of objects that have common attributes. Siff and Reps use concept analysis to identify software modules [Siff and Reps 1997]. The process starts with building a context with code entities as the objects and their attributes derived from static analysis. Then, a concept lattice is built from the context. Finally, concept partitions, which are collections of concepts whose extents partition the set of objects, are identified and form the recovered modules.

Dominance analysis identifies related parts in an application, exploring the concept of graph dominance over a program graph extracted from source code by static analysis [Pollet et al. 2007]. From a directed program graph, dominance analysis tries to find the dominant nodes in it. A node D dominates a node N if all paths from a given root to N go through D. Identified nodes result in a set of candidate architectural entities. Dominance analysis produces good results to identify certain types of components, namely passive components, but not sufficient to recover the complete architecture [Lundberg and Löwe 2003]. A passive

component is a set of code entities having a single entity as interface, not using any other entity other than the dominant node to communicate to other components.

3.1.5 Software Clustering Techniques

Clustering techniques organize a collection of patterns (usually represented as a vector of measurements) into clusters, based on some kind of similarity [Jain et al. 1999].

Pattern similarity is measured either with distance measures or with association coefficients, and clustering algorithms use this similarity to produce clusters of related entities [Wiggerts 1997]. Wiggerts also classifies clustering algorithms as hierarchical, partitional, graph-theoretical and construction. Hierarchical clustering algorithms build a hierarchy of clusterings in either an agglomerative or a divisive fashion, and a cutoff in the resulting dendrogram produces a clustering. As an example, suppose that an object-oriented class is characterized by a feature vector extracted from its vocabulary in an information retrieval approach. All classes from a system may be compared using a vocabulary similarity measure and a similarity tree (dendrogram) can be built. Cutting the tree at a given cutoff point produces a clustering based on vocabulary similarity. Partitional algorithms, on the other hand, take an initial partitioning and iteratively improve it according to some heuristic. For instance, one may start with singleton clusters, and clusters may be joined iteratively, producing changes to an objective function to be maximized. Using heuristic search, one ends up with a suboptimal value of the objective function and a suboptimal clustering. Graph-theoretical algorithms represent software entities as a graph and try to find subgraphs, which will form the clusters, according to a graph-theoretical concept. For example, some graph-theoretical algorithms try to split connected subgraphs where they have the weakest connection, resulting in clusters of structural-related entities. Finally, construction algorithms assign the entities to clusters in one pass. For instance, entities can be placed in clusters according to their spatial position in an n-dimensional feature vector space.

In the context of software reverse engineering, patterns are derived from any relevant available information used as input to architecture recovery such as structural dependencies, source code vocabulary similarity, dynamic information from software traces, physical organization of software files or historical information from software repositories. Software clustering techniques usually process input information and group low-level software entities

into clusters that represent high-level modules [Wiggerts 1997].

Anquetil and colleagues use agglomerative hierarchical algorithms to automatically cluster software, experimenting with different parameters such as how the entities are described, how the entities are coupled and which algorithm is used [Anquetil et al. 1999]. Maqbool and Babri use hierarchical clustering algorithms to groups entities based on different similarity measures [Maqbool and Babri 2004]. They also propose a weighted combined linkage algorithm to measure the distance between clusters and aggregate the most similar ones in an agglomerative fashion.

Mancoridis and colleagues have built a software clustering tool named *Bunch* [Mancoridis et al. 1998; Mancoridis et al. 1999; Mitchell and Mancoridis 2006]. This tool organizes source-level modules and dependencies in a graph and uses optimization algorithms such as hill climbing or genetic algorithms to find clusters based on an optimization function. This function works as a measure of modularization quality, rewarding partitions of clustered modules with high internal cohesion and low external coupling.

A *design structure matrix* (DSM) is a matrix representation of system entities and their interactions. It can be used as a means to better organize design activities and layouts, as an efficient design visualization technique and as a basis for system abstraction and mathematical analysis. DSMs have also been used for system clustering, initially in the manufacturing industry [Browning 2001]. Gutierrez-Fernandez represents design modules as a design structure matrix (DSM) and proposes an optimization algorithm to partition the design into clusters in order to minimize the coordination cost between teams that work on the modules [Gutierrez Fernandez 1998]. Later work improves the clustering algorithm [Thebeau 2001]. Sangal et al. apply DSMs to analyze software through the use of the *Lattix LDM* tool [Sangal et al. 2005]. Using matrix algorithms, they convert the DSM into a block triangular matrix, where blocks represent cohesive clusters.

3.1.6 Evaluation of Software Clustering Techniques

The work of Armstrong and Trudeau seems to be the first to evaluate architectural extractors, but focus is on quasi-manual recovery tools [Armstrong and Trudeau 1998]. The lack of benchmarks makes it difficult to compare different architecture recovery techniques in general, and software clustering algorithms in particular. Sim et al. propose benchmarks as

a challenge to improve software engineering research [Sim et al. 2003].

Koschke and Eisenbarth propose a framework for experimental evaluation of clustering techniques, which also includes a benchmark for four C procedural systems [Koschke and Eisenbarth 2000]. Anquetil et al. study three aspects of the clustering activity: abstract entity description, metrics that compute coupling between entities and clustering algorithms [Anquetil et al. 1999]. CRAFT is a framework for evaluating software clustering results in the absence of benchmark decompositions [Mitchell and Mancoridis 2001a]. CRAFT builds a reference decomposition automatically based on common patterns produced by various clustering algorithms.

To compare decompositions, measures have been proposed to quantify the difference between two software partitions. Tzerpos and Holt propose the $MoJo$ distance measure, that computes the number of entity moves and joins to transform one partition into another [Tzerpos and Holt 1999]. Later, the same authors discuss the notion of stability of software clustering algorithms, using $MoJo$ to define a stability measure [Tzerpos and Holt 2000]. Mitchell and Mancoridis propose two measures that consider both vertices and edges of a partitioned graph when computing the distance between two partitions: the $EdgeSim$ measure, that takes into account the intra- and inter-edges common in both partitions, and the $MeCl$ measure, based on cluster splitting and merging operations to transform one partition into another [Mitchell and Mancoridis 2001b]. In their work, they evaluate four measures: precision/recall, $MoJo$, $EdgeSim$ and $MeCl$.

Finally, the work of Wu et al. is, to our knowledge, the first to extensively compare different algorithms for software clustering based on three different dimensions relevant to software architecture. They have derived three criteria to evaluate aggregations of software units: authoritativeness, stability and extremity [Wu et al. 2005]. According to them, a good aggregation should resemble one made by an authority who knows well the system. Furthermore, small incremental changes should not produce too different clusterings, i.e., it should be stable. And finally, clusters should be non-extreme to be meaningful and useful, i. e., neither huge nor tiny.

3.1.7 Re-Clustering during Software Evolution

Most software clustering techniques focus on finding a clustering out of no previous arrangement. However, very few research deals with the issue of updating a clustering when software evolves [Tzerpos 1997]. Software changes produce the following possible scenarios concerning clustering: 1) A new entity is introduced in the system and needs to be clustered; 2) An entity is removed from the system and has to be removed from clustering; 3) An entity changes its dependencies to other entities in the system;

The issue in scenario 1 is named the *orphan adoption problem*, also known as *incremental clustering* [Tzerpos 1997]. Tzerpos proposes a solution to this case, by providing criteria and an algorithm to adopt an orphan. Criteria involves naming conventions, structural dependencies and programming styles. Naming conventions are captured by regular expressions, while structural dependencies are used to determine the module that depends most on the orphan, which adopts it. Programming styles capture specific issues such as the difference between feature and utility modules.

While scenario 2 simply leads to removing the entity from its parent cluster, scenario 3 may sometimes lead to increased communication between software modules that were previously low-coupled. Increasing coupling may sometimes suggest the re-clustering of an entity that is causing stronger coupling. This problem is named *maverick analysis* and solutions to it try to identify entities that appear to be in the wrong module [Schwanke 1991]. Such scenario also arises when poor quality clusterings are analyzed. Schwanke defines a maverick as an entity whose majority of its k nearest neighbors belongs to a module different than it does. Neighboring is computed from pattern similarity, as previously defined in subsection 3.1.5. Tzerpos considers a maverick as a special type of orphan. In fact, they name it a *kidnappee*, as if it had been "kidnapped" from its parent module. The issue is solved by making the appropriate parent module readopt the *kidnappee* [Tzerpos 1997]. Kidnapping uses the same criteria as orphan adoption, but it is avoided when the *kidnappee* belongs to a module with less than three entities or to an utility module, or when the original module interface increases with kidnappee removal.

Later work has produced a solution similar to orphan adoption to map newly added entities to the mapping of code entities onto modules in reflexion models [Christl et al. 2005]. In this work, automated clustering techniques are used to create additional candidate mappings

based on a previous mapping. Attraction functions based on structural dependencies are used to suggest the best candidate module for a given orphan.

Finally, it is worth mentioning that both automated clustering and re-clustering techniques may not produce the modules that the rationale of the software architect would. Thus, manual re-clustering is usually a necessary step after using automated architecture recovery.

3.2 Conformance Checking

The process of checking conformance between software architecture and implementation has been thoroughly studied by the scientific community. A recent work informally describes this process as a sequence of six steps [Lindvall and Muthig 2008]: 1) capturing and modeling the planned architecture; 2) extracting the actual architecture from source code; 3) mapping components in the actual architecture onto those in the planned architecture; 4) automatically comparing actual and planned architecture based on the previous mapping; 5) analyzing each deviation to determine whether it is critical; 6) defining a plan to remove violations deemed critical.

Two surveys extensively review the problem of architectural erosion and their possible solutions, including a detailed description of architecture conformance checking techniques [Hochstein and Lindvall 2005; de Silva and Balasubramaniam 2012]. Hochstein and Lindvall focus on tools and techniques to combat architectural degeneration, and also discuss the feasibility of its prevention [Hochstein and Lindvall 2005]. De Silva and Balasubramaniam, on the other hand, classify a range of techniques and approaches to either minimize, prevent or repair architectural erosion [de Silva and Balasubramaniam 2012]. They evaluate each category in their classification, also discussing their adoption in industry, their efficacy, and performing a preliminary cost-benefit analysis.

This section intends to establish a background on architecture conformance, starting from its concepts, passing by conformance of architecture module views, techniques for architecture conformance checking, reflexion models, applications of conformance checking, low-level design versus architecture conformance, architecture conformance checking during software evolution, and concluding with warning prioritization in conformance checking tools.

3.2.1 Definitions

The concept of architecture conformance checking is not yet a consensus in the scientific community. Various terms are used to name similar things: *compliance*, *conformance*, *conformity* and *consistency* sometimes have the same meaning, at times they slightly diverge. Moreover, the terms *checking* and *verification* may represent either similar concepts or strongly different ones, according to each academic sub-community that refers to them. To some in the formal methods community, checking is considered as a semi-formal method, while verification is understood as a formal method with proofs of correctness of an implementation respecting a specification. I shall use these terms indistinctly, although different definitions may exist in the literature.

The Open Group is a consortium of organizations that aims to reach open standards and global interoperability in software development. They define a framework for software architecture named TOGAF (The Open Group Architecture Framework). In this framework, a specific chapter on architecture conformance defines the following levels of architecture conformance: irrelevant, consistent, compliant, conformant, fully conformant and non-conformant [The Open Group 2008].

Figure 3.1 depicts these levels, showing to what extent architecture specification and implementation agree. When architectural specification and implementation have no features in common, the issue of conformance is *irrelevant*. Architecture and implementation are *consistent* when they share some features and such features are implemented according to the specification, although there might be non-implemented specified features as well as implemented features not covered by specification. Implementation is *compliant* to the architectural specification when all implemented features are covered and in accordance with specification, even though there might be non-implemented specified features. On the other hand, implementation is *conformant* to the architectural specification when all specified features are implemented, despite the existence of some additional implemented features not covered by specification, i.e., implementation is sound regarding architecture. The strongest level of conformance is when specification and implementation are *fully conformant*, i.e., there is full correspondence between specification and implementation. A fully conformant implementation is both sound and complete regarding the architecture. Finally, the *non-conformant* level happens when, in any of the previous scenarios, some features are implemented not in

accordance with architectural specification.

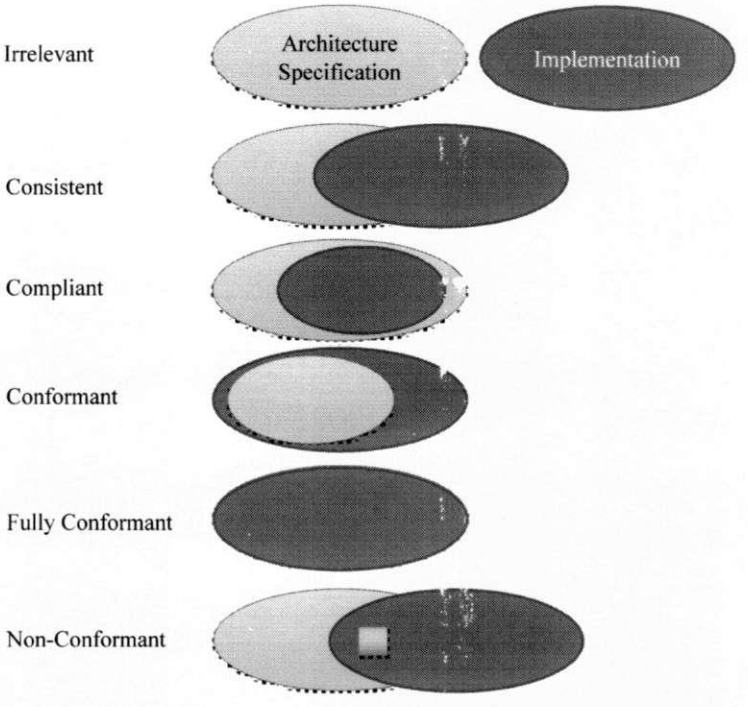


Figure 3.1: Architecture conformance levels

The concept of conformance that is used throughout this work is based on the *conformant* level, i.e., all features specified in the architecture are implemented according to the specification, although additional features not covered by specification may be implemented. Soundness, and not completeness of specification, thus, is the main issue for conformance as I will refer in this work.

In a different way, Knodel and Popescu define architecture compliance as a measure to which degree an architecture implemented in source code conforms to the planned architecture [Knodel and Popescu 2007]. Hence, 100% compliance means that there are no architecture violations, while 0% compliance means that all imposed architecture restrictions are violated. Compared to figure 3.1, violations to architecture compliance move an implementation from the conformant to the non-conformant level.

3.2.2 Conformance of Module Views

The architecture module view is available in most architectural frameworks and is usually present in most architecture documentation [Clements et al. 2002], while also being the most popular view addressed in research [Hofmeister et al. 2000]. Furthermore, it is also important as a design-time representation to communicate software structure between software developers. Given its relevance, the module view is a strong candidate to aid in the maintenance of conceptual integrity inside the development team. For such reasons, I focus this work on the conformance of architecture module views.

The reader may refer to sub-section 3.1.1 for additional background on architecture module views.

3.2.3 The Reflexion Model (RM) Technique

The original paper on reflexion models is a seminal work on architecture conformance checking [Murphy et al. 1995; Murphy et al. 2001]. It provides the original definition of an architecture checking process, the one used to define the term *conformance checking* in the start of this section. It also defines the concepts of *convergences*, *divergences* and *absences* for structural module views (named high-level models in the original work), as detailed next. The reflexion model (RM) technique is also a top-down approach to architecture recovery. The process to compute reflexion models is shown in Figure 3.2. Starting from a high-level model proposed by the software engineer according to his or her comprehension of the system, a source-level model is extracted from source code and their entities are mapped onto high-level model entities. From this mapping, low-level relations are lifted to high-level implemented relations, which can be compared against high-level planned relations for conformance goals. From this conformance checking, it is possible to either refine the high-level model or perform reengineering activities to improve conformance between the high-level model and implementation.

In the context of architecture conformance of module views, the simplest static conformance check compares the existence of an implemented module against a planned module. Three cases may arise:

a) *Convergence*: a module exists both in planned architecture and in implementation;

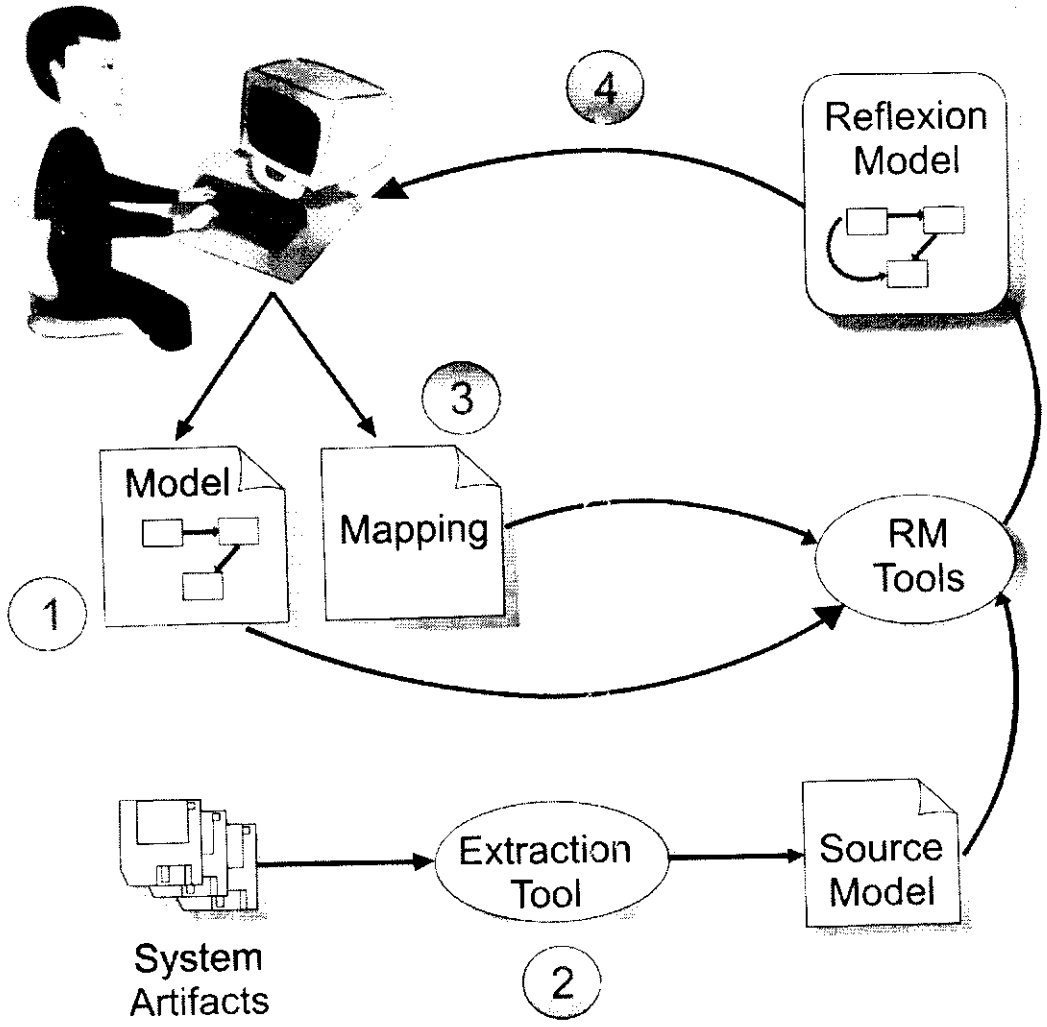


Figure 3.2: Process to compute reflexion models [Murphy 1996]

b) *Divergence*: a non-planned module exists in implementation;

c) *Absence*: a planned module does not exist in implementation.

Although module checking is important, most existing work on static architecture checking, including the RM technique, focuses on the conformance of *relations* between modules, since imposed architectural constraints are typically expressed as relations between modules. Three cases may arise during conformance checking of relations between each two planned modules and their equivalent implemented modules:

a) *Convergence*: a relation between two modules is implemented as planned, showing that implementation conforms to the planned architecture;

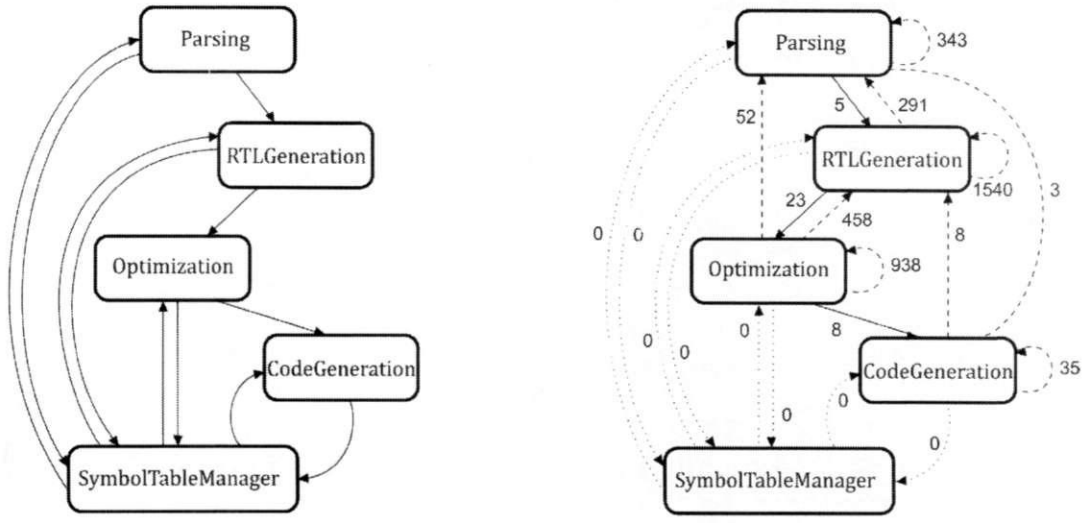
b) *Divergence*: a relation between two modules is not allowed but it is nonetheless implemented, showing non-conformance to the planned architecture;

c) *Absence*: a relation between two modules is planned but it is not implemented, showing non-conformance due to an absence in the implementation.

Figure 3.3 shows an example of reflexion model of the *gcc* compiler. On the left, the high level model shows compiler modules and expected dependencies. On the right, the reflexion model shows convergences, divergences and absences. Convergences are shown as solid arcs; divergences, as dashed arcs; and absences, as dotted arcs. The numbers for each arrow account for the number of source code relations associated to the high-level conformance relation.

The three main features of the RM technique are its *lightweight*, *approximate* and *scalable* nature. The technique is lightweight because it requires low effort from the software engineer to apply the technique. It is approximate, because the model can be iteratively refined, starting from a coarse mapping. And it is scalable because it can be applied either to small systems or to million lines-of-code systems [Murphy et al. 2001]. Its simplicity, not requiring to formally define module interfaces, ports or architecture connectors, make it adequate to lightweight development processes.

One of the most interesting case studies of reflexion models is the reengineering of 1.2 million lines-of-code Microsoft Excel [Murphy and Notkin 1997]. The engineer designs an initial high-level model with 13 modules and 19 relations. After an iterative process of



(a) High-Level Model **(b) Reflexion Model**

Figure 3.3: Convergences, divergences and absences in reflexion models [Murphy 1996]

computing reflexion models, he ends up with a high-level model with 16 modules and 114 relations, a source-level model with 131,042 relations and a map with 1,425 entries. It takes him one day to design and extract the initial models and four weeks are spent to iteratively refine the reflexion model. The engineer reports that the technique helps in: *i*) refining a system's architectural view; *ii*) investigating the connection between architecture and code; *iii*) reasoning about the reengineering activity; *iv*) automating parts of the reengineering activity; *v*) better understanding the code base; and *vi*) avoiding reasoning in terms of a high-level model alone. Additional case studies with reflexion models are performed to help with design conformance of three systems: a layered architectural design of a program restructuring tool, the SPIN operating system and an industrial C++ subsystem [Murphy et al. 2001].

Reflexion models may, thus, be used in a variety of software engineering activities: in architecture recovery with a top-down approach, in reengineering tasks as it has been done with Excel, in architecture conformance activities and in system understanding, among other activities.

3.2.4 Extensions to the RM Technique

Koschke and Simon extend Murphy's RM technique to account for hierarchical module views, motivated by practitioners' complaints that the lack of hierarchies was a shortcoming for the practical use of the technique [Koschke and Simon 2003]. In a hierarchical model, one needs to redefine reflexion relations and mappings, given the existence of container entities in the level of planned architecture. Since lower-level relations are more specific than higher-level ones, one can define that the former override the latter. A reasonable interpretation of this definition is that lower-level rules are exceptions to higher-level rules. With container entities in the planned architecture, one also needs to define "lifted" relations: a container entity A references another entity B if at least one of A parts references B or at least one of B parts. In other words, reference relations are lifted from the part to the whole. With the existence of container entities and the definition of lifted relations, one can redefine, for hierarchical reflexion models, the checking relations *convergences*, *divergences* and *absences*.

Another limitation of the RM technique is that it demands the software engineer to know the mapping between source code entities and planned architectural modules. Even though one may count, in some cases, with conventions based on directory hierarchies and file names, the mapping process still demands great manual effort. To our knowledge, the first attempt of performing automated mapping in the RM technique uses set-based flow algorithms to provide a basis for refining an initial partial high-level model and an initial partial mapping [Zhang et al. 2004]. Improvements to the mapping step in the RM technique are proposed in a semi-automated mapping technique that uses automated clustering techniques [Christl et al. 2005; Christl et al. 2007]. The rationale behind the technique is that semi-automated mapping does not replace manual mapping, but, instead, it assists the engineer to reach a correct mapping faster. With the use of structural dependencies, they define a similarity function (attraction function) that determines the closeness of low-level concrete entities to high-level planned modules. With the attraction value given for every pair of concrete entities and planned modules, mapping can be performed either fully automatically or by the software engineer using a rank of attraction values from an entity to the candidate modules. In this dissertation, I use information retrieval techniques to improve mapping accuracy.

An extension of the RM technique is proposed to consolidate software variants into prod-

uct lines [Frenzel et al. 2007]. In this case, three levels of models are used: the implemented model, the single product architecture and the product line architecture. The process starts with computing a reflexion model for a single product. Then, mapping between already analyzed products and an additional product is aided by clone detection techniques that find their commonalities and variabilities. This mapping process is repeated for other additional products. Mappings and module views are completed for each product. Then, a similar process is done to map each variant onto the product line architecture. Mappings are validated and completed for each product with variabilities accommodated in the product line module view.

In the RM technique, to produce the initial high-level module view of a system with an unknown architecture, the software engineer may review artifacts, interview experts or look at similar architectures [Murphy and Notkin 1997]. Together with the mapping of source entities onto modules, deriving the initial high-level model is one of the most challenging steps of the RM technique. Using software reverse engineering techniques is a natural choice to aid in this derivation. This is what is proposed in the *Reconn-exion* method [Le Gear et al. 2005], which is a combination of software reconnaissance technique with reflexion models. Software reconnaissance is a feature detection technique based on dynamic analysis. Reconn-exion uses its results to find the reuse perspective of a system, which is the set of software elements shared between two or more distinct features, taken for all the features of interest in a feature set. The authors assert that the reuse perspective is part of the core modules of a system architecture and, as such, might be a good start to begin the process of defining the high-level planned module view of a reflexion model. After that, the original RM technique is iteratively applied until a refined reflexion model is found.

3.2.5 Alternatives to Reflexion Models

The most traditional alternative to the RM technique for architecture conformance checking is the use of architecture description languages (ADLs). ADLs allow specifying architectural models by means of a formal language, where an architectural model is an artifact that captures some or all of the design decisions that make up an architecture [Medvidovic et al. 2007]. Various ADLs have been proposed since software architecture has become a major concern in software development processes [Medvidovic and Taylor 2000;

Medvidovic et al. 2007]. In their latest survey on ADLs, Medvidovic et al. assert that one trouble with most ADLs is that, although they allow characteristics of architecture descriptions be verified using analysis tools, there is usually no way to ensure conformance between architecture and implementation. One solution to the issue of lack of conformance of ADL descriptions and implementation is embedding the ADL in the programming language, which is the approach taken in ArchJava [Aldrich et al. 2002]. ArchJava is an extension of the Java language aimed at providing integration between architecture and implementation. It adds to Java the concepts of component, port and connector. Components are instances of special component classes. Ports represent a logical channel of communication between components and declare provided, required and broadcast interfaces. Connectors join two components by means of their compatible ports. The language tries to enforce communication integrity, a consistency property that asserts that direct communication between implemented components shall only happen when the respective architectural components are also connected.

Instead of using a full ADL for architecture description and conformance, one can use domain-specific constraint languages suited to the needs of architecture conformance. Inspired by other logical constraint languages and the RM technique, Terra and Valente propose a dependency constraint language, called DCL, to provide architecture conformance by construction, using a static and declarative constraint language [Terra and Valente 2009]. The language specifies modules as sets of classes, and relations between modules with constraints like `only A can-* B`, `A cannot-* B`, and `A must-* B`, where `*` shall be replaced by the appropriate source-level relation (e.g.: `access`, `extend`, `create` and so on). In addition to DCL, they have developed the *dclcheck* prototype tool, which interprets declared modules and constraints and checks whether the implementation conforms to the constraints.

In another track, Sullivan et al. first apply the notion of *design structure matrix* (DSM) to the study of modularity of software designs. They represent software entities as the matrix design parameters (represented both as rows and columns in a square matrix) and relations between entities as matrix dependencies expressed as marks in matrix cells [Sullivan et al. 2001]. Figure 3.4 shows an example of DSM, where entity B depends on entity A, and both B and C depend on each other.

	A	B	C
A	.		
B	X	.	X
C		X	.

Figure 3.4: A design structure matrix (DSM)

DSMs have been later applied to manage the architecture of software systems using dependencies in the DSM with the *Lattix Dependency Manager* (*LDM*) tool [Sangal et al. 2005]. In that work, structural architecture views are expressed as hierarchical DSMs, based on package or directory structure decompositions, as can be seen in figure 3.5. Design rules are defined as allowed or not allowed dependencies between design parameters and come in two forms: `A can-use B` or `A cannot-use B`. Extracted dependencies from source code form a dependency matrix that is checked against the design rules in the architecture view. Design rule violations are shown in the DSM as small triangles in the matrix cells where violations arise.

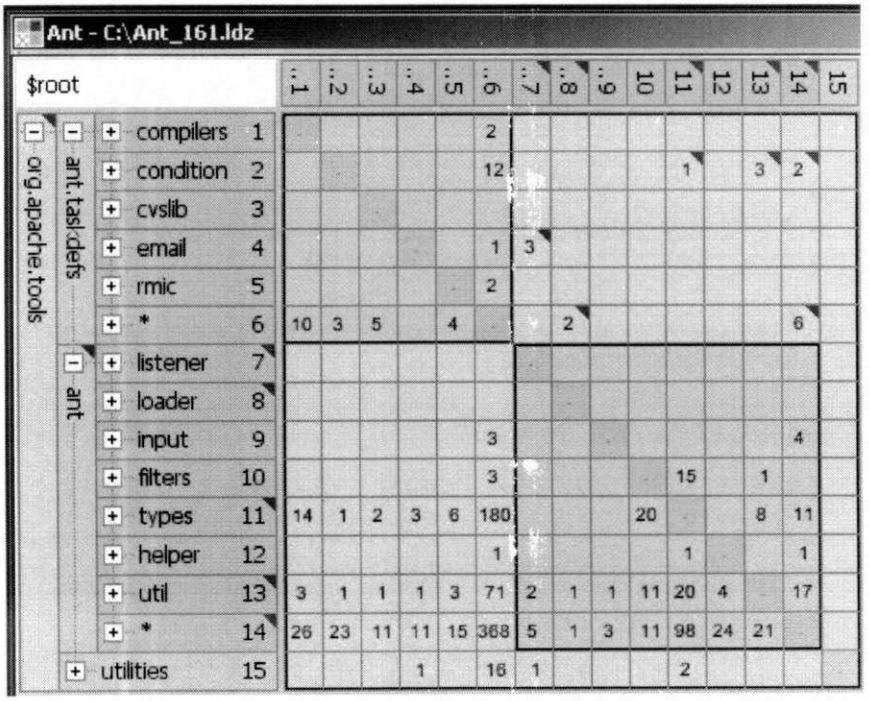


Figure 3.5: Lattix LDM tool [Lattix, Inc. 2008]

The tool above, *LDM*, is limited by the existence of package or directory structure decompositions that reflect the conceptual view. Either this may not always be possible or

there can be more than one structural view of interest. Mapping the source-level DSM onto an arbitrary architecture-level DSM is an issue that another work tries to solve [Huynh et al. 2008]. It uses a genetic-algorithm-based clustering tool named *Glusta* to map, in an automated fashion, a source DSM onto an architecture DSM using various heuristics. In *Glusta*, both DSMs are represented as directed graphs and the genetic algorithm fitness function finds solutions that minimize the edit distance between the architecture graph and a clustered version of the source graph. The fitness function also penalizes empty mappings between both graphs, and rewards mappings based on directory groupings and name patterns in source code. This work, thus, tries to solve a problem similar to the previously described work of Christl et al. [Christl et al. 2005]. Besides that, it models design rules as a formal logical constraint network named *augmented constrain network* (ACN), which is then converted into dependencies in the architecture DSM.

Conformance between an specified module architecture (SMA) and an implicit module architecture (IMA) derived from source code is the aim of another work [Postma 2003]. In it, software architects and system documentation help derive and formalize a specified architecture view and architecture rules, and produce a mapping between code and architecture elements. Formalization of rules is done by means of a relation partition algebra (RPA) [van Ommering et al. 2001]. The IMA is extracted from source code, abstracted by means of the previous mapping and verified against architecture rules. Architecture-level violations are concretized, and both architecture- and code-level violations are presented in a visualization tool. The work is similar to Murphy's [Murphy et al. 1995], although verifying architecture rules by means of RPA is more generic than verifying specific relations in reflexion models, since architecture rules may express conditions on multiple relations at once.

Component access rules are another architecture checking approach, inspired by ADL ports [Medvidovic and Taylor 2000] and OSGi exported packages [OSGi 2008]. They allow specifying simple ports for components, which other components may access [Knodel and Popescu 2007]. They help to increase component information hiding to a level not supported by current implementation languages. Different from most previous checking approaches, where rules specify allowed or forbidden relations between modules, a component access rule concerns only one component.

Comparative studies between different conformance checking techniques have also been

pursued [Knodel and Popescu 2007; Passos et al. 2010]. Knodel and Popescu compare three different techniques, namely, reflexion models, component access rules, and relation conformance rules expressed in RPA [Knodel and Popescu 2007]. By means of a case study, they qualitatively compare the techniques in thirteen different dimensions. Then they recommend a goal-driven selection approach based on the these dimensions to decide which technique to apply in a given scenario. Passos and colleagues, on the other hand, compare design structure matrices, source code query languages and reflexion models [Passos et al. 2010]. A source code query language adopts an SQL-like syntax that includes features aimed at improving expressiveness of queries over source code. They also offer an illustrative overview of the techniques with a working example, and synthesize the lessons learned by comparing the techniques in terms of four dimensions: expressiveness, abstraction level, ease of application and architecture reasoning and discovery.

3.2.6 Applications of Conformance Checking

Architecture conformance checking has been applied with different goals, as reported in the literature. Beyond the case studies reported by the authors of the RM technique [Murphy et al. 1995; Murphy and Notkin 1997; Murphy et al. 2001], previously discussed in subsection 3.2.3, additional applications are described below.

Lilienthal describes 24 case studies of architecture evaluation of industrial object-oriented systems developed in Java, with size ranging from 25 KLOC to 14 MLOC [Lilienthal 2009]. Three aspects of architectural complexity are analyzed: modularity, ordering and pattern conformity. The last aspect is analyzed by statically checking conformance of the implemented architecture against architectural styles and the intended architecture, using a commercial tool named *Sotograph*.

Knodel et al. perform nine case studies (5 industrial and 4 academic cases) to show how architecture conformance checking contributes to further development and evolution of architectures [Knodel et al. 2006]. The static architecture evaluations performed in the case studies with the help of the *SAVE* tool aim at ten different purposes: assessing the potential of building a product line from existing products, determining the alignment of a product to a product line architecture, assessing the reuse potential of a component or architectural fragment, measuring the internal quality of a component, achieving better pro-

gram comprehension by comparing a mental model of a software against its implementation, assessing the consistency of architectural documentation with its implementation, detecting non-documented architecture elements to improve completeness of documentation, re-documenting a software system or product line, monitoring the evolution of a software system, and assessing the structural decomposition of a software system or product line. The authors assert that "static evaluations of software architectures are a sound instrument to control, learn and assess architectural aspects and implementations of architectures" [Knodel et al. 2006, p. 293].

Another work describes the successful experience of transferring architecture conformance checking technology from the Fraunhofer IESE to industry [Knodel et al. 2008a]. With the use of the *SAVE* conformance checking tool, fifteen different instances from a product line are regularly checked at a partner company (Testo AG) over a two-year period. Starting with offline conformance checks offered as a service by Fraunhofer IESE, Testo architects later independently check architecture conformance of their products with the SAVE tool. At the end of the period, conformance checks are integrated into their regular development process.

A long-term qualitative case study over a two-year period in the context of forward engineering of a single software component is performed with the use of an adapted version of reflexion models [Rosik et al. 2008] to a context of forward engineering. The product is an IBM commercial product named Domino Application Portlet (DAP). The existing product is reengineered in a two-year period with regular architecture conformance checks every 4-5 months. The main findings reveals that the approach is successful in identifying architectural violations previously unnoticed by developers. However, several other violations are hidden by the technique and are only found by thoroughly examining convergences in the reflexion models. They believe that this evidence should be used to refine the approach and integrate it into the software development process.

In a study about the loss of architectural knowledge during software evolution, three industrial projects have their evolution analyzed to quantify: *i*) the degree of conformance between architecture documentation and source code; *ii*) the discrepancies between intended and documented architecture; and *iii*) the amount of architectural decay [Feilkas et al. 2009]. They also investigate the causes of nonconformance. To answer these questions, they use a

technique similar to the reflexion model technique. Their results show that between 70 and 90% of nonconformances are caused by documentation flaws and between 10 and 30% of them are architectural violations in the source code. They mention one main lesson learned: the intended architecture has to be made explicit, and conformance has to be continuously checked to minimize the loss of architectural knowledge.

Finally, architecture violations from conformance checks may also drive high-impact architectural refactorings [Bourquin and Keller 2007]. A case study on a mid-sized telecommunications software system uses architecture violations derived from reflexion models to identify high-impact refactoring opportunities. Finding "bad smells" in parts of the software with most violations helps choose the refactorings with more likely higher impact. This technique does not replace refactoring techniques. Instead, it complements them by focusing developer work on architectural issues.

3.2.7 Low-level Design Conformance Checking

Although it is not the intention of this dissertation to provide conformance checking between detailed design and implementation, work on this topic has trends similar to architecture conformance checking, and it also provides insights about solutions to the higher-level problem.

Fiutem and Antoniol check static conformance between class diagrams in the OMT notation and implementation in C++ object-oriented programs [Fiutem and Antoniol 1998; Antoniol et al. 1999]. Both design and implementation are converted into abstract object language (AOL) descriptions, which are parsed into abstract syntax trees (ASTs). Entities are matched in the ASTs using structural properties and name comparisons based on an edit distance algorithm and in regular expressions. Results are visualized in a graph format, using different colors for convergences, divergences and absences. Different from architecture conformance checking, there is no need to map low-level entities onto modules, since extracted code and design entities are at the same level.

Design Wizard is a tool and an API that allows expressing and checking design rules by means of design tests in a unit testing framework [Brunet et al. 2009]. *Design tests* differ from functional tests in that they do not test what a software should do, but, instead, how the software was built. Design tests are automated tests expressed in the same language as the implementation and check design rules expressed as an algorithm. *Design Wizard* is instan-

tiated for the Java language and the tests are written as automated *JUnit* tests. From some case studies, authors conclude that the API is easy to learn and that designers and programmers appreciate the use of design tests as executable design documentation. Performance is evaluated and results show that the technique scales well for large software systems [Brunet et al. 2011].

Pires et al. propose a model-driven architecture (MDA) approach to automatically generate design tests to check conformance between UML class diagrams and Java programs [Pires et al. 2008]. From an UML class diagram, design tests are generated for the *Design Wizard* API by means of a model-driven transformation, based on MDA models and metamodels. Tests are generated to check conformance of class, field and method signatures, and relations of association and generalization. Test generation is fully automated, as well as test execution.

3.2.8 Conformance Checking during Software Evolution

Maintaining software architecture and implementation conformant with the use of the previous techniques is a challenging task, especially in the context of software evolution. Both documentation and implementation evolve and add another dimension to the problem of conformance checking. The use of software configuration management (SCM) tools may help to deal with the time dimension. However, these tools have not been designed to work with conformance between software artifacts. Some solutions adapt conformance checking techniques to the context of software evolution, making use of SCM tools. Other solutions use online tools to provide conformance by construction.

ArchEvol provides a scenario with sequential steps to perform conformance checking of evolving software [Nistor et al. 2005]. Through language extensions, plug-ins and guidelines to manage the relationship between architecture and implementation, they provide a solution adapted to software evolution. In *ArchEvol*, architecture diagrams are written in xADL 2.0 with *ArchStudio* and implementation is written in Java with *Eclipse*. With *Subversion*, a popular SCM tool, both architecture diagrams and implementation files are stored as versions in a software repository. A scenario of architectural-driven development and evolution with sequential steps is devised to provide interactions between tools to keep artifacts conformant. A complementary approach based on the *ArchTrace* tool works to keep traceability links

between architecture and implementation up-to-date [Murta et al. 2006]. *ArchTrace* implements a set of policies to keep traceability links between xADL 2.0 architecture diagrams and source code stored in Subversion.

Lighthouse is a tool that focuses on low-level emerging design coordination, but also works as a conformance checking tool [van der Westhuizen et al. 2006; da Silva et al. 2006]. It helps to coordinate developers on the same project, to detect design decay and to manage the status of the software project. An emerging class diagram is generated from source code, but it is extended with evolution actions, developer ownership of changes and visual hints of conformance between conceptual design and implementation. Conformance is shown in *Lighthouse* as different colors in the diagram.

With the advances in modern IDEs, conformance checking may be applied during software development, providing conformance by construction. *LogEn*, a logic domain-specific language based on *Datalog* has been proposed, together with a visual notation for comprehensive specification of architectural dependencies (*VisEn*), and an Eclipse plug-in has been developed to incrementally check architectural properties in source code developed in Java. The plug-in shows violations to architectural rules in the developer's screen. The language enables various levels of granularity, from intra-class dependencies to architectural building blocks [Eichberg et al. 2008]. One disadvantage is that it requires learning a logic domain-specific language (*LogEn*) to express design rules.

Finally, an experiment to demonstrate support to constructive conformance checking has been conducted with the use of the *SAVE* tool [Knodel et al. 2008b]. The *Eclipse* client-server plug-in named *SAVE LiFe* was developed to extend *SAVE* with the ability to automatically check conformance during software development. The architect client allows to define the structural view and the mapping between high- and source-level entities, while the developer client triggers automatic fact extraction and conformance checking on the server side. Violations to architecture are presented to developers as highlighted source code statements and a tabular list of violations. The experiment performed with university students shows that the experimental group produces 60% less violations than the control group.

The solution proposed in this dissertation is meant to be used together with regular software quality assurance (SQA) activities, such code reviews or testing, while constructive conformance checking produces immediate results from online tools. Despite being good

for revealing violations upfront, such an immediate focus on violations may limit designs to emerge from coding, which seems inadequate for lightweight development processes. It seems better to delay the focus on architectural violations to the SQA phase, where software engineers work on a different mode, looking for inconsistencies.

3.2.9 Prioritizing Warnings in Checking Tools

Conformance checking tools can produce large lists of violation warnings, usually amounting to hundreds of violations [Terra and Valente 2009; Feilkas et al. 2009]. Such lists point where, in the source code, the architectural models a e violated. In this context, focusing the developers' attention to the most relevant warnings is an appropriate way to avoid overloading developers with tool results. Thus, prioritizing warnings from conformance checking tools seems to be an important research topic that has not been explored.

Instead, existing work on prioritizing warnings has been directed at bug finding tools as they usually produce long lists of bug warnings with a large amount of false positives [Kremenek and Engler 2003]. Most of this work is devoted to predicting the warnings most likely to be real bugs, usually by means of assigning priorities to warnings based on software analysis [Kremenek and Engler 2003; Williams and Hollingsworth 2005; Boogerd and Moonen 2006; Kim and Ernst 2007b; Ruthruff et al. 2008].

Kremenek and Engler use the co-location of bug warnings (i.e., warnings clustered in the same code region) and developer feedback about failures to distinguish true bugs from false positives [Kremenek and Engler 2003]. Williams and Hollingsworth mine data from source code repositories to improve static analysis results for a single bug pattern of return value checking [Williams and Hollingsworth 2005]. Boogerd and Moonen compute the execution likelihood of a warning location to assign the priority of a bug warning [Boogerd and Moonen 2006]. Kim and Ernst have developed a machine learning approach to assign priorities to bug warnings based on software history [Kim and Ernst 2007b]. They change the default priorities of warning categories based on the warning-fix history recorded in the software repository. They also show that warning duration plays an important role in determining warning priority [Kim and Ernst 2007a]. Finally, Ruthruff et al. perform a logistic regression analysis to predict the relevance of warnings from bug finding tools [Ruthruff et al. 2008]. They use fix information from the bug tracking system and a list of 33 factors of potential

impact to bug relevance: from both tool warning descriptors, in-house warning descriptors, file features, warning history in source code, source code factors, and churn factors. They end up with a model to predict actionable defects from bug warnings from a reduced list of 15 factors.

Chapter 4

Outline of the Dissertation

LINHAS GERAIS DA TESE

Neste capítulo, o problema tratado nesta tese é melhor caracterizado, com uma descrição das limitações da técnica dos modelos de reflexão. Questões de pesquisa são então levantadas e critérios de sucesso para resolvê-las são detalhados. Em seguida, as hipóteses centrais e os objetivos de pesquisa desta tese são estabelecidos. Finalmente, a metodologia usada para executar este trabalho é apresentada.

In this chapter, the problem treated in this dissertation is better characterized, with a description of limitations of the reflexion model technique. Research questions are then raised and success criteria to solve them are detailed. Then, fundamental hypotheses and the research goals of the dissertation are stated, followed by the methodology used to execute this work.

4.1 Problem Characterization

As previously stated in Chapter 3, the reflexion model (RM) technique can be used in various software engineering activities such as architecture recovery, reengineering, program comprehension and conformance checking because of its lightweight, approximate and scalable features. Nevertheless, the original technique poses some challenges for developers applying it.

Two challenges developers face in providing inputs to the RM technique are the produc-

tion of a high-level model, and the mapping between code entities and high-level modules. That is because they require knowledge both in the software domain and in software decomposition techniques [Koschke and Simon 2003]. Regardless of the increased design quality that can be achieved with the RM technique, there are costs associated to the time spent on producing models and mappings. And, in the context of software evolution, both high-level models and mappings must be kept up-to-date, which adds extra costs to the process.

A challenging issue for a software developer with respect to the output of the RM technique is the large number of source code violations that can be reported by the technique, which may overload software developers with excessive information.

4.1.1 Input to the RM Technique

Composing software into its architectural modules is an issue for architecture recovery techniques, which have been discussed in detail in Section 3.1. The RM technique helps recover modules using previous knowledge from a software engineer. One interesting question is whether this process may be improved with the aid of (semi-)automated architecture recovery techniques. As software evolves, high-level models used in the RM technique must also be evolved. Most existing architecture recovery techniques [Pollet et al. 2007] that can be used to produce a high-level model have not been evaluated in the context of evolution, especially as to whether these recovered models remain both accurate and stable.

One of the limitations of the RM technique lies in the mapping between code entities and high-level modules. Either the process of mapping is completely manual or it is based on directory structures and naming conventions captured by regular expressions, which are not able to fully capture all software modular properties. This issue is worsened by software evolution, where source code changes demand updating the mapping. Mapping, thus, can slow down and add errors to the conformance checking process. Partial automation of the mapping process is a possible solution that has been pursued [Christl et al. 2005; Christl et al. 2007]. Nonetheless, the solution proposed was based only on the use of *structural dependencies* to map new entities onto existing modules, resulting in low inter-module coupling. However, structure alone can not fully capture desirable module properties such as cohesion. Cohesion has various dimensions such as logical, temporal, procedural, communicational, sequential and functional, and any of these may be wished by software architects.

Software structure cannot capture all these dimensions, since each of them may lead to different modular mappings. Thus, additional information other than structural dependencies should be investigated if one wishes to automate the mapping step in reflexion models.

4.1.2 Output from the RM Technique

The RM technique generates both graphical and textual results as output. While the former give an overview of architectural violations, the latter provide information of where violations arise in the source code, usually as detailed violation lists. Such lists usually amount to hundreds of source code violations in each conformance check, which can overwhelm software developers with excessive information. When the list of violations is long, it can be difficult for a developer to find the violations that really matter, since the degree of relevance of these violations to a developer varies. Some architectural rules may be stricter then others, some rules may be more important to a team member than to others. In addition, the fact that architecture changes as software evolves can also change the relevance of each violation.

4.2 Research Questions

4.2.1 Main Research Question

The main research question that this dissertation investigates is:

- *How can we reduce the manual effort to apply the reflexion model technique in the context of evolving, sparsely documented software?* A guiding principle in investigating this central question is keeping the conformance checking provided by the reflexion model technique lightweight and scalable.

4.2.2 Complementary Research Questions

In particular, three questions were investigated as part of the research conducted into the central question.

1. How can semi-automated architecture recovery techniques enable the generation of high-level models in the context of software evolution and how can we evaluate the

techniques in terms of accuracy and stability?

2. What additional information from source code can be used to improve the automated mapping provided by techniques based on structural dependencies and how much improvement this information can bring?

3. How can the outputs of the reflexion technique be prioritized in order to reduce information overload to software developers?

4.3 Hypotheses

From the problem characterization and research questions stated above, I derive the following hypotheses to solve the research questions.

- Software clustering techniques can enable the semi-automated generation of high-level static models for the reflexion technique. In addition, it is possible to identify one or more clustering techniques most adapted to the context of software evolution, generating both accurate and stable models.

- Information from source code vocabulary can help to partially automate the mapping process, turning the RM technique more lightweight in the context of software evolution. Both structural dependencies and the vocabulary of source code can be used to detect likely mappings between newly added entities and high-level modules, and a combination of them can increase accuracy in a semi-automated mapping step.

- It is possible to correlate the relevance of violations discovered by reflexion models with a combination of measures extracted from the history of software artifacts. Moreover, such correlation based on past history can be used to prioritize violations in present architecture checks, reducing the overload on software developers when analyzing the outputs of the reflexion technique.

4.4 Success Criteria

To objectively evaluate the hypotheses raised in this work, we need success criteria against which these hypotheses can be tested. Criteria can be either quantitative or qualitative, as long as it is possible to objectively measure them by empirical observation. The reduction of manual effort to apply the reflexion model technique, which is the main research question that drives the hypotheses, is indirectly measured through specific success criteria for each of the hypotheses.

In this work, I evaluate software clustering techniques to recover software architecture views in terms of accuracy and stability. Building on the work of Wu and colleagues [Wu et al. 2005], I use their evaluation measures, two of them for accuracy and one for stability:

- *authoritativeness*: the degree to which a software partition found by clustering resembles a partition suggested by an authority;
- *non-extremity*: the ratio of software entities clustered in non-extreme (neither tiny nor huge) modules over the total of software entities;
- *stability*: the degree of similarity of two software partitions found by clustering two different software versions.

Establishing absolute values for the measures above to state success is complex, since they depend on a time series analysis of the measures over the evolving software system. In Chapter 5, I describe a relative measure called HML that aggregates data points of a time series of one measure. When more than 80% of the data points fall above a high-level threshold, I classify the technique as having a high value for the measure. Objectively, success of a clustering technique is clear when the relative HML measures of both authoritativeness, non-extremity and stability have a high value. Other than that, success is relative. It is also important to compare the techniques against each other, and, in the same chapter, I describe another relative measure called $Above$ that compares two or more data series. Success of a clustering technique against the other competing clustering techniques is given by the values of $Above$ for authoritativeness, non-extremity and stability.

Regarding the issue of automated or semi-automated mapping in reflexion models, two important measures that allow to objectively evaluate the quality of the mapping functions

are:

- *precision*: metric that evidences the soundness of a mapping approach, measuring the ratio of correctly found mappings over the total of mappings found by the approach;
- *recall*: metric that evidences the completeness of a mapping approach, measuring the ratio of correctly found mappings over the total of correct mappings.

Stating success for a mapping technique in terms of absolute values of precision and recall is also complex, since there are no previous baselines to compare with. Precision has to be at least 50% to be better than a random mapping, and this is a bottom baseline. Recall, on the other hand, has no bottom baseline, but their values cannot be very low (e.g., below 50%), or the technique would be useless to reduce manual effort. To account for both, I combine these measures in *F-measure*, the harmonic mean of precision and recall. Thus, a minimum value of *F-measure* to state success against a bottom baseline would be any value above 50%. As a rule of thumb, I establish an *F-measure* value between 60 and 75% to state medium success, and a value between 75 and 100% to state high success. On the other hand, given that a mapping technique is above the bottom baseline, it seems more important to compare it against other competing mapping techniques. Thus, a relative comparison between *F-measure* values for the evaluated mapping techniques helps to state their relative success against each other.

Finally, to evaluate the prioritizing of outputs in the reflexion technique, relevant violations must be distinguished from irrelevant violations, and the technique must prioritize the relevant ones. Three complementary information retrieval measures can be used:

- *specificity*: the ratio between violations correctly classified as irrelevant and the number of irrelevant violations found by the technique;
- *sensitivity*: the ratio between violations correctly classified as relevant and the number of relevant violations in the population;
- *precision*: the ratio between violations correctly classified as relevant and the number of violations found relevant by the technique.

There are two scenarios for the analysis of prioritizing violation, as later detailed in Chapter 7: the first, a top-K violation ranker, and the second, a violation filter. In the violation

ranker, precision is the only important measure, and success is stated from a comparison of precision values against a baseline of a random selection of K violations. If there is a relative improvement in precision by using the ranker, the technique is successful. On the other hand, filtering violations has a more complex success analysis. Discarding truly irrelevant violations is the main goal of a screening procedure with a violation filter. Thus, specificity is the most important measure. Nonetheless, the filter has to be sensitive and precise as well, although not as much as it is required to be specific, otherwise it would be useless to reduce manual effort. For success analysis, I establish a rule of thumb of 75% as the minimum accepted specificity, and 50% as the minimum values for both sensitivity and precision, since there are no known baselines for violation filters.

4.5 Research Goals

4.5.1 General Goal

The main goal of this work is to investigate and improve a conformance checking process between architecture module views and implementation adapted to lightweight development processes and to a context of software evolution.

4.5.2 Specific Goals

With the main goal in mind, specific goals are derived in order to reach the success criteria established in section 4.4. These goals are described below:

- Developing a set of tools to facilitate and partially automate the designed process of conformance checking of module views;
- Identifying a software clustering technique to enable the semi-automated generation of accurate and stable high-level models from evolving source code;
- Improving figures of recall and precision in the semi-automated incremental mapping of newly added software entities based on the use of source code vocabulary as an alternative or as a complement to the existing mapping techniques based on structural dependencies;

- Investigating how a set of factors extracted from the history of software artifacts correlates with the relevance of architectural violations identified in conformance checks;
- Prioritizing the results of architecture conformance checks by means of a recommender system based on software history;
- Enabling the filtering of conformance checking results to focus developer's attention to an accurate set of relevant architectural violations.

4.6 Methodology

This work intends to investigate and improve a conformance checking process to be continuously applied during software evolution in lightweight development processes. Since it is a process, the scientific methodology used is based on an analytical approach, in which individual steps of the process are analyzed and improvements considered. Individual steps lead to software tools and techniques that can be independently developed and evaluated. Hence, this approach led to the following methodology.

1. Literature review was performed on the fields of architecture recovery and conformance checking to identify the state-of-the-art on the dissertation theme. Here, related work was reviewed and open research questions were identified.
2. A conformance checking process based on an existing conformance checking technique was envisioned. The improved process was named the *evolutionary reflexion model* (ERM) process and it is presented in Section 2.2. From open research questions, hypotheses were stated about the ERM process. From these hypotheses, the process was then detailed (see Appendix A).
3. Investigation was accomplished by analyzing individual steps of the ERM process. The most challenging steps led to specific research questions, which were investigated by the design and construction of tools, and later evaluation. The evaluation of the the proposed solutions to the research questions was performed by means of empirical research:

(a) The step of design clustering was implemented in the *Design Abstractor* tool, whose main purpose was to cluster low-level design entities into high-level modules. Different clustering techniques implemented in the tool were evaluated with a quantitative case study [Bittencourt and Guerrero 2009]. This work is discussed in detail in Chapter 5.

(b) Visualization of recovered architecture module views was implemented on the *Design Viewer* tool. The tool was evaluated with a qualitative proof of concept [Bittencourt et al. 2009].

(c) Partial automation of the step of mapping from low-level code entities onto high-level modules was implemented in the *Design Mapper* tool, using mapping techniques based on structural dependencies and information retrieval. Evaluation of the tool was performed against criteria of precision and recall by means of two quantitative case studies [Bittencourt et al. 2010]. This work is thoroughly described in Chapter 6.

(d) Prioritizing and filtering results from the checking and logging step were implemented in the *Design Miner* tool. Relevant factors were identified by means of correlation with violation relevance, which was possible by analyzing against the history of existing projects. Results generated by the filter and recommender were evaluated against criteria of precision, specificity and sensitivity in retrospective quantitative experiments [Bittencourt et al. 2012]. This work is presented in detail in Chapter 7.

4. Previous work, proposed solutions, empirical results, a critical discussion and conclusions were synthesized in this dissertation.

Chapter 5

Empirical Studies of Clustering Algorithms for Architecture Module View Recovery

Estudos Experimentais de Algoritmos de Agrupamento para Recuperação de Visões Arquiteturais Modulares

Neste capítulo, é realizada uma avaliação experimental de quatro algoritmos de agrupamento aplicados na recuperação de visões arquiteturais modulares de software no contexto de evolução de software.

In this chapter, an empirical evaluation of four clustering algorithms to recover software architecture module views in the context of software evolution is performed.

5.1 Introduction

Support for checking conformance on an evolutionary setting requires up-to-date high-level models. Producing and updating such models requires manual effort by the software developers, but this work can be eased using reverse engineering techniques. Automated reverse engineering techniques based on software clustering have been

proposed as a means to facilitate the generation of static high-level models [Wiggerts 1997]. Although research has been prolific in finding algorithms for automated clustering [Pollet et al. 2007], little information concerning their empirical evaluation is presently available in the literature, especially in the context of software evolution.

In this chapter, I evaluate four clustering algorithms for the production of architecture module views. The first algorithm is a traditional algorithm used to cluster patterns: K-means clustering (km). The second is an algorithm used in the domain of social networks to cluster graphs based on their edge betweenness (eb). Finally, heuristics specific to software engineering are also used to find good software clusterings, and two algorithms that use such heuristics are evaluated: modularization quality clustering (mq) and design structure matrix clustering (dsm).

Wu et al. have previously derived three criteria to evaluate software aggregations: *authoritativeness*, *stability* and *non-extremity* [Wu et al. 2005]. A good aggregation should resemble one made by an authority who knows well the system. Furthermore, small incremental changes should not produce too different clusterings, i.e., it should be stable. And finally, clusters should be non-extreme to be meaningful and useful, i. e., neither huge nor tiny. These criteria were used to evaluate the clustering algorithms in this work.

By means of a case study, I analyze consecutive released versions of four open source systems. Results show that no algorithm performs best for all criteria. Furthermore, relative measures also point to poor results: medium values of non-extremity for km, and low values for the other three algorithms; low values of authoritativeness for eb, and medium values for the remaining three; and low stability for all algorithms, but eb, that was highly stable. These results show a limitation of these clustering algorithms to fully automate the production of high-level models.

5.2 Clustering Algorithms

All four clustering algorithms are described in detail below.

5.2.1 K-means Clustering

K-means is a popular algorithm in the pattern recognition community that clusters entities into a specified number of clusters, based on their proximity in a d-dimensional space. It takes as input M entities, represented as d-dimensional feature vectors, and a number K of clusters, and classifies each entity into the cluster with the nearest centroid, in an iterative fashion [Hartigan and Wong 1979] Entity feature vectors can be either binary vectors of structural dependencies or real vectors with information extracted from source code vocabulary, using an information retrieval technique [Manning et al. 2008]. In the latter case, vectors are compared against cluster centroids using Euclidean distance, while in the former case, comparison is made using the Jaccard distance [Anquetil et al. 1999].

In this work, a software entity to be clustered with K-means is a Java object-oriented type, which can be either a class or an interface. Dependency relations between types are extracted through static analysis, and are lifted from source code dependencies, as explained in Appendix A. Each type is characterized by a binary feature vector. There is one dimension in the feature vector for each existing type, and each component has value 1 if the given type depends on the type referred by that dimension, and 0 if it is not related to that type. Similarity between vectors is computed with the Jaccard distance, which is higher if both vectors share more dependencies with existing types. With this setup, K-means ends up producing clusters of structurally-related types.

5.2.2 Edge Betweenness Clustering

This algorithm clusters graphs based on edge betweenness. The betweenness of an edge is the extent to which that edge lies along shortest paths between all pairs of nodes. Edges which are least central to clusters are progressively removed until the clusters are separated. This algorithm is based on the notion of community structure, rather common in complex networks [Girvan and Newman 2002]. Communities, in the context of software design, are the modules that join structurally related design entities.

In this work, with object-oriented types as entities (e.g., Java classes or interfaces), a system graph is produced with the existing types as the graph nodes, and with existing dependencies between types as the graph edges (see Appendix A. Thus, a type tends to be connected to other types that share structural dependencies with it. With this setup, one expects edge betweenness clustering to produce structure-based clusterings.

5.2.3 Modularization Quality Clustering

Clustering algorithms based on optimization start from an initial partition and improve it according to some heuristic [Wiggerts 1997]. Modularization quality clustering finds clusters through optimization of a function that maximizes structural cohesion and minimizes structural coupling [Mancoridis et al. 1998; Mitchell and Mancoridis 2006]. Starting from an initial random partition, neighboring partitions are explored in order to find the one that reduces the optimization function the most. Using hill climbing or genetic algorithms, one ends up with a sub-optimal solution to the clustering problem. Figure 5.1 shows an example of partitioning of a software system represented as a program graph: the partition on the left shows better modularization quality than the partition on the right.

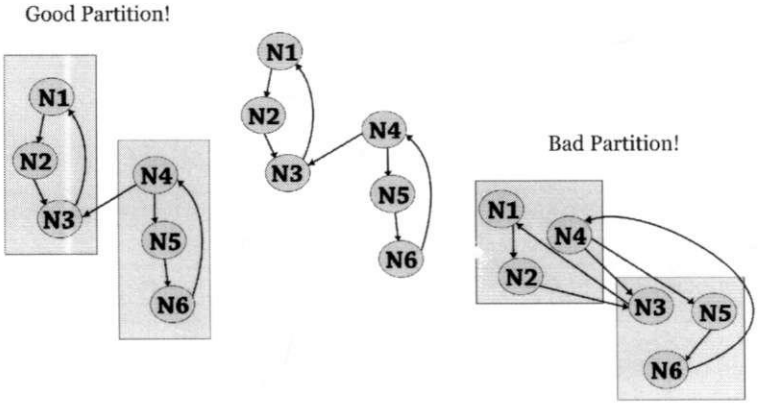


Figure 5.1: Modularization quality clustering [Doval et al. 1999]

Similar to edge betweenness, a system graph is made up of types as nodes and type dependencies as edges. Using the optimization function, one expects the initial random partition to change to neighboring partitions that are structurally more cohesive and

less coupled, and to converge to a partition that is, in terms of structure, strongly cohesive and weakly coupled.

5.2.4 Design Structure Matrix Clustering

Design structure matrix clustering is another optimization algorithm that can be used for design clustering. It uses an optimization function that minimizes the coordination cost between design entities. Design entities are represented in a square matrix, with dependencies between entities filling the matrix cells. The optimization function takes into account the internal cost of coordination between entities inside a module and the external cost of coordination between modules. Starting from singleton clusters, entities are joined in larger clusters through bids that minimize the coordination cost [Gutierrez Fernandez 1998; Thebeau 2001].

A system graph such as the one used in the two previous algorithms can also be represented as a square design structure matrix. With this representation, there will be one row and one column for each existing type in the system. If type A on the third row depends on type B on the fifth column, a value different from zero fills the matrix cell intersecting row 3 and column 5. The cell value depends on the strength of the type dependency between A and B. The design structure matrix algorithm starts with singleton partitions and joins types into the same module when this minimizes the coordination cost, which penalizes communication between modules. With this setup, one expects strongly cohesive and weakly coupled clusterings, in terms of structure, similar to the modularization quality clustering algorithm.

5.3 Evaluation Criteria

Before defining evaluation criteria, some concepts must be discussed. Architecture recovery can be achieved through clustering of lower-level software entities, which are usually source files, classes or any other design-level entities.

A software clustering may be formally defined as the partitioning of a set of design-level entities. Although it is possible to deal with software decompositions that are

not partitions (e.g.: an entity that belongs to more than one cluster or an entity that belongs to no cluster), most automatic clustering algorithms allocate each entity into one and only one cluster. Similarity between partitions expresses how close they are to each other. The lesser the number of operations required to transform one partition into another, the more similar they are.

Tzerpos and Holt have defined a measure of dissimilarity called $MoJo$ [Tzerpos and Holt 1999]. Given two partitions A and B, $MoJo(A, B)$ is the number of entity *moves* plus the number of cluster *joins* needed to transform A into B. To measure similarity, one derived relative quality measure could be:

$$MoJoSim(A, B) = 1 - \frac{MoJo(A, B)}{n} \tag{5.1}$$

where n is the number of entities to be clustered. In this work, I used an existing algorithm to compute $MoJo(A, B)$ [Wen and Tzerpos 2003].

Figure 5.2 shows an example of the $MoJo$ measure, computed between partitions A and B. Since there was the move of type $t4$ from cluster $M1$ in partition A onto cluster $M2$ in partition B, and the clusters $M1$ and $M3$ in partition A were joined in partition B, $MoJo(A, B) = 2$. Computing the associated similarity measure for the example, $MoJoSim(A, B) = 0.8$.

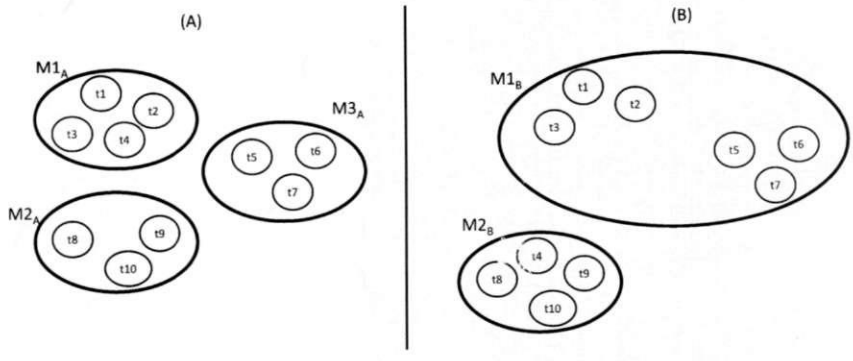


Figure 5.2: a) Original Partition A; b) Partition B, with 1 move and 1 join from A.

5.3.1 Authoritativeness

Probably the most important measure of the utility of a software clustering algorithm is how close its resulting partition resembles one created by an expert. This expert authority is usually an experienced software engineer or architect who produces a module view of the system, usually a view of high-level components and its dependencies.

Nevertheless, it is often hard to find an authoritative partition of a software system due to lack of documentation. Furthermore, even if this partition is available, it may not accurately reflect the software architecture as the system evolves. A partial solution to this issue that was adopted in this work is using the development view (from an allocation viewtype) as the authoritative partition. For instance, joining all classes that belong to the same package in a cluster usually aggregates architecturally-related entities. While it may not accurately express the module view, it is well known that both views share a considerable amount of architectural decisions.

To compare a partition P generated by a clustering algorithm to an authoritative partition PA, $MoJoSim(P, PA)$ was measured. The closer it is to 1, the better the algorithm is considered from an authoritative perspective.

5.3.2 Stability

Another important criterion to compare clustering algorithms is stability. Informally, small incremental changes in the system should not produce too different clusters. Good algorithms should be stable enough to produce similar clusters when small changes happen, but still produce different clusters when architectural changes happen. Clearly, this balance makes it hard to measure the quality of an algorithm regarding stability because one has to be able to distinguish minor changes from architectural ones. Nonetheless, avoiding high stability as well as high instability is, in fact, a relevant quality of a clustering algorithm.

Algorithm stability was measured through the similarity between two partitions P_i and P_{i+1} generated by the same algorithm in two consecutive versions of a system. Measuring $MoJoSim(P_i, P_{i+1})$ allows finding the stability of an algorithm. To make it

fair, this measure has to discard both added and removed entities during software evolution, and the comparison is made only between entities existing in both versions. Moreover, a better sense of stability should be noticeable in the behavior of this measure along the software evolution axis.

5.3.3 Non-Extremity of Cluster Distribution

Another desirable property of an architectural clustering is that a cluster should resemble architectural components. Usually, these components are formed by a set of design-level entities. Neither huge clusters nor singletons are usual in architectural components. Thus, one measure of clustering quality could be the non-extremity of the generated clusters. Wu et al. have proposed a measure called non-extreme distribution (NED) [Wu et al. 2005]. NED can be defined as:

$$NED = \frac{\sum_{\substack{i=1 \\ i \text{ not extreme}}}^{k} n_i}{n} \tag{5.2}$$

where k is the number of clusters in the partition, n_i is the size of cluster i and n is the number of entities to be clustered.

Obviously, the limits of non-extremity depend on system size. Usually clusters with less than 5 entities are considered extreme (dust clouds), while the upper limit, from which clusters are considered black holes, may vary depending on the system. In this work, 20 was chosen as the upper limit, both because some of the systems were not very large and because reducing one order of magnitude, instead of two orders, seemed an appropriate choice for abstracting modules from lower-level code.

For example, a clustering of a Java system with 100 classes (entities) that produces 1 extreme cluster with size 50, 3 non-extreme clusters with size 10, and 20 extreme clusters of size 1, has a non-extremity of 30%, because only 30% of the classes fall into non-extreme clusters.

One important concern regarding cluster average size is that a good clustering algorithm should reduce the order of magnitude of system complexity so that it is easier to

understand. Supposing hierarchical clustering, it could be defined, as a rule of thumb, that reducing one order of magnitude in each clustering application should be a reasonable expectation for a good clustering algorithm.

5.4 Experimental Design

The experimental procedures followed four steps: fact extraction, design abstraction, software clustering and comparison.

The first three steps were performed with the *Design Suite* toolset, presented in Appendix A. In the last step, clustering algorithms were compared in terms of the criteria described in sub-section 5.3: authoritativeness, stability and non-extremity. For each criterion, curves were plotted for better comprehension and relatives measures were derived so that algorithms could be ordered according to their results.

The process is illustrated in figure 5.3 and detailed below. For this experimental design, a case study based on evolving software versions was used.

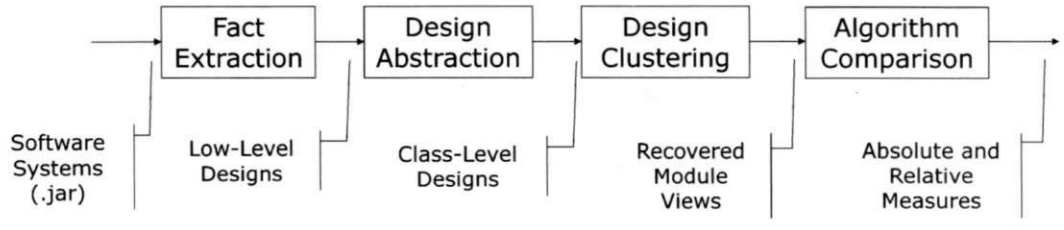


Figure 5.3: Experimental design layout

5.4.1 Choice of Subject Systems

Since the aim of this work is providing empirical evidence about the power of automated clustering algorithms, applying them in real-world publicly available systems was a straighforward idea. The evaluation was accomplished with software available from SourceForge, an open source software repository [SourceForge 2010]. The requirements for choosing the systems for the case study were: popular systems with a

Table 5.1: Subject systems

System Versions	# of versions	Size (KLOC)	Design Level	# of nodes	# of Edges	Graph Size (KB)
JUnit 2.0-4.5	14	1.4-9.5	code	368-2464	935-7017	254-1860
			design	20-133	51-507	16-134
EasyMock 1.0-2.4	13	1.3-5.8	code	407-831	1204-2128	311-589
			design	20-63	63-192	18-55
JEdit 2.3-4.2	21	36.1-140.7	code	225-7931	556-34095	156-8145
			design	19-312	20-1938	7-443
JabRef 1.0-2.4b	35	17.8-110.0	code	2524-11259	7270-33990	1885-8702
			design	91-461	394-2598	100-613

large number of downloads, source code in Java, and availability of at least ten different released versions. Their *.jar* files were input to design extraction after removing third-party libraries. Table 5.1 shows a summary of the four systems, with data about the evolution of their source code and designs.

Consecutive versions of the four systems were input to the experiment. The research question was: *how do the different clustering algorithms behave during software evolution in terms of stability, authoritativeness and non-extremity?*

5.4.2 Validity Evaluation

Results from the case study cannot be generalized to contexts different from the chosen systems. Nevertheless, I tried to reduce external validity threats by choosing popular systems from a well-known open source repository (SourceForge), by using systems developed in an industrial language (Java), and by choosing systems with a large evolution period.

The construct validity is threatened by lack of consensus in the community about the most appropriate measures to evaluate clusterings. Authoritativeness and stability are relevant concerns for evaluation, but the *MoJo* measure may not appropriately capture the differences between two clusterings, since it does not take relations between entities into account [Mitchell and Mancoridis 2001a]. In addition, using package clusterings as an oracle for authoritative clusterings is limited, since this type of aggregation is only one of the different views software developers are interested in. However, since evaluations like this work are not common in the literature, the empirical

results found here, together with the related literature [Koschke and Eisenbarth 2000; Anquetil et al. 1999; Mitchell and Mancoridis 2001a; Wu et al. 2005], may help catalyze a discussion about appropriate measures to evaluate software clusterings. The construct may be threatened as well by the implementation of the algorithms, which were coded according the the specification available in the literature. This threat was mitigated by the use of unit and integration tests, an attempt to assure the correct implementation of the algorithms.

Typical internal validity threats are not present in this retrospective study, since all measures are taken from previous history extracted from software repositories. The factor studied is the clustering algorithm, and all clustering algorithms are applied to the same subject systems, and the measures are taken in a retrospective fashion. Developers were not aware of these measures during development time, thus, mitigating most internal validity threats.

Since this is a case study based on the selection of four typical cases, and not a randomized experiment, the issue of conclusion validity is not important. The use of statistical tests would make sense in a different experimental setting, but not in this context.

5.5 Results

Results are shown below. Charts were plotted for authoritativeness, non-extremity and stability. Relative measures were derived to better position each algorithm in relation to the others.

5.5.1 Relative Measures

To compare data series, Wu et al. defined ordinal measures to rank two or more data series [Wu et al. 2005].

For two series DS_i and DS_j, the relative measure $Above$ is defined as:

$$Above(DS_i, DS_j) = \frac{|\{n|DS_i[n] > DS_j[n], 1 \leq n \leq |DS_i|\}|}{|DS_i|} \tag{5.3}$$

Table 5.2: Relative non-extremity scores

	System			
Algorithm	**JUnit**	**EasyMock**	**JEdit**	**JabRef**
eb	0.27	0.14	0.36	0.00
km	3.00	2.64	2.84	2.94
mq	0.80	1.00	1.44	1.34
dsm	1.87	2.14	1.24	1.71

For k data series, the relative measure for a particular series DS_i in relation to all the other k series is defined as:

$$Above(DS_i) = \sum_{j=1}^{k} Above(DS_i, DS_j) \tag{5.4}$$

Another relative measure useful to classify data series into high, medium and low regions is the HML measure, defined as:

$$HML(DS_i) = \begin{cases} H & \text{if } Above(DS_i, hm) \geq 0.8 \\ M & \text{else if } Above(DS_i, ml) \geq 0.8 \\ L & \text{otherwise} \end{cases} \tag{5.5}$$

where hm and ml are constant-valued series that respectively divide the high and medium regions and the medium and low regions. The exact values of hm and ml will depend on the specific comparison.

5.5.2 Non-extremity of cluster distribution

The non-extreme distribution (NED) measure was calculated for the case study, as defined in 5.3.3. Figure 5.4 shows the NED data series for each algorithm for 35 versions of *JabRef*. The other NED series, omitted here for the sake of brevity, are shown in Appendix B.

Table 5.2 shows the relative non-extremity measure $Above$ and Table 5.3, the non-extremity HML values, for each algorithm. The hm and ml thresholds were set to 0.75 and 0.50, respectively.

The results suggest that *km* clustering performs best in terms of non-extremity, fol-

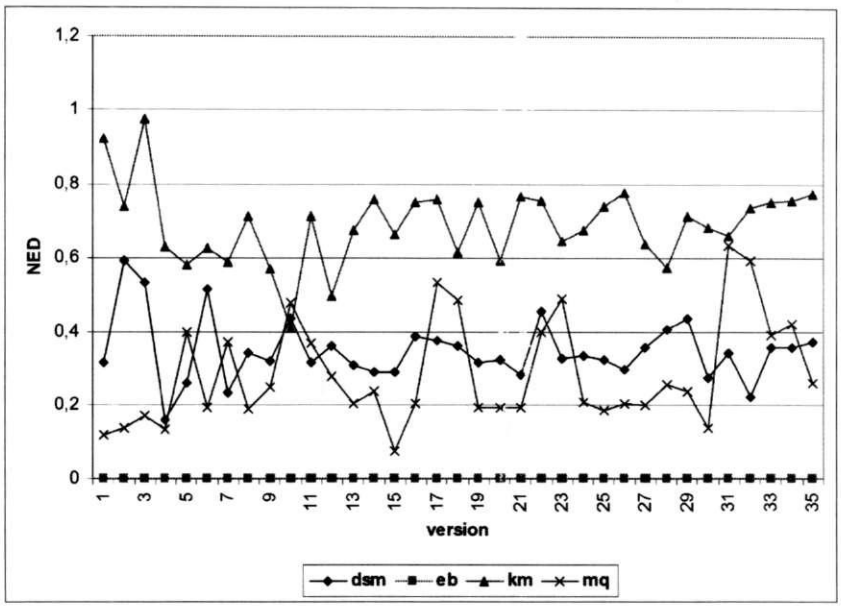


Figure 5.4: NED scores for *JabRef*

Table 5.3: HML non-extremity scores

	System			
Algorithm	**JUnit**	**EasyMock**	**JEdit**	**JabRef**
eb	L	L	L	L
km	H	L	L	M
mq	L	L	L	L
dsm	L	L	L	L

lowed by *dsm* and *mq*, and *eb* performs worst. The number of clusters in *km* is parameterized to 10% of the number of entities, thus, easily forming non-extreme clusters. Furthermore, most clusterers produce low NED values (below 0.5), except for *km*, that produces, on average, medium NED values. Looking closely at the clusters formed, one can see that dsm and mq form some non-extreme clusters and many small clusters, while eb usually forms one huge cluster and many singletons.

5.5.3 Authoritativeness

For each version of the four systems in the case study, the similarity between the partition P formed by the four studied algorithms and the authoritative partition PA formed by the package decomposition was computed, using the $MoJoSim(P, PA)$ measure, as defined in 5.3.1. Figure 5.5 shows the $MoJoSim$ data series for each algorithm for 35 versions of *JabRef*. The other $MoJoSim$ series are omitted here, but are available in Appendix B.

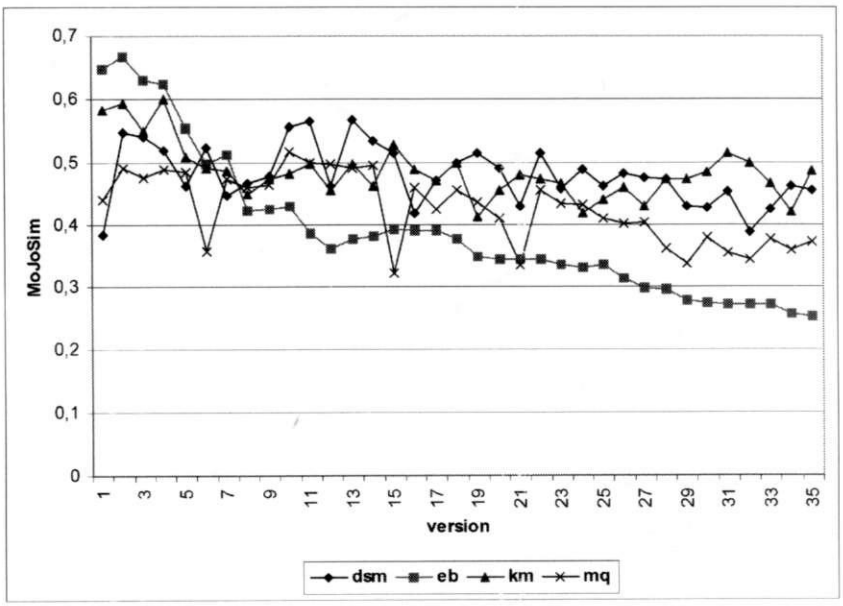


Figure 5.5: $MoJoSim$ authoritativeness scores for *JabRef* versions

Table 5.4 shows the relative authoritativeness measure $Above$ and Table 5.5, the HML authoritativeness values, for each algorithm. The hm and ml thresholds were set to

Table 5.4: Relative authoritativeness scores

	System			
Algorithm	**JUnit**	**EasyMock**	**JEdit**	**JabRef**
eb	0.00	0.92	0.52	0.63
km	1.73	2.50	2.76	2.05
mq	1.87	1.00	1.28	1.08
dsm	1.93	1.07	1.32	2.14

Table 5.5: HML authoritativeness scores

	System			
Algorithm	**JUnit**	**EasyMock**	**JEdit**	**JabRef**
eb	L	M	L	L
km	M	M	M	M
mq	M	M	M	L
dsm	M	M	L	M

0.80 and 0.50, respectively.

The results point that *km* and *dsm* compete for best authoritativeness, with some advantage for *km*, being followed by *mq*. Algorithm *eb* ranks worst. HML scores show that, on average, all algorithms but eb rank medium authoritativeness, while eb ranks produces low scores.

5.5.4 Stability

Finally, clustering stability for the four systems was measured. To do such, the similarity between partitions of consecutive versions of each system was measured, using the measure $MoJoSim(P_i, P_{i+1})$, as described in 5.3.2. Figures 5.6 and 5.7 respectively show $MoJoSim$ stability data series for each algorithm for *JUnit* and *JabRef*. The other charts are omitted here for brevity, but are shown in Appendix B.

For the evolving versions of the four systems, Table 5.6 shows the relative stability measure $Above$ and Table 5.7, the HML stability values, for each algorithm. The hm and ml thresholds were set to 0.90 and 0.70, respectively.

From the figures and tables, it is easy to notice that *eb* is too stable, producing similar clusters (black holes and dust clouds) along all versions. The algorithm *mq* is more stable than *dsm* and *km* is the least stable. Except for *eb*, all the other algorithms present, on average, low stability, as can be seen from the HML scores.

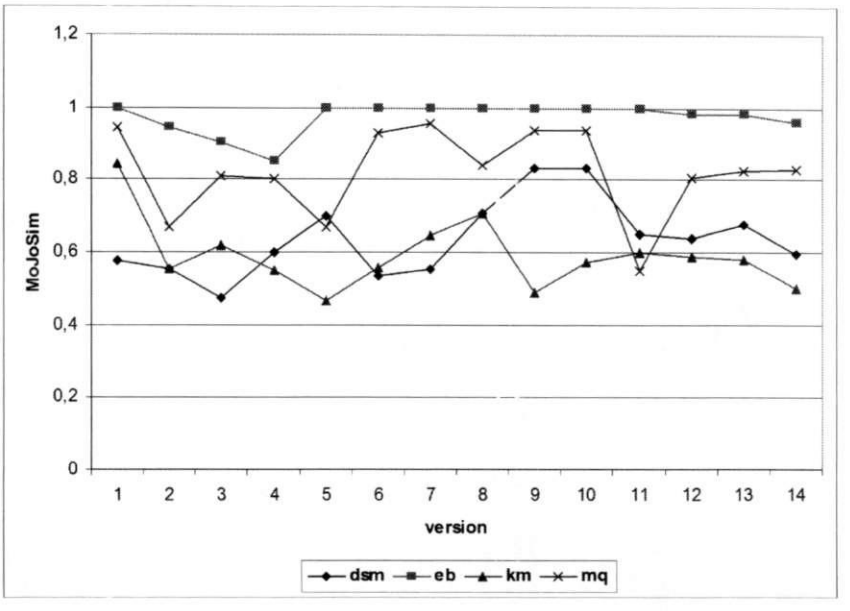


Figure 5.6: *MojoSim* stability scores for *JUnit*

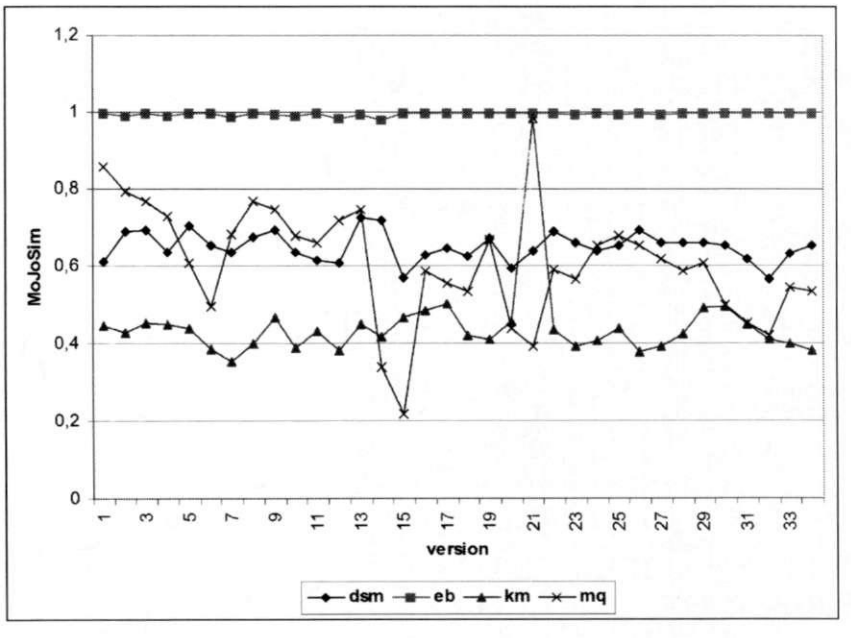


Figure 5.7: *MojoSim* stability scores for *JabRef*

Table 5.6: Relative stability scores

	System			
Algorithm	**JUnit**	**EasyMock**	**JEdit**	**JabRef**
eb	3.00	2.62	2.58	3.00
km	0.36	0.31	0.33	0.15
mq	1.79	1.85	1.46	1.26
dsm	0.71	0.61	0.83	1.59

Table 5.7: HML stability scores

	System			
Algorithm	**JUnit**	**EasyMock**	**JEdit**	**JabRef**
eb	H	H	H	H
km	L	L	L	L
mq	L	M	L	L
dsm	L	L	L	L

5.5.5 Summary of Results

The three studied dimensions for each algorithm are summarized in Table 5.8.

5.6 Analysis

Edge betweenness clustering (*eb*): although this algorithm is useful to discover communities in social networks, it does not perform well for software clustering. It typically generates a huge cluster and many tiny clusters. Its authoritativeness is also low, usually below 0.40, especially for larger systems. It is also too stable, showing no capacity to reflect architecture changes during software evolution.

K-means clustering (*km*): this typical algorithm for pattern recognition, combined with a distance better adapted to graph clustering (Jaccard distance) performs medium in terms of non-extremity and authority. Adjusting the number of clusters to 10% of the number of vertices allows reducing one order of magnitude in system complex-

Table 5.8: Evaluation summary

	Measure		
Algorithm	**Non-extremity**	**Authoritativeness**	**Stability**
eb	Low	Low	High
km	Medium	Medium	Low
mq	Low	Medium	Low
dsm	Low	Medium	Low

ity for each clustering application, making K-means especially suited to hierarchical clustering. Its authoritativeness ranges from 0.40 to 0.80, although the higher values are more usual. It is the less stable algorithm, with stability usually ranging from 0.35 to 0.60, which may be an issue when studying module view changes during software evolution.

Modularization quality clustering (*mq*): this algorithm was based on the steepest ascent hill climbing, using a modularization quality optimization function, as suggested in the papers about the *Bunch* tool [Mancoridis et al. 1998; Mancoridis et al. 1999]. From the three useful algorithms (since *eb* only produced useless clusters), it provides the highest stability, being especially useful for software evolution studies. On the other hand, it produces clusterings with medium authoritativeness, usually a little worse than *dsm* and *km*, with values ranging from 0.30 to 0.70. Non-extremity is only less worse than *eb*, with non-extreme cluster distribution ranging from 0.1 to 0.5.

Design structure matrix clustering (*dsm*): this algorithm behavior is, in a way, similar to *mq*, due to its non-deterministic behavior related to the optimization process, but shows better figures for non-extremity and authoritativeness. Its NED typically varies from 0.2 to 0.7 and its authoritativeness typically ranges from 0.35 to 0.75. Although worse, on average, than *km*, it outperforms this algorithm in authoritativeness for system versions of *JUnit* and *JabRef*. In terms of stability, it is worse than *mq* and better than *km*.

Results were also compared with the work of Wu and colleagues [Wu et al. 2005]. Measures were kept the same as defined by Wu et al., for both authoritativeness and stability. Although their absolute measures were computed from dissimilarity, relative measures were equivalent, with the same thresholds kept. On the other hand, for non-extremity, the NED measure was changed, so that clusters with more than 20 entities were considered extreme, instead of 100. That was because the rationale was that each clustering application in an hierarchical clustering context should reduce only one order of magnitude, on average, in system complexity. Besides that, one additional difference is that the studied software in this work is object-oriented Java code, while theirs seems to be procedural C code.

Results show that *eb* falls in the same category of the least useful algorithms of single linkage in their work. The results of *mq* agree with their results of the similar algorithm in the *Bunch* tool in terms of stability, but disagree in terms of authoritativeness (*mq* performs medium while *Bunch* performs low) and non-extremity (*mq* performs low while *Bunch* performs high). This divergence, thus, points to the need of additional studies for this algorithm. Probably, that is because *Bunch* uses some additional heuristics to avoid dust clouds that commonly happen in *mq* [Mitchell and Mancoridis 2006]. Finally, in this study, *km*, *mq* and *dsm* all present medium authoritativeness, while all their algorithms perform low in this dimension. Whether that is due to package decomposition in Java OO software reflecting better the modular architecture than directory decomposition in procedural C software remains an issue to be studied.

One additional evaluation dimension not previously discussed here is efficiency. All the studied algorithms perform well with small designs (e.g.: less than 500 classes). However, when designs grow to more than 1000 classes, the algorithms *mq* and *dsm* take significantly longer to converge to good clusterings. Although the complexity of these algorithms is polynomial, with much larger designs they may become unfeasible. Thus, additional work to optimize the calculations involved in *mq* and *dsm* optimization functions may prove relevant for larger software. For instance, in the implementation in this work, time was traded for space to improve *dsm* performance.

5.7 Discussion

Unfortunately, the results show that all the studied algorithms are incapable of deriving fully automatic architecture decompositions, since they still generate many tiny clusters that do not correspond to real architectural modules. K-means clustering, that produces less tiny clusters, has a drawback, requiring a predefined number of clusters, which can be hard to determine in advance in most cases. These results point to the need for additional processing after clustering: either manual adjustments to better position misplaced entities or automated techniques such as orphan adoption [Tzerpos 1997] that could reduce the number of tiny clusters. Thus, with this additional pro-

cessing, automated clustering techniques may show its value as an important step in architecture recovery.

On the other hand, the use of quantitative measures to evaluate architecture decompositions has some inherent limitations. Real authoritativeness depends on expert decompositions made by architects, which are not always available, especially in the context of software evolution. The adopted solution of package decompositions as an authoritative partition can only approximate expert decompositions. Non-extremity of cluster distribution has the limitation of ignoring some exceptions in architecture decompositions such as singleton modules with architectural meaning (e.g.: façade). Finally, stability is relative: sometimes, one needs stability because small changes with no architectural impact should not disturb automatic decompositions, but, at times, changes may be significant and affect the architecture.

One final remark is that the software architecture of a system has to express design rules that satisfy the authority of a specific architect. Automated clustering usually produces generic decompositions that may prove useful for documentation and comprehension, but may not express the decomposition that the actual system architect wished, the way he or she understands the system. Current research focus has been on finding a one-size-fits-all solution, based on algorithms that use general clustering heuristics. However, it seems appropriate to move the focus to decomposition techniques that abide by design rules expressed by an architect.

Thus, the results found and the limitation of the measures used point to the need of additional research that combines quantitative and qualitative methods. Both subjective analysis of recovered views and authoritativeness-oriented decomposition techniques should be pursued. It is also important to improve existing metrics and developing new benchmarks of authoritative decompositions, since the available ones are typically targeted at procedural C software. Last but not least, there is still room for further empirical investigation with other clustering algorithms.

5.8 Summary

This empirical evaluation compared four graph clustering algorithms in the context of software architecture module view recovery. The domain of investigation was restricted to open source object-oriented Java software available in public repositories. A case study analyzed consecutive released versions of four systems. Evaluation dimensions were non-extremity of cluster distribution, authoritativeness of generated decompositions and stability of decompositions during software evolution. Algorithms were compared through relative measures derived from data series for each algorithm and for each dimension.

Results showed that K-means clustering (*km*) performs best in terms of non-extremity and authoritativeness, followed by design structure matrix clustering (*dsm*) and modularization clustering (*mq*). Edge betweenness clustering (*eb*) performs worst, generating meaningless decompositions. For stability during software evolution, the analysis showed that *mq* performs best, followed by *dsm*, and that *km* is the least stable algorithm. Edge betweenness ranked highest in stability, but its results were irrelevant since the decompositions were meaningless.

Finally, relative measures showed that *km* ranks medium in non-extremity while the others rank low, that *eb* ranks low in authoritativeness while the others rank medium, and that *eb* ranks high in stability while the others rank low. These results add to the body of empirical knowledge regarding comparison of software clustering algorithms, especially in the recovery of static software architecture module views.

This study tried to answer the question of whether architecture recovery techniques based on software clustering can enable the generation of high-level models in the context of software evolution. Results based on quantitative criteria show that the studied techniques are limited to automate the generation of high-level models. On the other hand, collected empirical evidence suggests that further studies based on more solid benchmarks, which should be more consensual in the academic community, are needed to evaluate the power of these techniques to recover high-level models.

In terms of the main research question (reducing manual effort to apply a confor-

mance checking technique in the context of evolving, sparsely documented software), I found no strong empirical evidence that clustering algorithms can reduce the effort in the generation of high-level models. This is the answer, at least, for the studied algorithms applied to the evolving versions of the subject systems. Nonetheless, this study provides insights based on empirical evidence of how to better evaluate software clustering techniques in order to find valid answers to the research question. Further discussion on this matter and on the use of clustering algorithms to generate and update high-level models in the context of the ERM process is presented in Chapter 8.

Chapter 6

Automated Incremental Mapping Techniques

Técnicas Automáticas de Mapeamento Incremental

Neste capítulo, é apresentada uma técnica de mapeamento incremental de entidades de design de baixo nível para módulos arquiteturais baseada na recuperação de informação do vocabulário do software, assim como a combinação desta técnica com outras baseadas em dependências estruturais. Em seguida, uma avaliação experimental de diversas técnicas de mapeamento é realizada.

In this chapter, I present an incremental mapping technique from low-level design entities onto architectural modules based on information retrieval of software vocabulary. A combination of this technique with other structure-based mapping techniques is also presented, followed by an empirical evaluation of various automated mapping techniques.

6.1 Introduction

As previously stated elsewhere in this work, a limitation of the reflexion model technique is that it depends on a given mapping between code entities and high-level modules. In its original definition, the mapping was stated manually by a developer, with regular expressions used to describe mappings of multiple parts of the implementation at a time.

Figure 6.1 shows an example of a mapping for a Java system that consists of three main architectural modules (*business*, *communication* and *ui*), and a library module (named *rest*). It consists of rather simple regular expressions that map Java classes and interfaces onto the modules, since it is based on the package structure.

```
MODULE          REGULAR_EXPRESSION
business        business\.
communication   communication\.
ui              ui\.
rest            ^(?!.*(business\.|communication\.|ui\.)).*$
```

Figure 6.1: Example of a mapping with regular expressions.

However, when the implementation of a system comprises many entities or the naming of the entities does not follow patterns, specifying the mapping manually with regular expressions can be cumbersome and time-consuming. If conformance checking is to be applied frequently to the system, specifying the mapping manually can become even more cumbersome as the mapping must be maintained as the system evolves.

To ease the effort required to apply the reflexion model technique, approaches have been proposed to automatically produce a mapping [Christl et al. 2005; Christl et al. 2007]. Given a partial mapping, these approaches rely on structural dependencies between implementation entities and on heuristics of structural cohesion and coupling to map implementation entities onto design entities. These approaches can help reduce the effort required to use the reflexion model approach to check incremental changes during software evolution tasks.

In this chapter, I propose automated mapping techniques based on information retrieval (IR) to map implementation entities onto design entities based on the similarity

of vocabulary between the implementation and the design. Similar to the existing automated mapping techniques, the IR-based techniques I introduce are meant to improve an initial mapping to accomodate incremental changes. I also consider whether combining the IR-based techniques with previous structural-based techniques can improve the accuracy of automated mapping. Through experimentation across four systems, I show that the nature of the design model used when applying reflexion model affects which automated mapping technique performs best. I also show that the best results are achieved by combining structural-based and information retrieval-based techniques; a combination of the techniques generally increases recall, while keeping precision similar as the best approach used in isolation.

6.2 Mapping Techniques

The high-level model used in the reflexion model technique is typically small, on the order of ten to fifty entities. The source-level model is large, on the order of tens or hundreds of thousands entities, but is typically extracted automatically. Scalability of the technique depends upon the ease by which a developer can specify the needed mapping. If conformance checking is to be done on a regular basis, the need for the mapping to be easily stated and updated becomes even more important. I describe the existing approaches to ease the specification of the mapping before introducing the IR-based approach.

6.2.1 Manual Mapping Techniques

In the original definition of the reflexion model technique, regular expressions were used to enable a software developer to more easily specify a mapping. This process works well when naming conventions are present and the hierarchical structure of the software can be used to map many implementation entities with one regular expression rule. Others have proposed using tools to help specify the mapping manually. For instance, in the SAVE tool, the software designer may either use list boxes and buttons

to select entities to be mapped or load a manual mapping from a file [Knodel and Popescu 2007].

6.2.2 Automated Mapping Techniques

Automated mapping techniques help incorporate new or changed implementation entities automatically into an existing mapping; for simplicity, I will often refer to the existing mapping as a *pre-mapping*. The new or changed implementation entities to incorporate are called *orphans*. One way of mapping orphans is by using an attraction function, which produces a value for an orphan and design entity pair to represent how likely it is that the orphan should be mapped to the design entity.

Structural Mapping Functions

Christl and colleagues investigated whether the mapping could be automatically created by using a structural attraction function [Christl et al. 2005]. They proposed two attraction functions: $countAttract$ and $MQAttract$. The former relies solely on structural coupling; the value of the function is calculated from the dependencies between the orphan and the implementation entities pre-mapped to the entity in the design model. The latter defines a modularization quality function that takes into account module structural cohesion and inter-module structural coupling.

The function $countAttract$ increases with the structural coupling between the orphan o_i and the design entity (module) m under scrutiny. Equation 6.1 describes the function with w_{ij} representing the coupling between implementation entities o_i and o_j, where coupling is usually computed as the cardinality of the set of implementation dependencies between o_i to o_j.

$$countAttract\,(o_i, m) = \sum_{\forall o_j \,|\text{maps-to}(o_j)=m} w_{ij} \tag{6.1}$$

In comparison, the second function, $MQAttract$, increases when the structural modularization quality improves because an orphan is adopted by a particular module. In

this function, the attraction is higher for the orphan mapping that produces higher module cohesion and lower inter-module coupling. Equation 6.2 describes the function

$$MQAttract\,(o_i, m) = \sum_{k=1}^{|N_M|} CF_k \quad and \quad \text{maps-to}(o_i) = m \tag{6.2}$$

with N_M as the number of modules and the cluster factor CF_k given by

$$CF_k = \begin{cases} 0 & \mu_k = 0 \\ \dfrac{2\mu_k}{2\mu_k + \sum\limits_{\substack{j=1 \\ j \neq k}}^{|N_M|} (\epsilon_{kj} + \epsilon_{jk})} & \text{otherwise} \end{cases}$$

Parameter μ_k is the structural intra-module coupling of module k, while ϵ_{kj} accounts for the inter-module coupling from module k to module j, and ϵ_{jk} works the other way around.

Evaluation results of Christl and colleagues are limited by the use of a filtering function to reduce the number of orphans submitted to automated mapping. Only orphans with a given ratio of dependencies to mapped entities over the total of the orphan's dependencies were input to the mapping technique [Christl et al. 2007]. Such filtering can mask the effective power of a mapping function, especially when comparing it to functions not based on structural dependencies.

Information Retrieval Mapping Functions

Mapping may be seen as an instance of a classification problem. Given an oracle mapping produced by software developers that provides an assignment of each orphan to a module, a mapping technique classifies correctly when it maps an orphan to the expected module in the oracle mapping, classifies incorrectly when it performs a wrong mapping, or does not classify at all when it does not suggest a mapping. Since the vocabulary of implementation entities (e.g., identifiers and comments) frequently captures what each entity semantically represents, it seems natural to use source code vocabulary as feature inputs to the classification problem. Using this idea, I introduce

and investigate two variants of a new automated mapping technique based on information retrieval to map orphan implementation entities onto architecture modules based on the similarity of their vocabularies.

To apply information retrieval (IR) techniques to the mapping problem, I had to adapt IR concepts to the context of software reverse engineering. I consider a software entity as a document and consider that the document text is made up of the entity's identifier as well as any identifiers enclosed in the entity. For example, given an entity that is a Java class, I consider the class document to contain the vocabulary of the class name plus the names of its methods and fields. I consider a module document (representing a design entity) to contain the vocabulary of the module name, plus the vocabulary of the mapped implementation entities. Terms in the vocabulary are stored after tokenizing identifiers, removing stop words and stemming tokens.

One can then use a vector space model to represent the implementation and design entity modules as vectors. Each term from the vocabulary represents one vector dimension. The whole corpus is usually represented as a term-document matrix. Various schemes for computing each vector component in a document have been developed, such as absolute or relative term counts and *tf-idf* weights [Salton et al. 1975]. After a preliminary evaluation, I decided to use relative term counts, which have produced the best classification results among the various schemes.

In the IR-based mapping technique, I use the vector space model to compute the attraction between an orphan and its most similar module in terms of source code vocabulary. Specifically, I define $IRAttract$ as the cosine similarity measure between document vectors. Equation 6.3 defines the $IRAttract$ function:

$$IRAttract\,(o_i, m) = \cos\theta = \frac{\mathbf{o}_i \cdot \mathbf{m}}{\|\mathbf{o}_i\|\,\|\mathbf{m}\|} \tag{6.3}$$

where $\mathbf{o}_i$ and $\mathbf{m}$ are, respectively, the feature vectors of the orphan o_i and of the module m, extracted from source code vocabulary by an information retrieval technique.

Latent semantic indexing (LSI) is a technique that produces a low-rank approximation to the term-document matrix by means of a singular value decomposition [Deerwester et al. 1990]. LSI reduces the vector space by grouping terms into concepts, which

solves the issues of synomymy and polysemy that appear in the basic vector space model. It usually produces better precision and recall during classification. Here, I also investigate the utility of an attraction function based on LSI, called $LSIAttract$, that is based on a reduced vector space with one hundred dimensions.

6.2.3 Mapping Algorithm

An attraction function must be used within an algorithm to produce a mapping given a set of orphans, a set of design entities and a pre-mapping. The algorithm I used was established in the candidate detection technique [Christl et al. 2005]. Given an orphan, this technique generates a candidate set of design entities formed by choosing those with an attraction value that exceeds a threshold. When there is only one design entity in the candidate set, the orphan is considered to be automatically mapped to that design entity.

Choosing the value for the theshold is hard, since attraction values are both system- and function-dependent. To allow comparison to previous work [Christl et al. 2005], I chose a threshold equal to the average attraction of an orphan to the design entities plus its standard deviation. When this threshold generates an empty candidate set, the threshold is lowered to the average attraction. This choice assures there always will be at least one candidate module.

The mapping algorithm above is the same as the one implemented in [Christl et al. 2005], except that entities with few or no relations to existing modules are not filtered out from the mapping process. This choice allows to more evenly compare all the mapping functions.

I also envisaged a variant of the mapping algorithm to take advantage of both structural dependencies and information retrieval functions. In this variant, mapping is first performed using a given mapping function. Then, mapping is performed once more over the remaining unmapped entities using a different mapping function. Changing the order of the mapping functions may change classification results and we decided to investigate this issue as well.

6.3 Evaluation Design

Optimally, an automated mapping technique would correctly assign each implementation entity to the appropriate module, where correct assignment is judged by the software developer applying the technique. There are many variables that could affect the accuracy of an automated mapping technique: the number of modules in the design, the size of the system, the meaning of design modules, whether there is a correct mapping for part of the system available, and the age of the system, amongst others.

To determine which of the automated mapping techniques performs best in terms of correct classification, I conducted two case studies: one in which the design model is an abstract structural view of the system, such as a layered view, and a second in which the design model is a physical break-down of the structure of the system, such as the directory structure of the code. For each case study, I considered four systems and had available an oracle mapping that was produced with the help of the system's developers. Classes were the basic implementation entities used in this evaluation. I took the same approach for each case study, as explained in Figure 6.2. I removed a fraction of the entries from the oracle mapping and placed the implementation entities from these entries into a set of orphans. Using the remaining oracle mapping, I applied six different techniques, four described in the previous section plus two combinations of them, to map each oracle, and compared the resulting mappings of the orphans to the oracle mapping. The choices of the fraction of mapping entries to remove were driven by different scenarios that occur during a system's evolution.

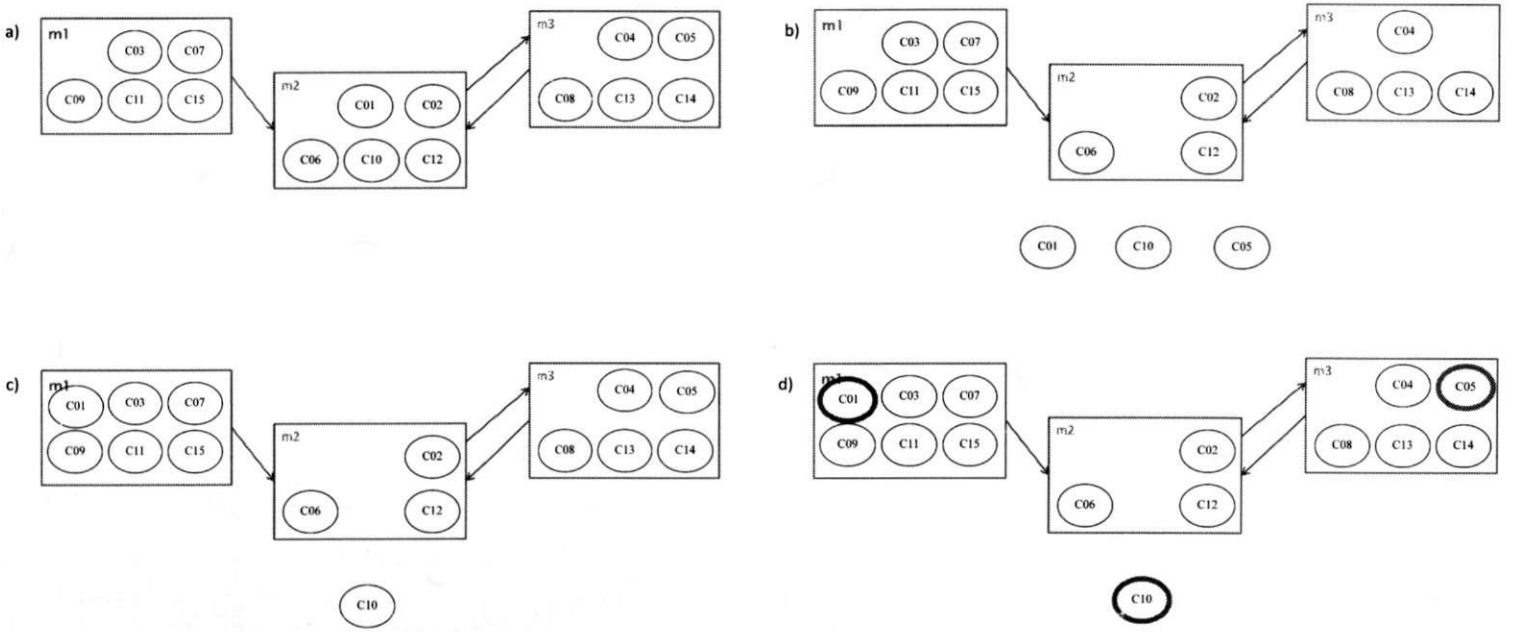


Figure 6.2: Illustrating evaluation design with an example: a) Oracle mapping; b) Orphans removed from mapping; c) Orphans mapped after applying automated mapping technique; d) Measures computed for the automated mapping technique under evaluation (in the example, C01 is incorrectly mapped, C05 in correctly mapped, and C10 is not mapped).

Table 6.1: Systems under study

ID	System	Version Date	Size (KLOC)	Jar Size (KB)	# of classes
DW	Design Wizard	17-May-2010	7.0	75	37
DS	Design Suite	20-Apr-2010	24.2	429	234
OG	OurGrid	20-Apr-2010	119.0	3135	1685
ML	Mylyn	1-Jun-2010	695.8	7671	1412

I describe the evaluation design in further detail before presenting the results.

6.3.1 Target Systems and Designs

I used four systems of varying size and purpose for the evaluation. `Design Wizard`, a tool to extract and query designs, is a research prototype[1] that represents a small system (37 classes). `Design Suite`, a toolset to abstract designs through lifting, clustering, filtering and mapping operations, and to visualize abstractions in different layouts, is a research prototype[2] that represents a medium-sized system (234 classes). `OurGrid`, an open source middleware[3] that enables the creation of peer-to-peer computational grids, represents a large distributed system (1685 classes). Finally, `Mylyn`, an open source Eclipse plugin[4] that provides a task-focused interface to Eclipse IDE to reduce information overload, represents a large system (1412 classes). Table 6.1 provides an overview of each system. All systems were developed in Java.

For $CaseStudy_1$, I gathered a high-level design model from architects and personnel associated with each system. The `OurGrid` architect provided two high-level design models: one based on the layers in the system's architecture and the second providing a run-time view. For $CaseStudy_2$, I gathered more detailed design models, which I will refer to as low-level models. Table 6.2 describes the design models used for each system and each case study. Module view diagrams and oracle mappings can be found in Appendix C. Provided oracle mappings were either based on the project directory structure (`Mylyn` and `OurGrid`) or on a manual architecture recovery step performed by developers (`Design Wizard` and `Design Suite`).

[1] `www.designwizard.org`
[2] `www.gmf.ufcg.edu.br/~roberto/papers/paperWMSWM2009.pdf`
[3] `www.ourgrid.org`
[4] `www.eclipse.org/mylyn`

Table 6.2: Module views

System	View Description	Size
DW	Abstract high-level modules	5
	Detailed structural decomposition	7
DS	Independent standalone components	6
	Standalone components split into their building blocks	17
OG	Layered architecture	4
	Main run-time components: broker, peer, worker and others	6
	Layers split into building blocks	8
ML	Abstract high-level components	9
	Physical breakdown of system structure into directories	27

6.3.2 Scenarios

Conformance checking is performed after changes to a software system. Previous studies on software changes have shown that changes typically follow a power-law long tail distribution, with most changes touching very few classes and few changes touching many classes [Hattori and Lanza 2008]. Given this distribution for the size of changes, I consider three different scenarios for the use of an automated mapping technique: singleton changes, small changes (two up to six classes) and large changes (greater than six classes). In the following scenarios, I used classes as the basic implementation entities to be submitted to a mapping procedure.

Scenario 1: Singleton Changes Singleton changes account for roughly half or more of software changes [Hattori and Lanza 2008]. In the evaluation, I modelled these singleton changes by removing each class from the oracle mapping in turn and applied the automated mapping technique to produce a mapping for the class. This scenario models small changes, such as bug fixes, and small increments that might occur in systems with frequent commits.

Scenario 2: Small Changes Changes involving from two to six classes account for around 20 to 40% of commits in a rough estimate extracted from a study of three large open source systems.[5] For the evaluation, I modelled small changes as the addition of a small number of classes. I produced small changes by randomly removing a number of classes from the oracle mapping and adding them to an orphan set. For sizes of

[5]Personal communication with Neil Thomas. June 5, 2010.

Table 6.3: Mapping techniques

Name	Description of Mapping Technique
count	Uses *countAttract* function
mq	Uses *MQAttract* function
ir	Uses *IRAttract* function
lsi	Uses *LSIAttract* function
lsi_count	Uses *LSIAttract* function followed by *countAttract* function
count_lsi	Uses *countAttract* function followed by *LSIAttract* function

small changes between two and six classes, I produced 100 random removals of each size, resulting in 500 trials. To make the randomly produced changes more realistic, only the first removed entity was chosen in a fully random fashion. The following entities in the change were randomly selected either from the first entity's neighbors (with 50% probability) or from neighbors of the other removed entities (also with 50% probability). This process tried to model incremental changes, which can be both broad, with changes localized around an entity, and deep, with changes crosscutting the software modules.

Scenario 3: Large Changes I modelled large changes as the addition of a large fraction of new classes. I chose changes ranging in size from 10 to 100 classes in increments of ten. For each change size, I produced 50 random removals, resulting in 500 trials. Different from the previous scenario, all removals were fully random. This case study tried to resemble large changes, such as major features additions or restructurings that affect the whole code.

6.3.3 Mapping Techniques

I evaluated six different mapping techniques (Table 6.3). Four techniques were obtained by combining the attraction functions described in Section 6.2 with the algorithm in Section 6.2.3. The other two techniques were obtained by first using the mapping algorithm to map the orphans with one mapping function and then applying the algorithm once more with a different function to map the remaining orphans. For these two functions, we chose the combinations with the best average precision in a preliminary assessment.

6.3.4 Measures

Correct classification is given by an oracle mapping supplied by the software architects. For the automated mapping evaluation, I need to identify the subset of the oracle mapping that will form the orphans to be mapped. This subset is given by the relation $CM = \{(o_1, cm_1), (o_2, cm_2), \ldots, (o_{n_c}, cm_{n_c})\}$, where o_i stands for each orphan entity i in a software change c with size n_c and cm_i stands for the correct module to map o_i, $1 \leq i \leq n_c$.

For each trial in each scenario, I apply the mapping technique k and obtain the relation $M_k = \{(o_1, m_{1k}), (o_2, m_{2k}), \ldots, (o_{n_c}k, m_{n_ck})\}$. It may happen that one or more orphans is not mapped by the technique, reducing the cardinality of the M_k relation in comparison to the CM relation.

By comparing each technique's mapping M_k to the oracle mapping CM, I was able to compute standard classification measures such as *precision*, *recall* and *F-measure*.

Random effects are taken into account by means of replicating trials within each scenario. Average measures are, thus, good candidates for dependent variables. Three measures were taken as the dependent variables in the case studies:

(a) *average precision*: the ratio of the technique's correct mappings (tcm) over its mappings(tm), given by Equation 6.4, averaged over a number of software changes;

(b) *average recall*: the ratio of the technique's correct mappings (tcm) over the expected oracle mappings (cm), given by Equation 6.5, averaged over a number of software changes;

(c) *average F-measure*: the harmonic mean of precision and recall, given by Equation 6.6, averaged over a number of software changes.

$$precision = \frac{tcm}{tm} \tag{6.4}$$

$$recall = \frac{tcm}{cm} \tag{6.5}$$

$$F\text{-}measure = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{6.6}$$

6.3.5 Validity Evaluation

This study is limited in the ability to generalize results to other systems — external validity — because the evaluation considered only four systems and all systems were developed in Java. The range of sizes and purposes of the systems considered does help to mitigate this concern to some extent.

The construct validity of the findings is also threatened because the way in which I model incremental changes may not necessarily represent real changes. In practice, changes usually consist of simultaneous additions, removals and changes both in the implementation entities and in their relations, whereas the evaluation considers only purely additive changes. Similarly, the model for large changes, which randomly spreads changes over all modules, may not reflect actual large changes.

Typical internal validity threats are not present in this study, since all measures are retrospective and taken from existing software systems. The factor studied is the mapping algorithm, and all mapping algorithms are applied to the same subject systems. Furthermore, measures are taken retrospectively. Developers were not aware of these measures during development time, thus, mitigating most internal validity threats.

Finally, the conclusion validity of the results in this study is limited due to the absence of statistical tests. Nonetheless, results are averaged over a large number of trials. There are 100 trials for each change size for the scenarios of small and large changes. And in the scenario of singleton changes, the whole population of possible class additions is used. Although statistical tests could improve the confidence in the results, I argue that it is more important to present a descriptive behavior of the algorithms for the three scenarios as it is done in this work, than to use statistical tests. Statistical tests would produce results valid only for the four studied systems in the case studies, and not for a larger population of, for instance, the set of all open source systems available in repositories such as SourceForge.

6.4 Results

Below I describe how the mapping techniques performed across the four systems and the different kinds of models. For brevity and clarity, I show only the most relevant results in this chapter. Readers are referred to Appendix D for the complete evaluation results.

6.4.1 $CaseStudy_1$ Results

I first consider the results when the mapping techniques are used for the high-level design models.

Scenario 1: Singleton Changes Table 6.4 displays the average *F-measure* for each mapping technique applied to singleton changes in high-level models. Results show that automated mapping onto high-level views seems a promising approach for the most common kind of software changes. Large values of *F-measure* (between 0.78 and 0.92) were found for each system for more than one technique, which shows that full automation may correctly solve a large part of the mapping problem. For the mapping functions, $countAttract$ provides the best attraction values with the exception of the `OurGrid` layered view. The lower values for `OurGrid` layered view may be due to the fact that entities inside a layer may not be as strongly connected to each other. $MQAttract$ shows no regular pattern: it does not discriminate for `OurGrid`, but gives good answers for `Design Suite` and `Mylyn`. Information retrieval functions perform worse than $countAttract$, with the exception for `OurGrid` layered view. $LSIAttract$ generally gives better results than $IRAttract$. The combination of $countAttract$ and $LSIAttract$ in a double mapping technique also improves the *F-measure*, which shows that structural dependencies and information retrieval may be complementary. The improvement is usually maximized when the most precise function is used in the first mapping, with the exception for the `Design Wizard` view.

Figure 6.3 captures another perspective of the mapping techniques: the ratio of cor-

Table 6.4: F-measure for singleton changes: high-level views

Mapping Technique	Module View				
	DW	**DS**	**Mylyn**	**OG1**	**OG2**
count	0.76	0.89	0.85	0.59	0.81
mq	0.54	0.87	0.79	0.14	0.08
ir	0.68	0.75	0.57	0.70	0.54
lsi	0.62	0.82	0.63	0.72	0.58
lsi_count	0.78	0.88	0.81	0.79	0.68
count_lsi	0.76	0.92	0.88	0.73	0.84

rectly mapped, incorrectly mapped and unmapped classes over the total of mappings done for the scenario of singleton changes for `Mylyn` and `OurGrid` high-level views. The results show that 78 to 92% of the changes are correctly mapped fully automatically, significantly reducing the work left to the developer. Very few entities—between 1 and 6% for the best performing techniques—are left unmapped. On the other hand, incorrectly mapped classes account for between 6 and 16% in the best results, which suggests that it is important that developers to review mappings soon after performed to fix potential misclassifications.

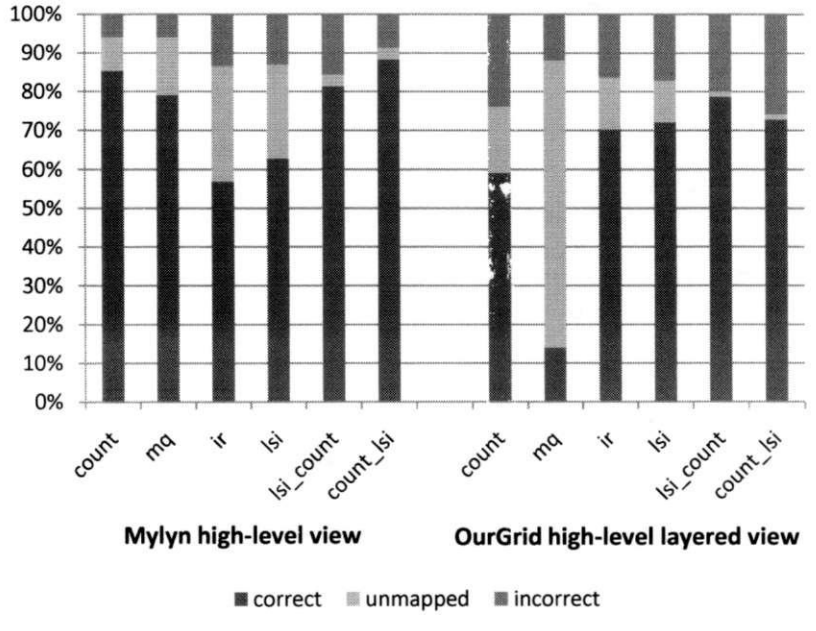


Figure 6.3: Classification distribution for singleton changes in two high-level views

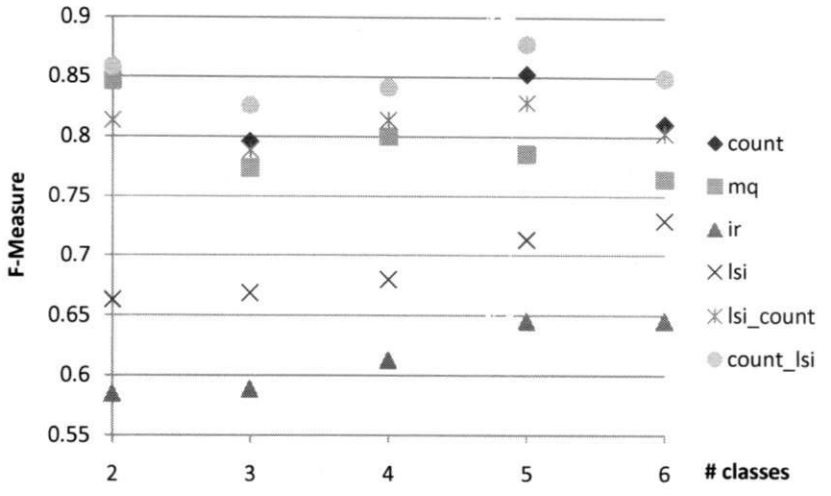


Figure 6.4: F-measure for small changes: Mylyn high-level view

Scenario 2: Small Changes Figures 6.4 and 6.5 show the average *F-measure* for each mapping technique applied to small changes in two representative high-level views.There seems to be no visible trend for each technique when the change size increases from 2 to 6. Large values of *F-measure*, between 0.7 and 0.9, are found for the best performing techniques. In isolation, $CountAttract$ performs generally better for `OurGrid` component view, `Design Wizard` and `Mylyn`, and $LSIAttract$ performs better for `OurGrid` layered view and `Design Suite`. Combined techniques almost always outperform isolated techniques, and maximization is achieved when the most precise function is used in the first mapping.

Scenario 3: Large Changes Although graphs for large changes (from 10 up to 100 classes, except for `Design Wizard`, where the maximum change size was of 30 classes) are not shown here for the sake of brevity, results were similar to the ones in small changes. A decreasing trend is observed in the *F-measure* for `Design Suite` and `Design Wizard`, since larger changes mean a significantly smaller pre-mapping for these systems. For `Mylyn` and `OurGrid`, there is no observable decreasing trend, which is explained by the fact that changes with 100 classes account for less than 10% of the system size. Other than that, results were similar, with combined techniques performing best, with values between 0.75 and 0.92.

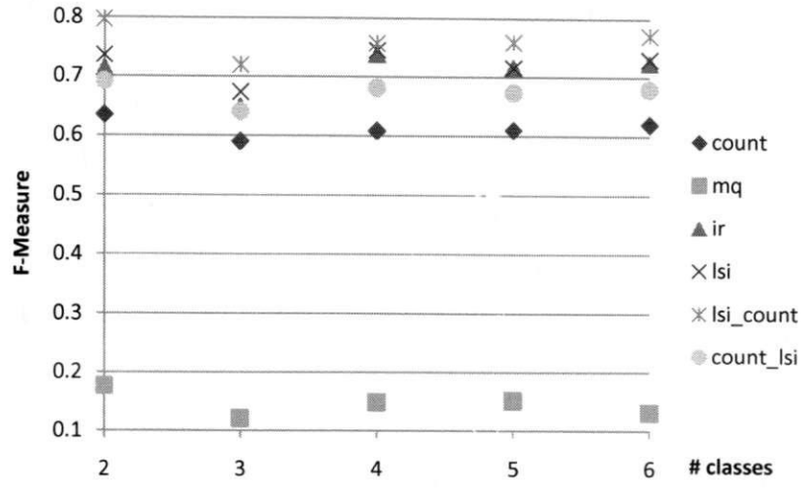


Figure 6.5: F-measure for small changes: OurGrid high-level view

6.4.2 $CaseStudy_2$ Results

Scenario 1: Singleton Changes Table 6.5 displays the average *F-measure* for each mapping technique applied to singleton changes in low-level views. Results show that medium values of *F-measure* were obtained, with the best values between 0.49 and 0.79. Looking at the functions in isolation, $countAttract$ was generally better than the others, except for `OurGrid`. $MQAttract$ was irregular at classifying, performing medium for `Design Wizard` and `Design Suite`, but performing very low on the large systems. $LSIAttract$ was better than $IRAttract$, which performed null on `Mylyn`. $LSIAttract$ performed well on `OurGrid`, medium on `Design Wizard`, and low on `Design Suite` and `Mylyn`. Combining functions in a double mapping procedure improved values for *F-measure*, reaching the best results in all cases. Performing structural mapping before LSI mapping was generally better, except for `OurGrid` view, which was formed by layers decomposed into building blocks. In that case, starting with LSI gave better results. One point worth mentioning was the difficulty to correctly map to the `Mylyn` view, whatever the chosen mapping technique.

Figure 6.6 shows the ratio of correctly mapped, incorrectly mapped and unmapped implementation entities over the total of mappings for the scenario of singleton changes

Table 6.5: F-measure for singleton changes: low-level views

Mapping Technique	Module View			
	DW	**DS**	**Mylyn**	**OG**
count	0.64	0.59	0.43	0.57
mq	0.42	0.54	0.11	0.09
ir	0.61	0.21	0.00	0.67
lsi	0.53	0.36	0.11	0.70
lsi_count	0.67	0.69	0.49	0.79
count_lsi	0.69	0.69	0.49	0.72

for `Mylyn` and `OurGrid` low-level views. Results are not as good as in $CaseStudy_1$; 49 to 79% of the changes are correctly mapped fully automatically. Between 3 and 44% for the best performing techniques are left unmapped and must be handled by a developer. Two systems had a large ratio of unmapped classes: `Design Suite` and `Mylyn`. A number of classes were also incorrectly mapped, accounting for between 7 and 28% in the best results. The other two systems, `Design Wizard` and `OurGrid`, had a large ratio of incorrect mappings, which suggests that, as with high-level views, it is also important to fix eventual misclassifications by reviewing mappings right after performed.

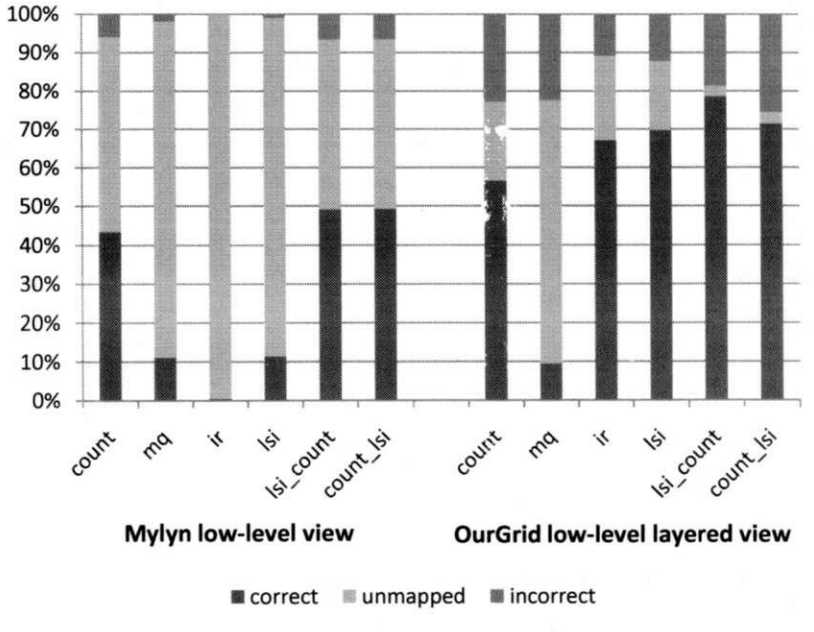


Figure 6.6: Classification distribution for singleton changes in two low-level views

Scenario 2: Small Changes Figures 6.7 and 6.8 show the average *F-measure* for each mapping technique applied to small changes in two representative low-level views. Small changes showed no perceivable trend when change size grows from 2 to 6 classes. Looking each function in isolation, $countAttract$ performed best for three systems, except for `OurGrid`, where $LSIAttract$ was best. Information retrieval functions performed very low on `Mylyn`, but, in general, $LSIAttract$ was better than $IRAttract$. As in the singleton changes, $MQAttract$ performed low on the large systems, and medium, on the smaller. Combined mapping performed best in all cases, with figures between 0.48 and 0.8, starting with $LSIAttract$ for `OurGrid` and starting with $countAttract$ for the other three systems.

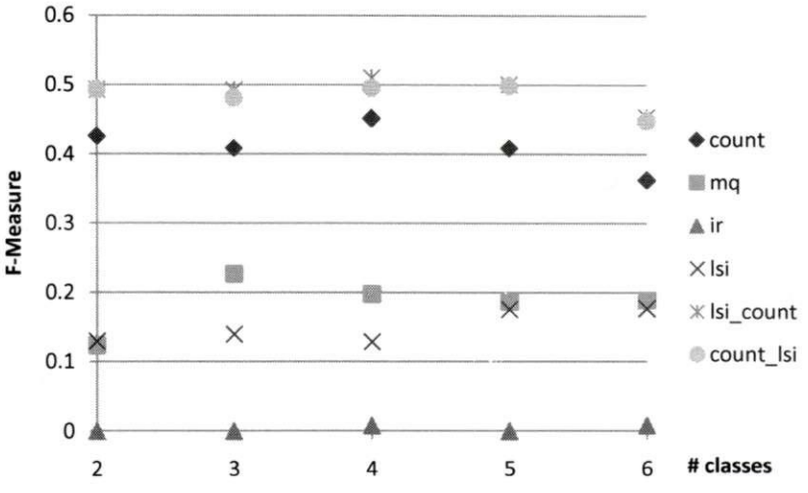


Figure 6.7: F-measure for small changes: Mylyn low-level view

Scenario 3: Large Changes Large changes (from 10 to 100 classes) were not different from small changes. A decreasing trend in *F-measure* values was noticed in `Design Wizard` for all techniques, probably due to the smaller pre-mapping. For `Design Suite`, a slightly decreasing trend was noticed in the structural techniques, while no trend was observed for `Mylyn` and `OurGrid`. The comparison between techniques was similar to the small changes, with values of *F-measure* slightly better than in that scenario, but the relative order of the results was mostly the same, with combined techniques performing between 0.6 and 0.8.

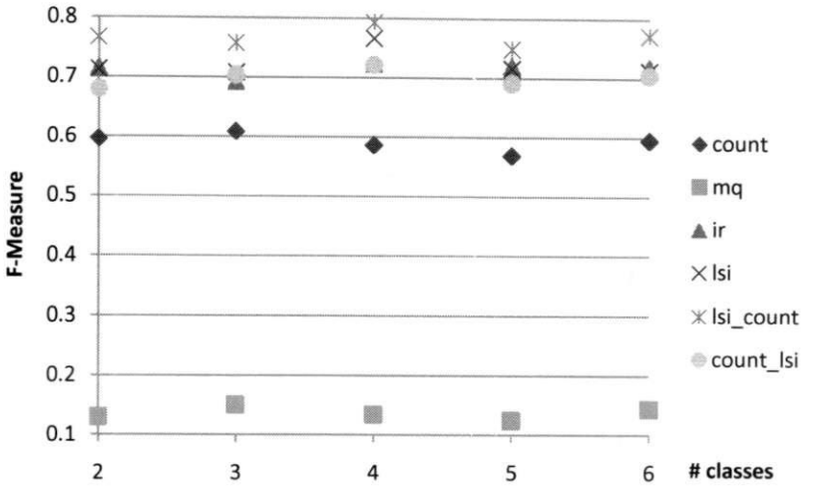


Figure 6.8: F-measure for small changes: OurGrid low-level view

6.5 Discussion

Discriminating power From the results, I observed that mapping techniques show less discriminating power for low-level views ($CaseStudy_2$) than for high-level views ($CaseStudy_1$), although values of *F-measure* between 0.5 and 0.75 are still achieved for low-level views by the automated techniques. On the one hand, such results might seem strange, since one would probably expect that design concepts closer to source code should result in better mapping accuracy. On the other hand, low-level views usually add more categories of concepts to be recognized, which makes the pattern recognition issue implicit in the mapping problem more complex to be solved.

Qualitative evaluation Besides the case studies, I also conducted a qualitative evaluation of the four attraction functions and their candidate sets. I randomly selected 27 trials from all modeled changes, 3 for each available module view, and thoroughly looked at each generated ranking, candidate set, and classification performed.

Regarding candidate sets, evaluation showed that:

- $countAttract$ usually generates sets of size 1 (most dependencies to only one module), 2 (orphan in the border between two modules) or a large set whose size is the number of modules (happens when there are no dependencies to mapped

entities);

- $MQAttract$ usually produces very large non-discriminating sets for large systems, since, in this context, an orphan mapping changes the modularization quality function very little;
- $IRAttract$ and $LSIAttract$ usually produce candidate sets with size between 1 and 3;
- in general, $LSIAttract$ reduces the noise of $IRAttract$, making one candidate "stand out" from the others.

Regarding the type of classification performed, qualitative data suggests that:

- $countAttract$ better maps orphans in structural decompositions, where they are strongly connected to one module only, but usually misplaces entities that are on module borders (e.g.: facade);
- $MQAttract$ performs better in mapping orphan hubs, i.e., entities with strong dependencies to various modules, where a wrong mapping could strongly increase inter-module coupling;
- $IRAttract$ and $LSIAttract$ usually adequately map orphans from libraries or general modules, orphans bordering two modules, and orphans in functional decompositions , but do not perform so well as $CountAttract$ for strongly connected orphans, and make some erroneous classifications for modules that partially share a common vocabulary but have no relation at all, neither functional nor structural.

Decision threshold The mapping algorithm influences the results of each function. The threshold chosen for creating candidate sets is variable and depends on an average computed over a small number of entities. This limited average may negatively influence the classification decision. An adequate value for such threshold remains an open issue.

Feasibility Executed case studies strongly suggest the feasibility of the automated and semi-automated mapping in evolutionary reflexion models, since figures show

that it is possible to correctly map around four out of each five changes. These results suggest that the overhead for the software developer in an evolutionary setting will be reduced. However, special care must be taken with inappropriate mappings and unmapped orphans. Adequate tool support is needed to review and correct wrong mappings and to support semi-automated mappings by means of a recommendation system based on an attraction ranking.

6.6 Summary

This chapter contributes an automated mapping technique for evolutionary reflexion models based on information retrieval (IR). The technique captures a more semantically-oriented form of cohesion in design model entities, a concept that is lacking in other automated mapping techniques. Another contribution is the combination of different mapping techniques based on both IR and structural dependencies, which generally increases mapping recall, while keeping precision figures. Finally, a last contribution is an empirical evaluation of mapping techniques in the context of incremental development by means of two case studies.

Results show that automated mapping performs best when information retrieval is used in combination with structural dependencies in a two-step mapping technique. Different combinations of the mapping functions could have been tried in order to take advantage of the best of each mapping function. The approach considered provides a good starting point that actually produced better results than when the functions were used in isolation.

A variety of cases, scenarios, systems and views was used in this work. Such variety resulted in different performance numbers for each mapping technique, suggesting that no technique is capable of outperforming the others in all possible situations. Views with structural decompositions seem to be better mapped with structural-based techniques, while views describing functional decompositions seem to better mapped by IR-based techniques. Nonetheless, that remains an speculation to be further researched, as case studies cannot be generalized to every possible context.

This study tried to answer the question of whether additional information from source code can be used to improve automated mappings produced by techniques based on structural dependencies. Results are positive and show that using information from software vocabulary in combination with structural dependencies produces mappings with high figures of both precision and recall. High values of *F-measure* were obtained in both singleton, small and large changes, showing the feasibility of an automated mapping technique. On the other hand, results were better for mappings with coarser-grained models than with finer-grained models.

In terms of the main research question, of reducing the manual effort to apply the reflexion model technique in the context of software evolution, this chapter gives empirical evidence that suggests the feasibility of this reduction. Although I do not quantify such reduction in this chapter, it is clear that incrementally automating the mapping step frees the developers from fixing mappings at each conformance check, which they would have to do in order to apply the RM technique in an evolving context. Furthermore, even if the developers cannot count with a perfect mapping technique, it is possible, with the high values of *F-measure* obtained, to use the provided mapping technique as a sound advice to developers in a semi-automated mapping procedure.

An additional discussion on the use of mapping techniques in the context of the ERM process is presented in Chapter 8.

Chapter 7

Prioritizing Warnings Using Software History

PRIORIZAÇÃO DE AVISOS A PARTIR DO HISTÓRICO DO SOFTWARE

Neste capítulo, é realizado um estudo de fatores de potencial influência na relevância de violações arquiteturais. Em seguida, são apresentadas técnicas de priorização e de filtragem de violações, ambas baseadas no histórico do software, assim como uma avaliação experimental destas técnicas.

In this chapter, I perform a study of factors of likely influence in the relevance of architectural violations. Then, I present techniques for prioritizing and filtering violations, both based on software history. Finally, I perform an empirical evaluation of these techniques.

7.1 Introduction

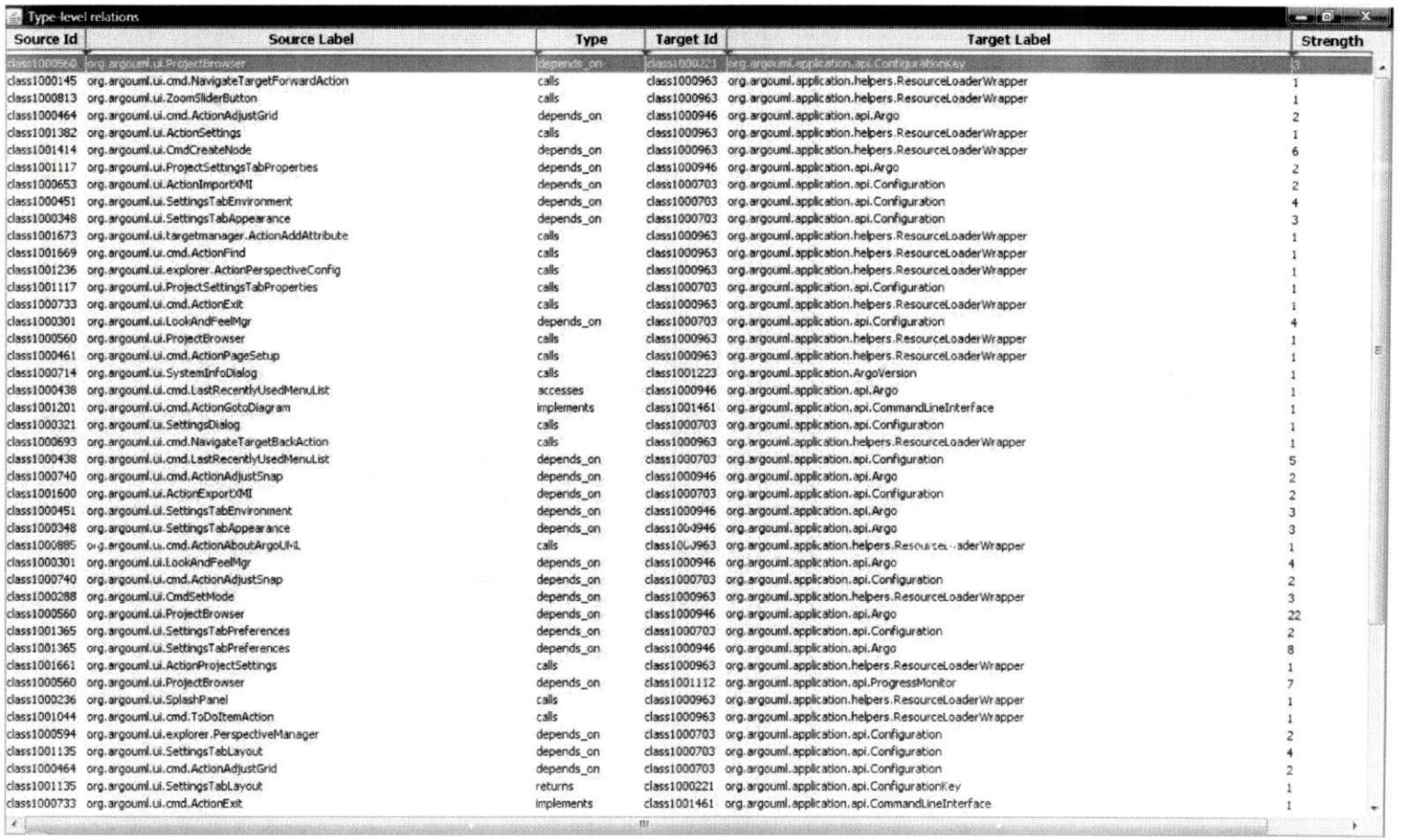
The reflexion model (RM) technique affords the production of detailed violation lists of where source code does not conform to the architecture [Knodel and Popescu 2007]. When the RM technique is applied to a moderate or large system, the number of divergences reported as violations can amount to hundreds of warnings per check [Terra and

Valente 2009; Feilkas et al. 2009]. Figure 7.1 shows an example of a list of violations between two modules in *ArgoUML*, a system later described in this chapter. The list focuses on the types (either classes or interfaces) in the user interface (*ui*) module that depend (but should not) on types in the *application* module. This list amounts to more than one hundred violations, and these are just between these two modules. Had the list been shown for all modules and at a finer-grained level (e.g., methods and fields), it would amount to several hundreds of violations.

The degree of relevance of these violations to a developer varies. Some divergences may never be solved because the violations may be seen as exceptions to a general rule. Others may need to be attended to in a short period of time to prevent erosion of important architectural features. When the list of violations is long, it can be difficult for a developer to find the violations that really matter. To help a developer cope with the long list of violations typically produced, I propose to prioritize the violations using information about the software, its evolution and the people who developed it.

To determine which factors might be of use in automatically prioritizing violations, I ran a study on four open source systems in which I investigated which of these factors (Section 7.2.1) correlated to which architectural violations were solved during the development of the system (Section 7.3). Through this study, I found that measures based on violation duration, with correlation values between 37 and 71% for three systems, violation co-location, between 20 and 42% for four systems, and, in a minor scale, the developer's degree-of-authorship, with values between 14 and 28% for three systems, play an important role in predicting violation relevance.

I then considered whether the influential factors determined in this study could be used with a machine learning approach to assign priorities to violations. I evaluated this approach with an experiment with four different systems. Precision of a top-K ranking varied from 62 to 99%, and the improvement, when compared to the baseline, ranged from 57 to 214%, which suggests the feasibility of an automated approach to prioritizing violation warnings in architecture checkers.



Type-level relations

Source Id	Source Label	Type	Target Id	Target Label	Strength
class1000560	org.argouml.ui.ProjectBrowser	depends_on	class1000221	org.argouml.application.api.ConfigurationKey	3
class1000145	org.argouml.ui.cmd.NavigateTargetForwardAction	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000813	org.argouml.ui.ZoomSliderButton	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000464	org.argouml.ui.cmd.ActionAdjustGrid	depends_on	class1000946	org.argouml.application.api.Argo	2
class1001382	org.argouml.ui.ActionSettings	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1001414	org.argouml.ui.CmdCreateNode	depends_on	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	6
class1001117	org.argouml.ui.ProjectSettingsTabProperties	depends_on	class1000946	org.argouml.application.api.Argo	2
class1000653	org.argouml.ui.ActionImportXMI	depends_on	class1000703	org.argouml.application.api.Configuration	2
class1000451	org.argouml.ui.SettingsTabEnvironment	depends_on	class1000703	org.argouml.application.api.Configuration	4
class1000348	org.argouml.ui.SettingsTabAppearance	depends_on	class1000703	org.argouml.application.api.Configuration	3
class1001673	org.argouml.ui.targetmanager.ActionAddAttribute	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1001669	org.argouml.ui.cmd.ActionFind	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1001236	org.argouml.ui.explorer.ActionPerspectiveConfig	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1001117	org.argouml.ui.ProjectSettingsTabProperties	calls	class1000703	org.argouml.application.api.Configuration	1
class1000733	org.argouml.ui.cmd.ActionExit	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000301	org.argouml.ui.LookAndFeelMgr	depends_on	class1000703	org.argouml.application.api.Configuration	4
class1000560	org.argouml.ui.ProjectBrowser	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000461	org.argouml.ui.cmd.ActionPageSetup	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000714	org.argouml.ui.SystemInfoDialog	calls	class1001223	org.argouml.application.ArgoVersion	1
class1000438	org.argouml.ui.cmd.LastRecentlyUsedMenuList	accesses	class1000946	org.argouml.application.api.Argo	1
class1001201	org.argouml.ui.cmd.ActionGotoDiagram	implements	class1001461	org.argouml.application.api.CommandLineInterface	1
class1000321	org.argouml.ui.SettingsDialog	calls	class1000703	org.argouml.application.api.Configuration	1
class1000693	org.argouml.ui.cmd.NavigateTargetBackAction	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000438	org.argouml.ui.cmd.LastRecentlyUsedMenuList	depends_on	class1000703	org.argouml.application.api.Configuration	5
class1000740	org.argouml.ui.cmd.ActionAdjustSnap	depends_on	class1000946	org.argouml.application.api.Argo	2
class1001600	org.argouml.ui.ActionExportXMI	depends_on	class1000703	org.argouml.application.api.Configuration	2
class1000451	org.argouml.ui.SettingsTabEnvironment	depends_on	class1000946	org.argouml.application.api.Argo	3
class1000348	org.argouml.ui.SettingsTabAppearance	depends_on	class1000946	org.argouml.application.api.Argo	3
class1000885	org.argouml.ui.cmd.ActionAboutArgoUML	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000301	org.argouml.ui.LookAndFeelMgr	depends_on	class1000946	org.argouml.application.api.Argo	4
class1000740	org.argouml.ui.cmd.ActionAdjustSnap	depends_on	class1000703	org.argouml.application.api.Configuration	2
class1000288	org.argouml.ui.CmdSetMode	depends_on	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	3
class1000560	org.argouml.ui.ProjectBrowser	depends_on	class1000946	org.argouml.application.api.Argo	22
class1001365	org.argouml.ui.SettingsTabPreferences	depends_on	class1000703	org.argouml.application.api.Configuration	2
class1001365	org.argouml.ui.SettingsTabPreferences	depends_on	class1000946	org.argouml.application.api.Argo	8
class1001661	org.argouml.ui.ActionProjectSettings	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000560	org.argouml.ui.ProjectBrowser	depends_on	class1001112	org.argouml.application.api.ProgressMonitor	7
class1000236	org.argouml.ui.SplashPanel	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1001044	org.argouml.ui.cmd.ToDoItemAction	calls	class1000963	org.argouml.application.helpers.ResourceLoaderWrapper	1
class1000594	org.argouml.ui.explorer.PerspectiveManager	depends_on	class1000703	org.argouml.application.api.Configuration	2
class1001135	org.argouml.ui.SettingsTabLayout	depends_on	class1000703	org.argouml.application.api.Configuration	4
class1000464	org.argouml.ui.cmd.ActionAdjustGrid	depends_on	class1000703	org.argouml.application.api.Configuration	2
class1001135	org.argouml.ui.SettingsTabLayout	returns	class1000221	org.argouml.application.api.ConfigurationKey	1
class1000733	org.argouml.ui.cmd.ActionExit	implements	class1001461	org.argouml.application.api.CommandLineInterface	1

Figure 7.1: Example of violation list.

Finally, I adapted the machine learning system to work as a classifier so that violations likely to be irrelevant could be filtered out. The classifier reduced the amount of violation information to be investigated between 51.7 to 91.2% for the four studied systems, while still keeping good filtering quality, in terms of specificity, sensitivity and precision.

The findings suggest an inherent difference from previous work on prioritizing bug predicting approaches. While unstable source code is usually accompanied by increased bug density [Nagappan and Ball 2005], the same can not be said regarding architectural violations. And, while co-located bug warnings usually point to false positives in bug finding tools [Kremenek and Engler 2003], increasing co-located architectural warnings typically increases their relevance.

7.2 Driving Hypotheses

Violations about conformance to a stated architecture can represent unwanted coupling between modules from an architectural point of view. To perform this study, I postulated five hypotheses based on the literature about software development.

H1: Developer Centrality Developers new to a system may not appropriately understand the software architecture and as a result, may introduce undesired coupling. One indirect measure of a developer's experience with the code for a system is how much code he or she has shared with other developers. The more code shared, the more central a developer is to the project [Bird et al. 2006]. I hypothesize that a low value of developer centrality influences the addition of relevant violations in the source code.

H2: Degree-of-authorship A lack of appropriate knowledge of the source code, APIs and module interfaces by a developer may also result in the introduction of violations. The degree-of-authorship is a measure that quantifies the ebb and flow of source code knowledge by a software developer as a software evolves [Fritz et al.

2010]. I hypothesize that the lower the degree-of-authorship, the higher the number of violations that indicate unwanted coupling is expected.

H3: Code churn Source code instabilities usually result in software bugs [Nagappan and Ball 2005]. Similarly, unstable code may also affect architectural stability, producing architectural violations. I hypothesize that the higher the churn of the code, the more architectural violations that will result.

H4: Violation co-location A high frequency of co-located bug warnings has been used to predict false positives [Kremenek and Engler 2003]. Similarly, I hypothesize that violations happening in isolation are more important than violations strongly co-located with others. Co-location, in this case, means that a source code violation happens inside the same container (e.g., either a file, an object-oriented type or an architectural module) as other source code violations. The premise here is that a greater number of co-located violations means either an architectural rule that is thoroughly disregarded or a change in architectural decisions not reflected in the high-level model, and, as such, these violations are less important.

H5: Violation duration Previous research on bug warnings showed that bug warnings fixed quickly were important while warnings that were not removed for a long time were neglectable [Kim and Ernst 2007a]. The temporal history of an architectural violation may also reflect its importance. I believe that harsh static architectural violations are solved fast so that non-functional requirements such as maintainability are not undermined. And minor violations survive longer, provided they do not strongly affect such requirements. Thus, I hypothesize that short-term violations are more relevant than long-term ones.

7.2.1 Definitions

To support the investigation of these hypotheses on open source systems, several terms need to be defined. As described in the next section, this investigation focuses on Java

systems and thus the definitions are described in terms of Java. These definitions may easily be adapted to the context of other programming languages.

Version: A software version is the source code and documentation of a software extracted from a software repository from a certain moment in time. The superscript i will be used throughout this text to imply a version extracted at a discrete moment in time. The time difference between a given version i and its subsequent version $i + 1$ is given in time units, e.g., hours, days, weeks or months.

Code element: A code element c^i of a software version i is an entity that is found in the source code of version i. In Java, a code element can be a field, a routine (method, constructor or static initializer) or a basic type (abstract class, concrete class, interface or enumeration).

Type: A type t^i of a software version i is an outer container of code elements in version i. In Java, types enclose fields, routines or basic inner types. In the following, a type will be regarded simply as a set of code elements.

Module: A module m^i of a software version i is an architectural element of a module view in the documentation of version i. A module m^i conceptually encloses a set of types. Thus, a module will simply be considered a set of types.

Code relation: A code relation $cr^i_{c_x,c_y}$ of a software version i is a dependency existing in the source code of version i from code element c^i_x to code element c^i_y. Code relations in Java can be any one of the following relation types: method call, field access, parameter received, parameter returned, type thrown, type catched, is-a, contains, extends or implements.

Code-level violation: A code-level violation is an unexpected dependency (namely, a divergence in reflexion model terminology) between two code elements. It is uniquely defined by two participating source code entities that cause the violation and the violation type (e.g., field access, method call). More specifically, a code-level violation $v^i_{c_x,c_y,rt}$ is a set of one or more code relations $cr^i_{c_x,c_y}$ of the same relation type rt that should not exist in the implementation.

For each code-level violation, I need to define the concepts of violating file and violating developer. Although an object-oriented type is more important in our context of architectural violations, I also formally define files and violating files because most operations over source code repositories (e.g., diffs, commits) are performed over files.

File: A file f^i of a software version i is defined here as a container of types in version i. In Java, each file encloses one or more types.

Violating file: The violating file f_{v^i} is the file f^i involved in a code-level violation $v^i_{c_x,c_y,rt}$ (or simply v^i). Although a violation involves both a source file and a target file, the term violating file refers to the source file, since the unwanted dependency is created in the source file.

Violating developer: The violating developer d_{v^i} is the developer d that originally caused the code-level violation v^i, i.e., the developer that committed a file change, which, in turn, caused the appearance of a violation.

Given the definitions above, I can now define each of the investigated factors possibly involved in a code-level violation.

Developer centrality: The centrality $C_{d_{v^i}}$ of a violating developer d_{v^i} is a parameter extracted from the social network of developers that deliver changes to the software repository. Nodes in this network are made of developers, while edges between developers arise when they deliver changes to the same files during a pre-defined time interval. For different common files, a different edge is added between two developers, resulting in a multigraph. I define violating developer centrality as the degree of his/her node, i.e., the sum of the number of edges between the violating developer and all the other developers in the network [Lopez-Fernandez et al. 2004].

Degree-of-authorship: The degree-of-authorship $DoA_{f_{v^i},d_{v^i}}$ of a violating file f_{v^i} by a violating developer d_{v^i} is defined by equation 7.1 [Fritz et al. 2010]. FA, the *first authorship*, is a binary value that is one when the violating developer created the violating file, and zero otherwise, DL is the number of *deliveries* of

the violating file by the violating developer, AC is the number of *acceptances* of other developers' commits by the violating developer, and the constants are the same as in the original work that defined degree-of-authorship. Both DL and AC are considered over a time window of previous committed revisions. This time window can be fixed (e.g., three months as it was fixed in this work) or variable (e.g., the whole period since the first revision up to the present version).

$$DoA = 1.098 \cdot FA + 0.164 \cdot DL - 0.321 \cdot \ln(1 + AC) \tag{7.1}$$

Code churn: The code churn $CC_{f_{v^i}}$ of a violating file is defined as the relative amount of change between version v^i and version v^{i-1} in the source code of a file [Nagappan and Ball 2005]. In this work, I compute it as the number of added and deleted lines of code over the number of lines of code of the file.

Violation co-location: Module co-location MCL_{m_j,m_k,v^i} of a code-level violation v^i is the number of code-level violations that happen between its source container module m_j and its target container module m_k. Similarly, type co-location TCL_{t_l,t_m,v^i} of a code-level violation v^i is the number of code-level violations that happen between its source container type t_l and its target container type t_m.

Violation duration: The duration dur_{v^i} of a violation v^i is the time interval between the appearance of the violation and the analysis time. More specifically, given that a violation lasts for k software versions $i-(k-1), \cdots, i-1, i$, and each version is delayed to its consecutive version by fixed tu time units, $dur_{v^i} = tu \cdot (k-1)$.

7.3 Investigation of Factors

In this section, I describe the research question, experimental design, choice of subject systems, measures for evaluation, and experiment results.

7.3.1 Research Question

The research question I wanted to answer in this investigation was: *how do the factors defined in 7.2.1, isolated or jointly, correlate to the relevance of code-level violations in reflexion models?* Answering this question helps to quantify the influence of each factor in predicting the relevance of architectural violations during software evolution.

7.3.2 Experimental Design

In order to analyze architectural violations, some choices and assumptions about experimental design, software versions, evolution period, correlation, and architectural violations must be laid out.

First of all, the factors described in 7.2.1 are the independent variables, while the relevance of architectural violations is the dependent variable. I am interested in how violations change over time. As a result, I use an experimental design based on mining repositories that store information about a project's software development history.

The granularity of changes in each software revision may be too small for architecture checks. Instead of looking at each revision, I decided instead to focus on weekly versions. To account for a large portion of a project's evolution, I take into account a large development timeline, such as the time between major releases. Since most open source systems release versions between every three or six months, I choose six months as the minimum amount of time to accumulate historical information.

Correlation between independent variables and a dependent variable can usually be derived by means of fitting the data to a multidimensional function and, later, computing the correlation coefficient between the data and the function estimates. Various machine learning techniques are available for data fitting. Support vector machine (SVM) is a popular machine learning technique that has previously given good results in software engineering research [Anvik et al. 2006]. In this work, I use SVMs to learn which factors affect the removal of architectural violations.

Starting with a major release, I use a three-month time slot for machine learning training and an additional three-month slot for testing the correlation between factors and

violation relevance. I also add a past time slot to accumulate data on authorship and collaboration, and a future time slot to estimate violation relevance (see subsection 7.3.2), both slots of three-month duration. Figure 7.2 shows the timeline for this correlation study.

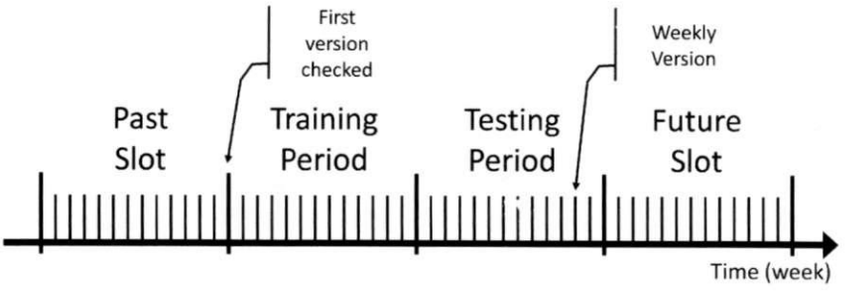


Figure 7.2: Timeline for experimental design

Degree-of-authorship and developer centrality are computed from accumulated data in a three-month period, while code churn is computed between versions one week apart.

Determining Violation Relevance

I need an oracle to assign values of violation relevance to evaluate the investigation of the research question. Doing such is complex since the assignment depends on subjective aspects from the software design. One might think of using a binary variable (either a violation is relevant or not), an ordinal variable (discrete relevance levels) or even a real variable (real values computed according to some measure). Whatever choice is made, an additional problem arises in how to automate the assignment of relevance values for each violation. In practice, the availability of software repository history can help to determine these values. I solve this problem by using an intuition similar to the one by Kim and Ernst in a study of static analysis tools [Kim and Ernst 2007b]. Their approach consists of assigning as relevant all bug warnings later solved by fix-changes. Adopting a similar approach in this work, I assign as relevant the static architectural violations that are later solved in the software history, and as irrelevant otherwise. Obviously, such approach focuses only on architectural violations that are usually solved by developers. I argue that these violations are the most important,

given that software developers perform periodic perfective maintenance, such as in code refactorings.

7.3.3 Choice of Subject Systems

Some of the requirements for the experiments' subjects were driven by the problem itself. Others were constraints of the technologies used. I chose systems from medium to large size, since architecture module views become more important when system size grows. I also chose systems with adequate commit policies, for the granularity of independent variables is important for meaningful cause-effect analysis. Commits should happen on a daily or short-term basis to allow the generation of meaningful weekly data. Software versions should be available from software repositories, either from *CVS*[1], *SVN*[2] or *GIT*[3] version control systems. In addition, there should be an adequate time frame for extracting empirical data, for some independent variables require accumulating past data, while estimating the dependent variable requires the knowledge of future violation data. Moreover, source code had to be in Java and software versions should be compilable, because *Design Wizard*, the chosen design extraction tool, extracts basic design facts from the bytecodes of Java systems [Brunet et al. 2011]. Furthermore, high-level models should also be available, since reflexion models require the definition of a high-level model to check conformance of the source code against it. Finally, I chose systems representative of real open source systems, extracted from different repositories such as SourceForge, Apache Software Foundation or other repositories.

Table 7.1 shows the chosen systems along with some of their features. *SweetHome3D* is an interior design application that allows placing furniture in 2D plants with 3D previews.[4] *Ant* is a popular Java-based build automation tool.[5] *Lucene* is a text search engine library written entirely in Java.[6] And *ArgoUML* is an open source UML mod-

[1] www.nongnu.org/cvs
[2] subversion.apache.org
[3] git-scm.com
[4] www.sweethome3d.com
[5] ant.apache.org
[6] lucene.apache.org

eling tool.[7] High-level models for these systems were extracted from system documentation and their size is also shown in Table 7.1. *SweetHome3D* had design tests in the *JDepend* tool with packages as modules and assertions as the allowed dependencies. *Ant* had a module view based on packages in the *Lattix LDM* tool. *Lucene* had a layered view diagram and I performed the mapping myself, using the package names as the basis for module attribution. Finally, *ArgoUML* had the most detailed design documentation: a set of module views and the packages that made up each module. The high-level models represent relevant features of the systems, but they are not intended to be complete. Thus, some features can be missing in the models. Models and mapping for the four systems are available in Appendix E.

Table 7.1: Subject systems

System	Version Timeframe	Size (KLOC)	# Monthly Committers	# Monthly Commits	High-Level Models	
	first / last	min-max	min-max	min-max	# Modules	# Edges
SweetHome3D	2009-03-08 / 2010-02-28	75-96	1	6-99	9	29
Ant	2006-10-29 / 2007-10-21	232-239	4-9	22-164	16	92
Lucene	2010-03-21 / 2011-03-13	247-336	7-9	58-173	7	16
ArgoUML	2006-11-19 / 2007-11-11	397-813	5-15	120-286	19	79

In Table 7.2, I describe some statistics for the architectural violations for each chosen system, for a testing period of three months. Relevant violations were found using the heuristics explained in 7.3.2. Weekly violations are also shown, amounting to hundreds for each system. These systems are good candidates for the experiments for two reasons: first, because there is a large percentage of positives for all systems (at least 27.5%), allowing for a good fitting procedure; second, because there is a large number of weekly violations, amounting to hundreds for each system, what results in large training and testing sets for the SVMs.

Table 7.2: Violations in the testing set

System	Violations in the Testing Period			# of Weekly Violations		
	Total	# Relevant	% Relevant	Minimum	Average	Maximum
SweetHome3D	5224	3164	60.6	390	401.8	414
Ant	8657	2383	27.5	623	665.9	688
Lucene	3192	885	27.7	233	245.5	268
ArgoUML	7692	2419	31.4	563	591.7	623

[7] `argouml.tigris.org`

7.3.4 Experimental Procedures and Evaluation

To better understand the relation between independent variables and violation relevance, I devised a simple scheme of data fitting, using SVMs trained with a three-month training set of violations. Then I computed regression with independent variables over a three-month violation testing set.

First, I computed regression with each of the factors individually. I also performed multiple regression with one of the factors, namely co-location, that was made of two components. Finally, I computed multiple regression with the factors altogether.

An informal measure of the effect of the independent variables on relevance would be the correlation between the variables. The higher the correlation, the higher the effect of the independent variable(s) on the dependent variable.

Correlation between the data values and the fit function values may be computed using the Pearson's correlation coefficient (r), which provides a measure of how well a dependent variable can be predicted from the scores of a set of independent variables. I used Cohen's values to report low ($r = 0.1$), medium ($r = 0.3$) and high ($r = 0.5$) correlation [Cohen 1988].

7.3.5 Results

Table 7.3 shows the results of using SVMs for data fitting. Values for Pearson's correlation coefficient are shown for individual factors, module and type co-location combined, and the factors altogether.

Table 7.3: Pearson's correlation coefficient between studied factors and relevance

	Systems			
Factor	SweetHome3D	Ant	Lucene	ArgoUML
Developer Centrality	0.0000	0.0000	0.3130	0.0000
Degree-of-Authorship	0.2806	-0.0216	0.1357	0.1697
Code Churn	-0.0767	0.0882	-0.0211	0.0080
Type Co-Location	0.2647	0.1918	0.1398	0.2557
Module Co-Location	0.1348	0.2046	0.4467	0.1315
Module & Type Co-Location	0.3458	0.2017	0.4201	0.2086
Violation Duration	0.7070	0.0000	0.3728	0.6277
All Factors	**0.7070**	**0.2030**	**0.4705**	**0.6331**

From the results, we noticed that the highest correlation coefficient was found when all the factors were used in the fitting, with its value amounting to between 20 and 71%. In isolation, the factors that best correlated were violation duration for *SweetHome3D* (71%) and *ArgoUML* (63%), and violation module co-location for *Ant* (20%) and *Lucene* (45%).

Looking at each factor for the four systems as a whole, the following results were found. The measure of developer centrality we used showed zero correlation with relevance but for the case of *Lucene*, which suggests that either the measure we used is not adequate or the developer centrality alone does not influence the violations. The degree-of-authorship showed weak correlation for three out of four systems. Code churn showed very weak correlation with the dependent variable, which suggests that this factor does not play an important role on architectural violations. Type co-location showed weak to medium correlation for all systems, while module co-location showed weak to medium correlation for three systems and medium to high correlation for a fourth system. Combining both co-location measures either gave similar correlation values to the best isolated measure or improved the results, as in the case of *SweetHome3D*. Finally, violation duration showed zero correlation with one system, medium correlation with another system and high correlation with the other two systems.

Thus, results suggest that three out of the five studied factors, namely violation co-location (both module and type), violation duration, and degree-of-authorship, seem to play an important role in predicting the relevance of architectural violations. Nonetheless, the relative contribution of each factor to the correlation with violation relevance depends on each system, and no factor can be taken as the most influential for every case. Last, taking all the factors together does not improve the correlation results when compared with the best of the isolated factors, which suggests that the interaction between independent variables does not influence the results.

7.4 A Recommender for Prioritizing Violations

The predictive power of some of the factors investigated in Section 7.3 led me to conceive a recommender system for prioritizing static architecture violations. More specifically, I devised a system that produces a top-K priority ranking of violations so that developer's actions regarding violated static architectural rules are focused to those most needing attention. For this work, I set the value of K as 10, because it is a small number of elements that developers can deal with, only a bit larger than the capacity of a person's short-term memory, of 7 ± 2 elements [Miller 1956].

To evaluate this recommender, I kept the same experimental design, assumptions, and subject systems from the experiment to investigate the factors in 7.3, but with a different research question, and different experimental procedures and evaluation, as stated below.

7.4.1 Research Question

The second research question was: *is it feasible to direct developer's focus in architectural checking results to a top-K rank of the most relevant violations?* If one or more of the investigated factors correlates to violation relevance, one can design a recommender system that uses the historical information from such factors to prioritize present violations to be analyzed.

7.4.2 Experimental Procedures and Evaluation

I first developed a recommender system to be used in the experiments. More specifically, a system that lists a top-K priority ranking of violations to direct software developer's actions regarding violated static architecture rules. To do such, I trained an SVM with a three-month training set with all the independent variables. Then I produced a top-K ranking, with K set as 10, of weekly violations using the SVM as a regression machine and taking the regression value as the ranking priority.

A success measure of such a system would be that a large proportion of the top-K

shown violations be, in fact, relevant violations. That is, the ranking must be very precise to direct software developers to actual static architecture issues, issues that really should be analyzed by developers to maintain design conformance. Thus, precision is the measure that matters in this scenario.

Formally, precision is the positive predicted value, i.e., the ratio between violations correctly classified as relevant and the number of violations found relevant by the technique. It is described by equation 7.2, where TP stands for true positives, and FP, for false positives.

$$Precision = \frac{TP}{TP + FP} \tag{7.2}$$

7.4.3 Results

Table 7.4 shows the results of the ranking based on an SVM that takes all factors into account. Regression with the SVM was computed and only the top-K violations were considered. For this work, I set K as 10. Average precision of the top-K ranking is shown in the third column, and its results are compared against a baseline of selecting K random violations from the full set of weekly violations (second column). The improvement given by the ranking, when compared to the baseline, is shown in the fourth column.

Table 7.4: Precision improvement with top-K ranking ($K = 10$)

System	Avg. Precision selecting K Random Violations (baseline) (%)	Avg. Precision in Top-K Ranking (%)	Improvement from baseline (%)
SweetHome3D	60.4	94.6	56.5
Ant	25.1	62.3	148.4
Lucene	27.7	78.5	184.1
ArgoUML	31.4	98.5	213.6

Results show a high precision level for the four subject systems, ranging from 62.3 to 98.5%. Moreover, the improvement against the baseline varies from 56.5 to 213.6%.

As fixing violations can require higher-level architectural or refactoring work, I believe

conformance checks are more likely to be performed after longer development periods (e.g., weekly or monthly). To simulate this scenario, I consider the performance of the recommender when it is applied weekly. Table 7.5 shows precision statistics per week, in terms of minimum, average and maximum values.

Table 7.5: Weekly precision results for top-K ranking ($K = 10$)

	Precision (%)		
System	**Minimum**	**Average**	**Maximum**
SweetHome3D	40	94.6	100
Ant	0	62.3	90
Lucene	60	78.5	90
ArgoUML	90	98.5	100

Results show some dispersion for the weekly ranking precision, especially for *SweetHome3D* and *Ant*, although the average precision values are very high, ranging from 62.3 to 98.5% for the four studied systems. These precision values suggest the value of warning prioritizing, which can bring a productivity gain in the analysis of the results of conformance checks.

7.5 Filtering Irrelevant Violations

Support vector machines are commonly used as classifiers. An SVM used to classify violations into relevant or irrelevant based on software history information is an interesting scenario to be analyzed. Such a system can be used in order to filter out irrelevant violations, reducing the amount of data to be analyzed by a software team.

Keeping the same experimental design, assumptions, and subject systems from the investigation of factors in 7.3, but, once again, with a different research question, different experimental procedures and evaluation, I ran an evaluation of a classifier of architectural violations.

7.5.1 Research Question

The third research question in this study was: *is it possible to accurately filter out irrelevant violations from architectural checking results from regular architecture checks?*

If so, it is possible to reduce the amount of data to be analyzed by software developers in architecture checking tools, thus, facilitating tool adoption.

7.5.2 Experimental Procedures and Evaluation

I first developed a classifier of violations to be used in the experiments. More specifically, a system that filters out violations deemed irrelevant. To do such, I trained an SVM with a three-month training set with all the independent variables. Then I classified a three-month testing set of weekly violations, using the SVM as a classifier.

Statistical measures are normally used to evaluate classifiers. The importance of each measure, however, depends on the context the classifier is being used. Below, I lay out the important measures for the context of this work. In the following, TP stands for true positives, FP, for false positives, TN, for true negatives, and FN, for false negatives.

Filtering is similar to a screening medical procedure. One wants to discard violations that are in fact irrelevant, and this goal is more important than keeping a very precise set of relevant violations. Thus, the measure of *specificity* was the most important measure in this scenario, since I wanted to filter out violations that were really true negatives.

Specificity: Specificity is the true negative rate, i.e., the ratio between violations correctly classified as irrelevant and the number of irrelevant violations in the population. It is described by equation 7.3.

$$Specificity = \frac{TN}{TN + FP} \tag{7.3}$$

Sensitivity or *recall* is another measure used in classifiers that is complementary to specificity. It suffices to say that very low sensitivity values would make a very specific classifier useless, because most violations would be tested as negatives.

Sensitivity: Sensitivity is the true positive rate, i.e., the ratio between violations correctly classified as relevant and the number of relevant violations in the population. It is described by equation 7.4.

$$Sensitivity = \frac{TP}{TP + FN} \tag{7.4}$$

As in the recommender described in section 7.4, most violations found by the classifier should actually be relevant. Although filtering is the main concern here, the classifier must also be precise to direct developers to actual static architecture issues. Thus, precision is a complementary measure in the filtering scenario, and its definition is restated here.

Precision: Precision is the positive predicted value, i.e., the ratio between violations correctly classified as relevant and the number of violations found relevant by the technique. It is described by equation 7.5.

$$Precision = \frac{TP}{TP + FP} \tag{7.5}$$

7.5.3 Results

Filtering can be evaluated first by the amount of information reduction that it achieves. Table 7.6 shows the amount of filtered violations that the developer will be able to ignore for a classifier based on an SVM that takes all independent variables into account. Results are compared to the relative number of irrelevant violations found in each system. Obviously, nonetheless, the filtering quality is even more important than the amount of information reduction it brings. Table 7.7 shows the filtering quality, in terms of specificity, sensitivity and precision for the same classifier. The very high values of specificity, with ratios larger than or equal to 87.5%, suggests a small number of false positives, and good screening power for the filter. The remaining violations to be analyzed after filtering have good values of precision, with all systems giving values of at least 62.4%; and the completeness of the positive detection is good for three systems (sensitivity between 59.8 and 80.9%), and bad for *Ant*, with sensitivity of 20%.

7.6 Discussion

Table 7.6: Tested and actual irrelevant violations.

System	Filtered Violations (%)	Irrelevant Violations (%)
SweetHome3D	51.7	39.4
Ant	91.2	72.5
Lucene	74.4	72.3
ArgoUML	66.4	68.6

Table 7.7: Specificity, sensitivity and precision results for filtering.

System	Specificity (%)	Sensitivity (%)	Precision (%)
SweetHome3D	96.4	77.4	97.1
Ant	95.4	20.0	62.4
Lucene	87.5	59.8	64.7
ArgoUML	88.1	80.9	75.7

Bugs versus Architectural Violations I was surprised to find that some of my hypotheses about the factors that contribute to the importance of architectural violations, that were based on intuition and interpretation of the literature, did not hold in the conducted experiment. The assumptions of H2 on degree-of-authorship and H4 on co-location that were maybe valid for bug warnings did not follow the same pattern for architectural violations. Direct correlation between degree-of-authorship and relevance may suggest that the most important violations are made by people who know more about the code and not the ones who know less. The same is valid for module co-location and type co-location, where positive correlation leads to think that isolated violations are less important than accumulated co-located violations, the opposite of bug warnings. Below, I consider each hypothesis in more detail.

H1: Developer Centrality The measure of developer centrality did not show correlation with violation relevance. I suspect that one cause is that committers are not necessarily the violation authors. The reduced number of committers in the analyzed systems suggests that the information about developer collaboration is somehow lost since one committer might aggregate data from various authors. The popularization of newer version control systems that inform both authors and committers in the commit metadata (e.g.: *GIT*) would help to provide more reliable data for developer centrality.

H2: Degree-of-authorship Direct correlation between degree-of-authorship and relevance suggests that the most important violations are made by people who know more about the code. This may be because even the most central developers sometimes need to disregard architectural rules, even though such violations are later fixed. Another hypothesis is that violations made by less experienced developers could go unnoticed for a longer time. Whatever the cause is, the existence of weak correlation between degree-of-authorship and violation relevance may be a reason for using this variable in a recommender system that ranks violations, given that the cost to compute it is not prohibitive for a weekly task of software quality assurance.

H3: Code churn From the results, I realized that code churn does not seem to play a role in violation relevance as it does in bug warning relevance [Ruthruff et al. 2008]. To explain this, I hypothesize that architectural violations can survive longer than bugs. Unstable code usually produces bugs, but bugs usually need to be fixed fast, thus, generating more churn. Architectural violations, on the other hand, may survive longer, since they worsen the design but do not prevent the system to compile and run correctly.

H4: Violation co-location An important experimental result was that the high unexpected coupling expressed by high module or type co-location revealed an important factor to predict violation relevance, which agrees with the heuristics of increased coupling causing increased design deterioration. That seems to be true at least for the subject systems in the experiments, where I found from weak to medium correlation values.

H5: Violation duration Finally, another important result was the medium to high positive correlation of violation duration with relevance in three out of four systems. This shows that violation correction in time usually follows a pattern, and that pattern deserves a more thorough investigation.

A Simple Recommender The fact that violation duration and co-location were together responsible for most of the correlation in the four studied systems suggests a very simple design for a recommender system for prioritizing architectural violations. This design would take into account only violation co-location and violation duration, which would simplify priority computation, since violations would be the only data to be gathered, discarding the computation of indirect factors like degree-of-authorship, code churn and developer centrality.

Top-*K* Ranking The results of the experimentation with a recommender suggest that violation prioritizing seems to be not only feasible, but also successful. High precision values, between 62 and 99%, strongly suggest that a recommender system would be helpful to direct developer's attention to architectural violations. When ones compares to the baseline of selecting K random violations from the full set of weekly violations, results are clearer, showing an improvement ranging from 57 to 214%. Focusing developer's attention to a much smaller list of violations may thus help to increase adoption of static architecture checking tools, since the time devoted to analyzing violations could be better used, and not wasted in analyzing false positives.

Insertion of prioritized architecture checks in the software development process by means of weekly procedures is lightweight and it can easily be added to the typical weekly cleanup that developers do after adding features to a software system.

Filtering Usually, the amount of irrelevant violations is a large fraction of the total amount of violations found by the architecture checking technique. For the analyzed systems, it ranges between 72.3 to 74.5% for three of them, and it is smaller only for *SweetHome3D*, with 39.4% irrelevant violations. Such numbers offer an opportunity for a filtering procedure that can reduce the amount of violations to be analyzed by the developer. In Table 7.6, one can see that the classifier developed in this study showed an amount of information reduction between 51.7 to 91.2%. The classifier also discarded a small fraction of relevant violations, but kept good filtering quality in general, as can be inferred from the results in Table 7.7 for specificity (between 87.5 and 96.4%), sensitivity (between 59.8 and 80.9% for three out of four systems) and

precision (between 62.4 and 97.1%), with bad results only for sensitivity in one of the systems, *Ant*, with 20%. Thus, I argue that there is value in an automated filtering procedure for violations. If developers want a more complete screening procedure for violations instead of a priority top-K ranking, a classifier may help them to focus on the violations more likely to be relevant, reducing their amount of work in more than half, while keeping acceptable filtering quality.

Validity Evaluation The results cannot be generalized to contexts other than the studied systems. However, I did try to reduce external validity threats by choosing typical systems from different software repositories (e.g., SourceForge, Apache Software Foundation, and Tigris). I also chose systems from medium to large size with existing high-level models, which are the typical candidates for architecture checks. Another issue on generalization is the focus on one architectural checker, reflexion models. I argue that most architecture checkers also output long violation lists, since few high-level architectural rules usually convert to lots of code-level checks, due to the detailed data and control dependencies between elements in source code. Long lists of violations have been reported in case studies of conformance checking [Feilkas et al. 2009; Terra and Valente 2009].

The construct validity is threatened by several assumptions. First, violation relevance is indirectly determined by heuristics based on violation solving, which might not necessarily match the relevance expressed by external input from experts on the software systems. I argue, however, that these heuristics are pragmatic and related to the actions performed by software developers during software evolution. Second, the high-level models and mappings were derived from the analysis of system documentation by the author of this work, and not from the developers themselves. To mitigate this threat, I tried to be as faithful as possible to the existing architectural documentation of these systems. Another issue on the construct is the lack of knowledge of the full software history, which prevents me from assigning some of the violations to their authors. However, I believe that the multivariate machine learning is robust enough to deal with missing values for some of the variables. Finally, violation duration is computed relative to the start of the training period, which masks some of the values for the

duration factor.

Most internal validity threats are tackled by experiment automation. All measures are taken from previous history in software repositories. Developers, at development time, were not aware of these measures, thus, mitigating most internal validity threats.

7.7 Summary

This chapter describes an approach to reduce information overload in static architecture checks. Code-level violations to static architecture rules can amount to hundreds or more for each architecture check. By means of software repository mining experiments, I investigated how five independent variables extracted from software artifacts correlate with the relevance of these violations. Results suggest that violation duration and violation co-location are the most important factors to correlate with relevance, but no factor seems to be the most relevant for every system. Furthermore, fitting data with all five factors does not add much in terms of better correlation when compared to the best variable alone. Finally, degree-of-authorship may also play a role in predicting violation relevance with smaller correlation values, while code churn and developer centrality do not seem to play an important role for most of the studied systems.

Results from the experiments suggest an inherent difference between bugs and architectural violations. Unstable source code usually points to potential bugs [Nagappan and Ball 2005], while the same was not true to architectural violations in our study. Determining relevance of bug warnings also seems to be very different of doing such with architectural violation warnings. While accumulated bug warnings in the same code area usually point to a false positive [Kremenek and Engler 2003], accumulated architectural warnings between two modules pointed to more relevance in this study, suggesting that increased unwanted coupling reveals architectural deterioration.

In this study, I introduced a machine learning system based on support vector machines to recommend the most important violations to be analyzed by software developers. Through the experiments, I found empirical evidence that automated recommendation of violations is not only feasible, but successful. I found that producing a ranking

with the top-K violations directs developers to a very reduced and precise list of violations. In order to achieve more completeness in a violation screening procedure, I also used the machine learning system as a classifier that filters irrelevant violations from the whole violation set. This classifier reduced the amount of work to analyzed violations in more than half for the analyzed systems, while still keeping good filtering quality. Such results are encouraging and might allow for an easier adoption of static architecture checks in the software development process.

This study tried to answer the question of how the outputs of the reflexion model technique can be prioritized in order to reduce information overload to software developers. Results show that measures extracted from software history in fact allow this prioritizing to happen with high precision values. By means of a Top-K prioritizer, architectural violations can be recommended to developers. Thus, their work on checking and fixing architectural violations is directed to more relevant issues.

Referring to the main research question of this work (reducing manual effort when applying the reflexion model technique in a context of evolving, sparsely documented software), this chapter effectively shows the feasibility of such reduction in the step of violation logging. Although I did not quantify the amount of effort reduction, I showed empirical evidence that supports this claim. Focusing on a shorter set of likely relevant architectural violations reduces the manual effort by avoiding wasting time on a larger set of violations that would not matter to software developers.

A discussion on prioritizing architectural warnings in the context of the ERM process is presented in Chapter 8.

Chapter 8

Discussion

DISCUSSÃO

Neste capítulo, é realizada uma discussão dos principais resultados desta tese, bem como de sugestões de trabalhos futuros.

In this chapter, a discussion of the main results of this dissertation is performed, together with suggestions for future work.

8.1 Clustering Algorithms

Analysis of four clustering algorithms in this dissertation showed that they are limited in their capacity of generating fully automated and meaningful software decompositions. In most cases, these algorithms produced very few large clusters and lots of singleton clusters. This result is a drawback for a fully automated architecture recovery process, since decompositions manually made by software developers usually show a different pattern, with lots of medium-sized clusters. The best performing algorithm in terms of non-extremity of cluster distribution, *K-means*, is limited by the need to define the number of clusters as an input parameter. When discovering modular concepts in architecture recovery, often there is no previous knowledge of the number of modules.

On the other hand, it is also important to state the limitation of a measure of non-extremity. Some real architectural modules are in fact extreme. For instance, architectural styles such as a mediator or a façade define modules that are singletons. It seems more accurate to analyze whether clusterings produced by algorithms resemble real architectural modules. In this context, related work has shown empirical evidence that type-level or file-level software networks follow a scale-free, heavy-tailed degree distribution [Myers 2003]. Myers has suggested that the modular design of software systems may be responsible for such distribution. If this hypothesis is further investigated for static architecture views, it may be that they follow a similar distribution.

One important issue was that stable clustering algorithms did not produce meaningful clusters in terms of non-extremity. The only algorithm found stable in this work, based on *edge betweenness*, did not produce meaningful clusterings at all. The other three had low values of stability. In addition, these results agree with the results of Wu and colleagues for other clustering algorithms [Wu et al. 2005]. Such results show an important drawback of clustering algorithms for architecture recovery in the context of software evolution. Applying a clustering algorithm from scratch to recover a static software architecture view at different time instants does not produce similar results. Thus, a different approach to architecture recovery must be pursued in the context of evolution. I believe that using knowledge from a previous architecture recovery step is the solution, in what might be called a process of incremental clustering. This belief is supported by the results of the incremental mapping evaluation shown in Chapter 6, further discussed below. Thus, clustering algorithms may be just a first step in discovering modular concepts. This step must necessarily be followed by a validation phase of manual recovery by a software designer. Only then, with a validated module view, one can count on automated incremental clustering techniques to provide stable clusters with architectural meaning.

For the first step described above, either *modularization quality* clustering or *design structure matrix* clustering would be recommended, due to their values of medium authoritativeness being the best found in this work. *K-means* clustering would be discarded in most cases, since it requires the knowledge of the number of modules as input. It is worth mentioning, however, that results of authoritativeness both from this

work as well as from Wu et al.'s [Wu et al. 2005] are limited by the lack of adequate benchmarks for architectural decomposition. In both cases, the used heuristics of low-level package decomposition as the oracle of authoritative clustering are limited; this oracle may not always approximate the intended architecture of most software systems. Hence, it seems adequate to follow the path suggested by Sim et al., of producing benchmarks as a means to better evaluate software engineering techniques [Sim et al. 2003]. This seems to be especially appropriate to evaluate architecture recovery techniques, an important research topic that has produced a large number of techniques and tools [Pollet et al. 2007].

8.2 Incremental Mapping Techniques

When analyzing the mapping step of the reflexion model technique, I found that an appropriate combination of structural dependencies and software vocabulary allows to achieve automated mapping results with high accuracy, as it can be seen in the high *F-measure* values found. In the evaluation, the proposed incremental automated mapping technique that combines structure and vocabulary showed the highest *F-measure* values for both singleton, small and large source code changes. However, high accuracy still does not prevent some wrong mappings. That is why I believe that a semi-automated approach for incremental mapping seems to be the most appropriate, with an automated step reviewed by software developers. This approach takes advantage of the knowledge existent in software artifacts extracted by the mapping techniques, while it also gives the final decision to developers.

The mapping approach in the original reflexion model technique is based on regular expressions [Murphy et al. 1995]. Whenever directory structure, package organization or naming conventions do not capture an architectural decomposition, both automated and semi-automated mapping techniques can be useful and effective. Thus, in a context of undocumented software, where module views are recovered from, e.g., clustering techniques, regular expressions may not be the solution to mapping code elements onto high-level modules. Moreover, architectural drift also happens when architectural

knowledge is lost. Initial package or directory decompositions usually reflect architectural decisions, but with architectural drift, code elements are added to the system that do not respect previous architectural decisions [Sutton 2008]. Incremental mapping techniques such as the ones discussed in this work can help to recover this knowledge, mapping elements to their appropriate modules according to their structure and/or to the concepts expressed in their vocabulary.

Christl et al. evaluate their results of automated structural mapping [Christl et al. 2005; Christl et al. 2007] in a similar fashion as in this dissertation. However, I avoid the use of any filtering functions as they do. Filtering functions reduce the number of orphans submitted to automated mapping, and that can mask the accuracy of the mapping technique. In their work, they only map orphans with a high ratio of dependencies to mapped code entities, which leaves weakly connected entities out of the evaluation. Apart from this issue, their mapping function based on structural dependencies still shows, most of the time, better results than my technique purely based on information retrieval of software vocabulary, as it was shown by the *F-measure* values in the case studies. On the other hand, a combination of structural dependencies and information retrieval in a mapping algorithm based on two mapping steps gave the best results in terms of F-measure, which shows that the two different types of information can be complementary and improve the accuracy of the isolated mapping techniques.

Although incremental mapping techniques were mostly discussed in this work to improve the mapping step in the reflexion model technique, they can also be used in a context of incremental clustering, since this step might be beneficial to the architecture recovery phase, as discussed in the previous section. Incremental mapping may possibly improve results of techniques both for orphan adoption [Tzerpos 1997] and for maverick analysis [Schwanke 1991], although additional research needs to be performed to validate this claim. As in the orphan adoption technique of Tzerpos [Tzerpos 1997], the solution I presented uses more than one type of information to improve the accuracy of incremental mapping (or incremental clustering, as they name it). While I compute a mapping function to find the most likely module to map a code entity, they use structural and naming criteria in decision conditions of a flowchart in order to perform the mapping. They also add additional knowledge, such as a stylistic criterion,

e.g., if the orphan is either being mapped to a library or to a high-cohesion module.

8.3 Prioritizing Architectural Warnings

Observations from a variety of different software systems in which static architectural violations can amount to hundreds of weekly discrepancies led me to design a technique to prioritize the most relevant violations. I pursued this approach in this work, investigating some factors likely to correlate to violation relevance. Such approach is novel in this context, although similar work has been reported in the domain of bug finding tools [Kim and Ernst 2007b]. Inspiration of factors to be analyzed also came from related work [Kremenek and Engler 2003; Nagappan and Ball 2005; Bird et al. 2006; Ruthruff et al. 2008; Fritz et al. 2010], that I used as a first informed hint on what factors to investigate, even though the hypotheses raised on violation relevance did not reveal straight as predicted. However, two factors, namely violation duration and violation co-location, proved important to determine violation relevance. Other additional factors are worth analyzing, and this dissertation can be seen as a first step towards a more thorough investigation of static architectural violations.

In terms of practical tools for software developers, I showed the feasibility of a violation recommender that focuses developers' attention to a reduced and precise set of violations. Furthemore, I also showed the feasibility of filtering violations when a more complete procedure of static architecture analysis is needed. Filtering, however, still produces a non-neglectful amount of false negatives, and more research might likely help to improve its accuracy.

Results from the investigation of factors suggest that bug warnings are similar to architectural warnings in that each warning is related to a specific static dependency, and in that the duration of each warning can be determined from checking whether the same dependency occurs as the software evolves [Kim et al. 2006]. On the other hand, co-location of architecture warnings refers to a relationship between either two modules or two types, usually at a higher level than the co-location of bug warnings, which are computed at the lower level of methods/procedures [Kremenek and Engler

2003]. Similar to bug warnings, I looked at source code instabilities using measures of code churn [Nagappan and Ball 2005], and tried to understand how they affected architectural violations. However, I also looked at concepts that have not been analyzed on studies of prioritizing bug warnings. Such concepts are more related to people and their relationships to artifacts (e.g., a developer's knowledge about the source code) and to themselves (e.g., a developer's centrality in a social network of source code committers).

8.4 The ERM Process

This dissertation focused on the steps of a static conformance checking process that require intensive manual effort by software developers, such as producing a high-level model, mapping code entities onto high-level modules, and prioritizing results from checking. The solution, named the *evolutionary reflexion model* process, changes the original RM technique, introducing a partial automation of the aforementioned steps. Clearly, results were more promising for mapping and prioritizing than they were for producing a high-level model, as I inferred from the quantitative measures in the empirical evaluations.

Returning to Figures 2.3, 2.4 and 2.5 in Chapter 2, we can look at the impact of the developed techniques in the whole ERM process.

Had the empirical results been good for any of the four clustering algorithms evaluated in Chapter 5, in terms of accuracy and stability, there would have been an impact on the *design clustering* step shown in Figure 2.4. Instead, what I found out is that the abstracting power of these algorithms is limited in the context of software evolution. On the other hand, I argue that there are likely more promising results in using the incremental mapping techniques developed in Chapter 6 in the *semi-automated design re-clustering* step shown in the same figure. In this case, there is still a large manual effort to recover an initial architecture, but the effort to keep it up-to-date is reduced, thanks to incremental mapping/re-clustering techniques.

From another standpoint, I found substantially more promising results in the steps of

the conformance checking subprocess shown in Figure 2.5. Results of this work can effectively reduce the manual effort in the step of *semi-automated design mapping*. In Chapter 6, I found that the proposed incremental automated mapping technique that combines structural dependencies and source code vocabulary gets the highest *F-measure* values for both singleton, small and large source code changes. Since lightweight development processes usually work with incremental changes, and the design mapping has to be revised each time changes are applied to the source code, one can infer that manual effort shall be reduced when applying semi-automated design mapping to the conformance checking process.

As another positive outcome in the conformance checking subprocess, the step of *violation logging* in Figure 2.5 benefits from the techniques developed in Chapter 7. By focusing developers' attention to a reduced and precise set of violations found by a violation recommender based on software history, it is possible to reduce their manual effort to analyze results from conformance checks. Empirical results with a top-10 violation recommender showed an improvement in precision of at least 57% for four subject systems, when compared to the baseline of selecting K random violations from the full set of weekly violations. Knowing that time constraints might prevent developers' adopting a conformance checking process, especially when tool results show irrelevant information, results as the ones found in this work may contribute to the adoption of conformance checking techniques.

8.5 General Issues

Beyond the specific results for the ERM process steps, empirical results also brought scientific insights on the nature of software development and evolution. For instance, case studies on mapping showed differences between the nature of modularization that mapping techniques achieve, when comparing structural-based versus information retrieval techniques. Empirical investigation of factors likely to influence violation relevance suggested that architectural violations are very different from software bugs.

Finally, it seems important to return to the main research question in this dissertation:

how the manual effort to apply the reflexion model technique could be reduced in the context of evolving, sparselely documented software. I tried to answer this question by showing empirical evidence that the adoption of the ERM process can reduce this effort. Using specific software tools, partial automation of the manual effort is possible. This was shown in this work, especially in Chapters 6 and 7, that respectively deal with the steps of mapping and violation logging. The hypothesis of finding a both accurate and stable clustering technique to produce high-level models could not be supported by empirical evidence. However, I found empirical evidence that supports the hypothesis that combining structural dependencies and information from source code can indeed partially automate the mapping step in the RM technique, with improvement against existing techniques. I also found empirical evidence that correlation between violation relevance and variables from software history can be used to prioritize violations resulting from conformance checks, with improved precision against a baseline of a random selection of violations.

Last but not least, it is worth mentioning that existing commercial tools can benefit from the results of this dissertation, especially the ones devoted to static architecture analysis. Reduced manual effort when using such tools can improve the productivity of software developers performing tasks of architecture evaluation.

8.6 Future Work

Future work is discussed below, both on the main threads of this dissertation and on additional issues.

8.6.1 Evaluation of Architecture Recovery Techniques

Results found both in this work and in related literature [Wu et al. 2005] point to the limitation of existing quantitative measures to evaluate clustering algorithms. Combining quantitative measures with qualitative evaluation seems to be the most appropriate way to analyze architecture recovery techniques, especially when it regards to the usefulness of these techniques to software developers.

Another research path on quantitative evaluation is the use of synthetic models of software modularization and evolution. Clustering algorithms can be evaluated using a framework of complex networks and random graphs. Initial work has been pursued in this area by Souza et al. [Souza et al. 2010]. They suggest the use of synthetic software networks as a means of evaluating clustering algorithms, and propose a model for generating synthetic networks that follow architectural constraints. This evaluation approach might capture the complex nature of software decompositions that is not done by a measure of non-extremity, as it was explored in this dissertation. Adding the dimension of evolution to these models shall allow the generation of evolving models that could be used to evaluate architecture recovery techniques in a context of software evolution.

Benchmarks are needed in order to improve evaluation in software engineering research, as it has been pointed by Sim and colleagues [Sim et al. 2003]. The use of benchmarks to evaluate architecture recovery techniques has been previously proposed, and a benchmark for systems developed in C has been produced [Koschke and Simon 2003]. Adding systems to this benchmark would be appropriate, and the Java systems studied in this dissertation can be a good starting point. The dimension of evolution shall be incorporated to this type of benchmark.

The evaluation of clustering techniques performed in this work was targeted on published software releases of open source software. Using finer-grained changes such as weekly or monthly versions, or even atomic revisions commited to a software repository might reveal important issues on stability of the techniques, although the amount of information to be analyzed might be much larger.

Last, but not least, once there is some basic consensus on the research community on how evaluation should be done, on what metrics to use on quantitative evaluation, on what subjective aspects must be taken into account on qualitative evaluation of these techniques, there is room for further empirical investigation of a larger range of architecture recovery techniques, especially the ones focused on automated recovery.

8.6.2 Incremental Mapping/Clustering

The use of software repository mining to evaluate incremental mapping is a natural follow-up of the work performed in this dissertation. Using real changes as the input to mapping techniques will help to improve the validity of this study. Changes both on a fine-grained (e.g., atomic changes) and on a coarse-grained level (e.g., weekly or monthly changes) can be input to these techniques, since the frequency of conformance checks may vary in different projects.

A study of semi-automated mapping seems relevant, using a mapping tool as a recommender, instead of fully automated approach as it was discussed here. Both quantitative and qualitative analysis of semi-automated mapping would be appropriate for evaluation, focusing on solutions to limit the size of generated candidate sets, and on the classification accuracy of the candidate sets.

8.6.3 Recommenders for Architectural Warnings

Both in the investigation of factors related to violation relevance and in the recommending and filtering tools proposed in this dissertation, I used automated heuristics to assign the value of the dependent variable, i.e., violation relevance. The use of human oracles to assign relevance values can improve the validity of the evaluations performed, albeit the amount of information required to classify hundreds or thousands of violations of systems subject to evaluation.

An interesting experiment is to repeat this study in proprietary systems. A larger number of committers might be available in such systems, what would also improve the analysis of factors such as degree-of-authorship and developer centrality.

Another interesting analysis that can be performed is the use of multivariate statistical regression techniques to investigate the factors that correlate with violation relevance, such as in a previous work on bug warnings [Kuthruff et al. 2008].

And last, but not least, the analysis of additional factors that might correlate with violation relevance and that have not been studied here is a natural complement to this

work. Typical candidate factors would be module or type complexity, and cohesion metrics for modules or types.

8.6.4 Additional Issues

Prioritizing warnings from software tools is an active field of research. It has been strongly focused on bug warning tools, but it can be extended to other tools that warn developers of software issues. The same way it was done here for architecture checkers, it can be done for other tools such as compilers, testing frameworks, and low-level design checkers.

Architectural violations are important to reveal discrepancies between intended and implemented design. Work that focuses on pointing issues in the intended design and suggesting changes in high-level models seems an important research topic. Suggesting architectural changes from the analysis of module coupling and cohesion, or from other measures that relate to violation relevance is in the scope of a broader research topic of recommender systems for software engineering.

This work focused on a specific architecture checking technique, namely the reflexion model technique. Additional research shall integrate results of this work with analysis of other architecture checking techniques and tools. This might provide more generality to the results found here.

Finally, empirical evaluation of an architecture conformance checking process, as the one discussed in this work, is an important topic that also deserves attention from the research community. User studies of architecture checking tools and how these tools fit into the software development process seem an appropriate evaluation approach for this topic.

Chapter 9

Conclusions

CONCLUSÕES

Neste capítulo, as conclusões desta tese são elencadas, assim como as contribuições deste trabalho de pesquisa, e alguns comentários finais.

In this chapter, conclusions are drawn, followed by a summary of contributions of this dissertation and some final remarks.

My thesis has been that it is possible to enable static architectural conformance checking of evolving software, reducing the manual effort in this process by using information available in software artifacts. I have focused on specific steps of a conformance checking process based on the reflexion model technique, namely: *i*) the generation of high-level module views of undocumented software; *ii*) the mapping between source code and high-level modules; and *iii*) the analysis of results produced by architecture checks. The results have been particularly encouraging in the last two steps.

Enabling the generation of high-level models has been well-explored by the research community, and I evaluated specific existing software clustering techniques to generate a high-level model. The evaluation was based on measures of accuracy and stability. I measured accuracy using metrics of non-extremity of cluster distribution and authoritativeness of generated partitions, and I measured stability between generated models for two software versions. Empirical results from a case study with four subject systems were derived for four different clustering algorithms. Analysis of the results

showed that no algorithm performs best for all measures, and that they are limited in providing fully automated module view recovery. Instead, I argued that such clustering algorithms can be a first step in a process guided by software developers to produce high-level models, and that this step can be followed, during software evolution, by incremental clustering techniques.

The reflexion model technique requires a mapping between source code entities and the modules in the high-level architectural model. I showed that this mapping can be enabled by the use of automated mapping techniques. To do such, I proposed a mapping technique based on information retrieval of software vocabulary, and combined it with an existing mapping technique based on structural dependencies. This combination led to the best results of precision and recall in an empirical study that compared mapping techniques based only on structural dependencies, only on information retrieval, and on both. The empirical study was made of two case studies with four subject systems, nine high-level models and three different scenarios of source code changes. Results also pointed that, for the studied systems, the proposed technique showed the highest *F-measure* values for both singleton, small and large source code changes. Such results suggest that the mapping step in the conformance checking process can be enabled by automated techniques. However, results still need to be revised by software developers, which suggests a semi-automated approach for mapping.

Finally, I showed that the production of results from the checking step in the conformance checking process can be enabled by the use of a recommending tool based on software history. In order to do this, I first investigated five different factors extracted from software history that could likely correlate with the architectural violations generated by the checking step. Correlation with violation relevance was stronger with violation duration and violation co-location and weaker with degree-of-authorship, while no correlation was found with code churn and developer centrality. Then, I showed that prioritizing resulting violations is feasible by producing a prototype recommender based on support vector machines. Subsequently, I evaluated the recommender by retrospective experiments on four open source software systems. Results from a prioritizer that focused developer's attention to the top-10 most relevant architectural violations showed an improvement in precision of at least 57%, when compared to a

baseline of randomly-selected violations. Changing the prioritizer into a classifier allowed to analyze a different scenario: filtering irrelevant violations. This scenario was evaluated by experiments, where the classifier reduced the amount of work to analyze violations in more than half for the analyzed systems, while still keeping good filtering quality, measured in terms of specificity, sensitivity and precision.

9.1 Contributions

This dissertation adds to the body of knowledge on static architecture conformance checking of evolving software. More specifically, the contributions of this work are:

(a) An empirical evaluation of clustering algorithms in the context of software evolution that shows their limitation to recover static architecture module views in a fully automated scale;

(b) An incremental technique to map source code entities onto modules in high-level models based on information retrieval of software vocabulary, and its combination with a structural-based mapping technique;

(c) An empirical evaluation of incremental mapping techniques in terms of both precision and recall;

(d) An investigation of static architectural violations of evolving software systems, and of some of the factors that could correlate to the relevance of violations;

(e) Design and prototyping of a recommender system that prioritizes the top-K violations based on learning from previous software history;

(f) Evaluation of the recommender system above in terms of precision of the recommendations;

(g) Design and prototyping of a filter that removes irrelevant violations from checking results;

(h) Evaluation of the violation filter in terms of information retrieval criteria of specificity, sensitivity and precision;

(i) Development of a prototype toolset to enable partial automation of a lightweight static conformance checking process.

9.2 Final Remarks

Adding the dimension of software evolution to the analysis of software engineering techniques, as it was done in this work with static architecture conformance checking, requires different evaluation approaches. It is possible to both model software changes as well as mine changes from software repositories. Different granularities may be used for the time variable, from atomic changes to a repository, weekly versions or even software releases. Evaluation criteria must deal with the dimension of evolution, and even new criteria may be derived from this dimension. Studies such as the ones performed in this dissertation can bring new insights to the scientific process of understanding software evolution as well as to the design of tools, techniques and approaches to keep software evolution and complexity under control by the software team.

Appendix A

Design Suite: A Toolset for the Evolutionary Reflexion Model Process

Design Suite: UMA SUÍTE DE FERRAMENTAS PARA O PROCESSO DE MODELOS DE REFLEXÃO EVOLUCIONÁRICS

Na Seção 2.2, foi descrito um processo de checagem de conformidade intitulado processo de modelos de reflexão evolucionários (ERM). Para tornar o processo ERM factível e facilitar o seu uso em um ambiente de pesquisa, foi projetada uma suíte de ferramentas chamada Design Suite. Neste Apêndice, esta suíte é descrita, assim como os detalhes de cada etapa do processo ERM, e como estas etapas são implementadas na suíte.

In Section 2.2, I described a conformance checking process named the *evolutionary reflexion model* (ERM) process. To make the ERM process feasible and to facilitate its use in a research enviroment, I designed a toolset named *Design Suite*. In this Appendix, I describe this toolset and provide details about each step of the ERM process and how these steps are implemented in the toolset.

The toolset is made of the following tools:

Design Wizard: Previously existing low-level conformance tool [Brunet et al. 2011] that was adapted to extract designs into a GXL file;

***Design Model*:** Stores a multi-level design in main memory;

***Design Abstractor*:** Lifts, filters and clusters designs;

***Design Viewer*:** Visualizes designs in different layouts and allows online interaction with the other tools;

***Design Mapper*:** Maps lower-level entities onto higher-level modules;

***Design Checker*:** Allows to define high-level models and check them by producing reflexion models;

***Design Miner*:** Improves results of checking by means of either prioritizing or filtering architectural violations.

A diagram showing the data flow in the tools follows a similar flow as the ERM process and is shown in Figure A.1. It can be seen that the *Design Viewer* tool provides a platform to interact with and to visualize results from the other tools.

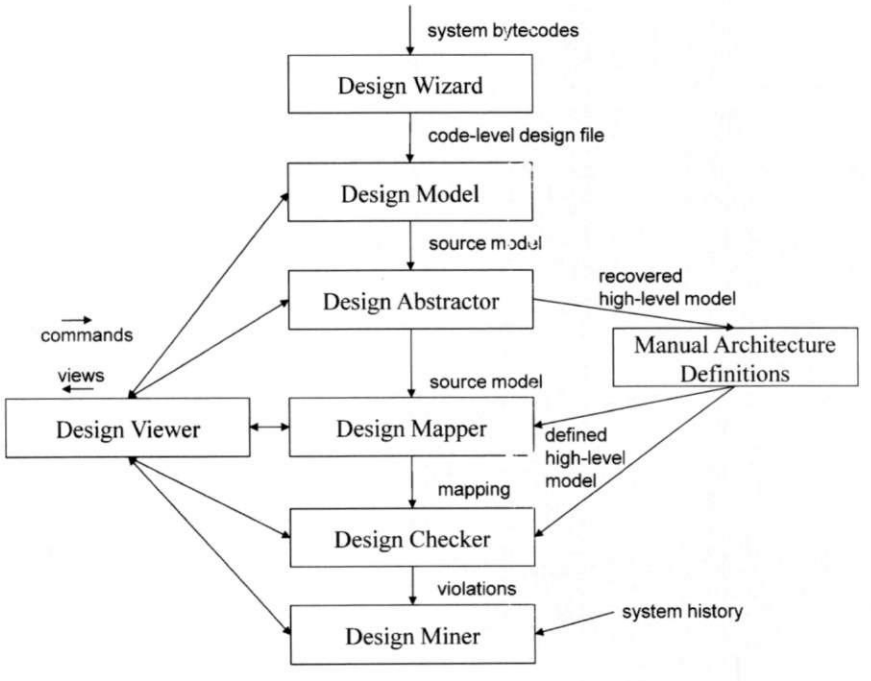


Figure A.1: Dataflow view for the *Design Suite* toolset

An architecture module view of the toolset is shown in Figure A.2. Static dependencies between components and tools are shown as directed edges.

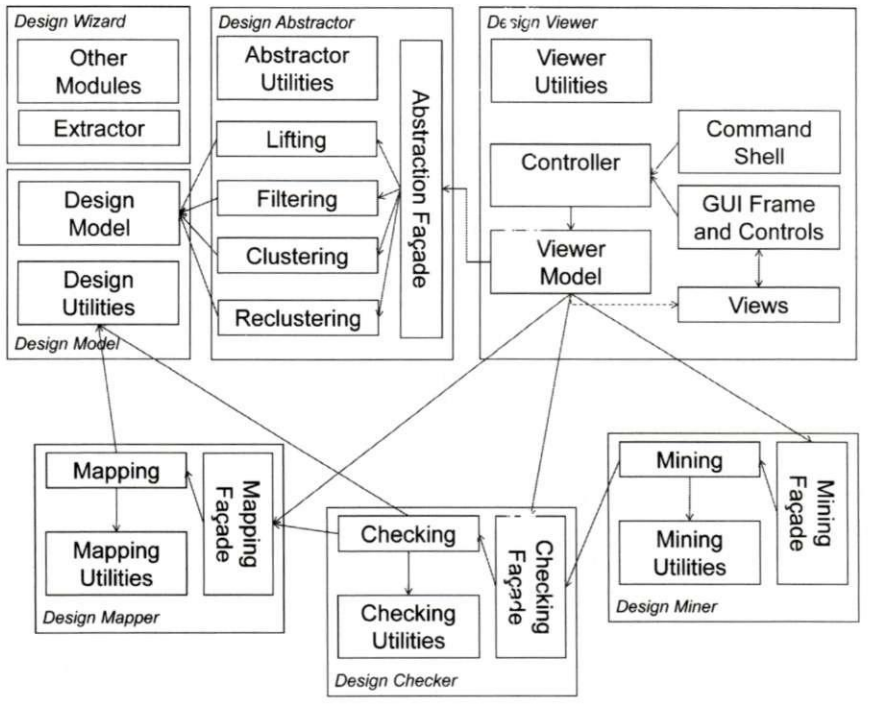


Figure A.2: Architecture module view for the *Design Suite* toolset

A.1 Architecture recovery

Different from the original reflexion technique, where a top-down process for architecture recovery is pursued, I propose a bottom-up process instead. Starting from a source code version retrieved from the software repository, design is extracted, lifted and clustered into a high-level design in an automated fashion. Semi-automated and manual steps allow to re-cluster the high-level design into a meaningful representation in terms of a module view, which is completed with the definition of structural architectural rules.

A.1.1 Design Extraction

The extraction of software design implemented in this work is limited to object-oriented code implemented in Java. Compiled bytecodes from a Java software are retrieved from the software repository. Then, facts are extracted by the *Design Wizard*

tool[1], which recovers entities and relations into a low-level design model. Extracted entities and relations are presented in Table A.1.

Entities and relations are extracted from the abstract syntax tree (AST) and only the relevant information is kept in the design model. In the extracted entities, information is limited to packages, types and members of Java types. Code statements are not kept in the design model, but are used to recover relations between entities. Relations include containment relations, inheritance between types and additional dependencies between types such as method calls and field accesses.

Table A.1: Entities and relations extracted by *Design Wizard*

Entity	Relation
package	package **contains** class
	package **contains** interface
interface	interface **contains** field
	interface **contains** method
	interface **extends** interface
class	class **contains** field
	class **contains** method
	class **contains** class
	class **extends** class
	class **implements** interface
field	field **is-a** class
method	method **calls** method
	method **accesses** field
	method **receives** field
	method **returns** field
	method **throws** class
	method **catches** class

Extracted design from *Design Wizard* is exported to a graph representation in an XML format named GXL, a format designed for data sharing between reengineering tools [Holt et al. 2006]. The result is a *typed*, *attributed* and *directed* multigraph.

[1] http://wwww.designwizard.org

A.1.2 Design Lifting

Although software clustering could be applied to low-level entities like methods and fields, Java object-oriented data types provide a better abstraction level to start the clustering step. In order to work with types, one first has to encapsulate lower-level entities such as fields, methods and inner classes in their containing types and lift lower-level relations to the level of type dependencies. This step is known as design lifting and is performed using the *Design Abstractor* tool, which was developed to support this work.

A special graph results from the lifting step: a *typed*, *attributed*, *directed*, ***hierarchical*** multigraph. It can either be stored in memory to allow for further processing or exported to other tools as a graph file in GXL.

A.1.3 Design Clustering

Clustering was chosen as the architecture recovery technique to abstract types into modules because of the variety of available algorithms for software clustering as well as the variety of information that clustering can take as input (e.g.: structural dependencies, source code vocabulary).

Five previously existing algorithms for software clustering were implemented in the *Design Abstractor* tool in order to give software developers a variety of architecture recovery techniques. Two of them are algorithms typically used in the pattern recognition community: *k-means clustering* and *hierarchical clustering*. They both take a feature vector as input. In the context of this work, a feature vector may either be a vector of structural dependencies or a vector with information extracted from source code vocabulary. Two other algorithms are *modularization quality clustering*, *design structure matrix clustering*, both optimization algorithms. They only take structural dependencies as input and use them to partition the software into structurally cohesive and loosely coupled clusters. The last algorithm is *edge betweenness clustering*, which uses heuristics from social networks to discover communities in a graph. These algorithms are further explained in chapter 5.

Clustering operations may either be done through an API supplied by *Design Abstractor* or through a GUI for design visualization named *Design Viewer*, also developed to support this work.

A.1.4 Semi-Automated Design Re-Clustering

An adaption of the orphan adoption technique [Tzerpos 1997] is used in order to improve the clusterings produced by design clustering algorithms. An attraction function is calculated between an orphan entity in a singleton cluster (or even in a small cluster) and another existing cluster. Clusters with higher function value will attract orphan entities into them, improving the non-extremity of the clustering [Bittencourt and Guerrero 2009]. This process may be fully automated or semi-automated. In the former case, orphan entities are adopted by the module with highest attraction, provided that this attraction value exceeds a threshold. In the latter case, software developers choose among a set of clusters, ranked by the attraction function value.

A.1.5 Manual Re-Clustering

Clustering and orphan adoption techniques impose a clustering based on the heuristics used in each algorithm. Since the software clustering must reflect the developers' view of the software, these automated and semi-automated techniques work as a first approximation to the final module view. Manual re-clustering allows developers to change and improve the design clustering according to their own heuristics and knowledge of the system. Three operations of manual re-clustering were implemented on *Design Abstractor*: entity *moves*, module *splits* and module *joins*. Executing these operations are facilitated by a graphical interface to the re-clustering process, implemented as a command in the *Design Viewer* tool.

A.1.6 Definition of Module Views and Architectural Rules

After discovering the modules that compose the software system, software developers have to name these modules for better understanding and to facilitate communication

among stakeholders. *Design Viewer* can be used to change names given by automated or semi-automated techniques.

In addition, developers also need to establish relations between modules. These are the structural *architectural rules* used to impose constraints over the development process to enforce architectural decisions. Rules are written in a plain text file, which is later input to the *Design Checker* tool.

As previously mentioned in this chapter and elsewhere in this text, the envisioned scenario for use of these rules is in lightweight development processes. In such a scenario, architectural rule checking should not impose a large burden. Instead, it might be appropriate to insert the conformance checking process into existing activities of quality assurance.

Design Checker also allows to generate architecture module view graphs based on the established rules in the textfile. By doing this, it is possible to view the list of recovered modules and the architecture constraints as either a graph diagram or another type of visualization layout in the *Design Viewer* tool.

A.2 Conformance Checking

The subprocess of architecture recovery ends up with an architecture module view and architectural rules. It might naturally be followed by checking architectural rules and that can be done straight after architecture recovery finishes. Nevertheless, as software evolves, checking must be done against a different source code. Source code entities may have been added, removed or changed. Even software architecture may have changed, although not so frequently as source code. Thus, updating representations of architecture and implementation may be needed in the conformance checking subprocess. Choosing when to update them, whether daily, weekly or arbitrarily is another research issue out of the scope of this work, since this update adds a burden to software developers.

The envisioned subprocess starts with repeating design extraction and lifting steps as in architecture recovery. Since clustering has previously produced a mapping from

type-level entities to modules, the next step here is keeping this mapping up-to-date, instead of doing clustering from scratch. Mapping is incrementally updated in a semi-automated fashion by exploring information from source code. It may happen that semi-automated mapping techniques follow heuristics that do not fit to developers' rationale and a manual re-mapping step may be needed. After updating mapping, developers may focus on the designed architecture. Although not so frequent as source code changes, architecture changes may be needed in order to allow flexibility in design and adapt to a changing context. Finally, with up-to-date representations of designed architecture and implementation, one may proceed to check conformance of architectural rules. Producing reflexion models with the *Design Checker* API allows to uncover eventual rule violations. *Design Checker* also logs rule violations, allowing for later analysis. Violation logs are also used in the *Design Miner* tool, together with additional measures based on software history, to improve the results of architectural checks. This improvement is be done by *Design Miner* either by filtering irrelevant violations or by producing a priority list of violations. Solving violations requires developers to either correct source code to abide by architectural rules, which might be the usual action, or fix architectural rules, which might be unusual but might also be relevant for not sacrificing design flexibility in a lightweight development process.

The steps of the conformance checking subprocess are described in detail in the following.

A.2.1 Architecture Changes

Software changes may sometimes have a negative impact on software internal quality. Architecture erosion may arise and one needs to be able to refactor software architecture the very same way source code can be refactored. Architecture module view changes may happen to either modules or dependencies. Module changes may either be module additions, removals, *joins* or *splits*. And dependency changes may be constraint *additions*, constraint *removals* and constraint *exceptions*. This step produces an up-to-date architecture module view and its associated rules that allow the following steps to be performed.

A.2.2 Design Re-Extraction and Re-Lifting

Design is re-extracted and re-lifted to update entities and relations that might have changed both in low-level and in type-level design. Both extraction and lifting happen in the same way as previously described in the section on architecture recovery.

A.2.3 (Semi-)Automated Mapping

Updating the mapping from type-level entities to modules is essential to compute reflexion models that take into account source code changes. Removed type-level entities are simply removed from mapping. On the other hand, newly added type-level entities should be mapped just like previously existing entities, if they are to be considered in architectural rule checking. Moreover, architecture changes may also drive mapping changes.

Adding new type-level entities to modules is solved by means of orphan adoption techniques. A new entity is called an orphan and information from source code may be used as a heuristic to compute the most likely module that should adopt the orphan. Mapping is done by means of attraction functions. An attraction function should rank modules according to their likelihood to adopt the orphan. Information from source code to compute the ranking is derived either from structural dependencies, from source code vocabulary or from a combination of both. Three attraction functions were implemented: $countAttract$, $MQAttract$ and $IRAttract$. The two first ones are based on structure and the last one on vocabulary.

Mapping is performed in *Design Mapper* either as a fully automated step or a semi-automated one. Fully automated mapping allows adopting orphans by modules whose attraction value is the highest and, besides that, detached from other modules' attraction values. Semi-automated mapping, instead, shows a list of modules ranked by attraction value and the developers themselves decide which module should adopt the orphan.

A.2.4 Manual (Re-)Mapping

As in manual re-clustering, manual re-mapping may be needed to map newly added entities to modules according to software developers' rationale, which may not be captured by semi-automated mapping heuristics. Part of manual re-mapping is similar to design re-clustering and it uses *Design Viewer* as an interface to move entities between modules. The remaining part of re-mapping consists of manually mapping orphan entities not mapped in the semi-automated mapping step.

A.2.5 Checking and Violation Logging

This is the most important step from the conformance checking process. It should be performed frequently, both in the pre-commit phase and in nightly or weekly builds. It simply means computing reflexion models betweeen the lifted source model and the high-level model, by using the *Design Checker* tool.

Checking during pre-commit changes allows developers to perceive violations before they are made permanent. It can become part of the development process just like running unit tests has.

During nightly or weekly builds, checking can be performed together with software quality assurance procedures. Architectural rule violations are logged, providing details about each violation, i.e.:

- which architectural rule is violated and which modules are involved;
- which type-level entities are involved in the violation and what is the associated dependency that causes it;
- which low-level source code entities are involved in the violation and what is the associated dependency in the source code that causes it.

The *Design Miner* tool flattens the hierarchical architectural violation logs into low-level violation logs. With these flattened violation logs, and from extracting additional measures from the history of software artifacts, it is possible to either filter irrelevant

violations or produce a priority list of violations to focus developer's attention to the most important architectural rule violations.

A.2.6 Violation Resolution

With details from violations from either graphical reflexion models or textual violation logs, developers may fix the problems by either changing source code or changing the architecture module view and rules.

Appendix B

Results of the Study of Clustering Algorithms

Resultados do Estudo de Algoritmos de Agrupamento

No Capítulo 5, foi apresentado o design experimental, alguns resultados e uma discussão de uma avaliação experimental de algoritmos de agrupamento. Aqui, eu apresento gráficos completos para as medidas absolutas de não-extremidade da distribuição dos clusters, autoridade e estabilidade.

In Chapter 5, I presented the experimental design, some results and a discussion of an empirical evaluation of clustering algorithms. Here, I present the complete graphs for the absolute measures of non-extremity of clustering distribution, authoritativeness and stability.

B.1 Non-Extremity of Cluster Distribution

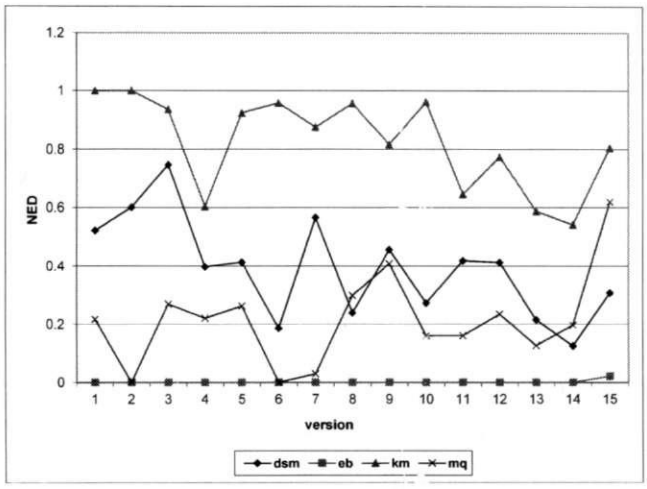


Figure B.1: NED scores for *JUnit*

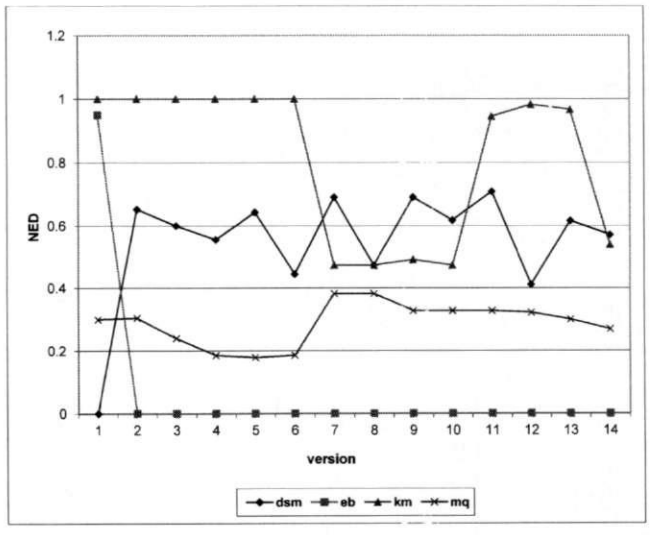


Figure B.2: NED scores for *EasyMock*

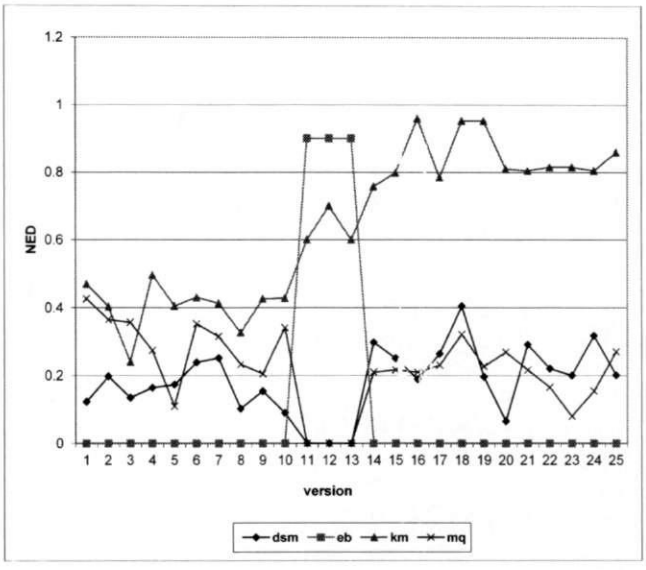


Figure B.3: NED scores for *JEdit*

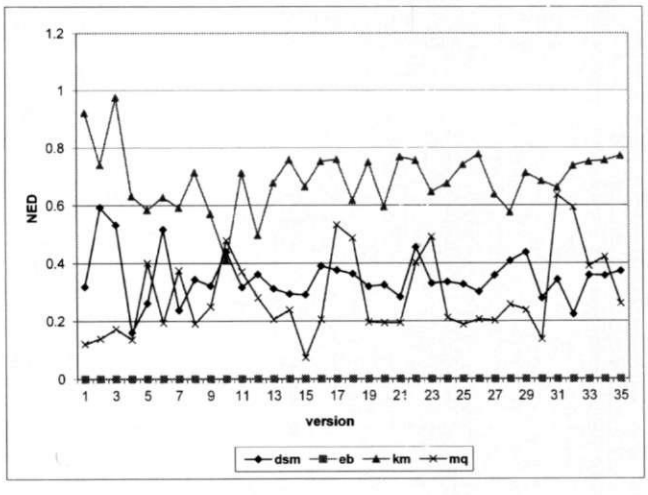


Figure B.4: NED scores for *JabRef*

B.2 Authoritativeness

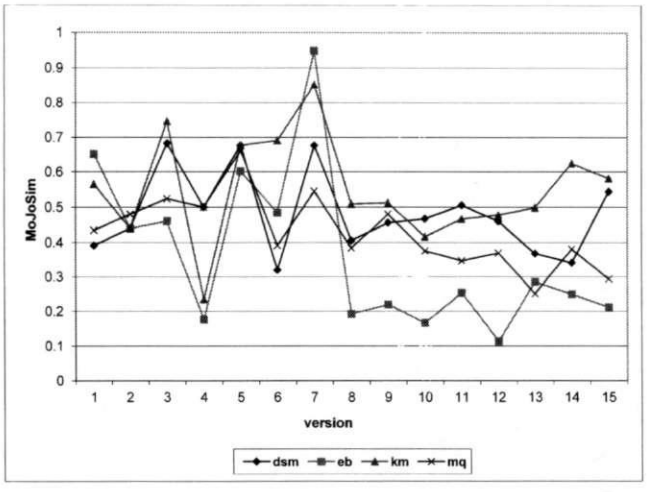


Figure B.5: $MoJoSim$ authoritativeness scores for *JUnit*

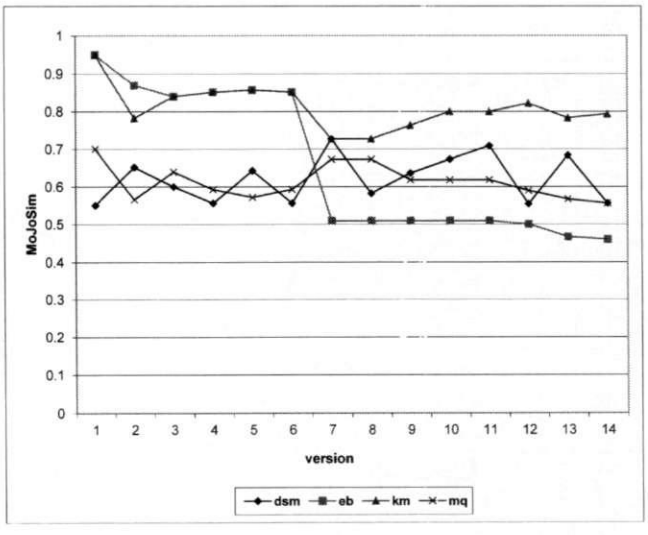


Figure B.6: $MoJoSim$ authoritativeness scores for *EasyMock*

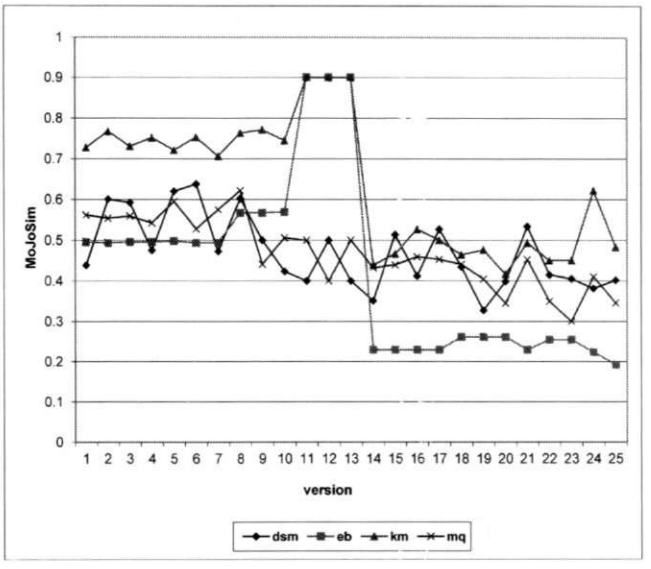


Figure B.7: $MoJoSim$ authoritativeness scores for *JEdit*

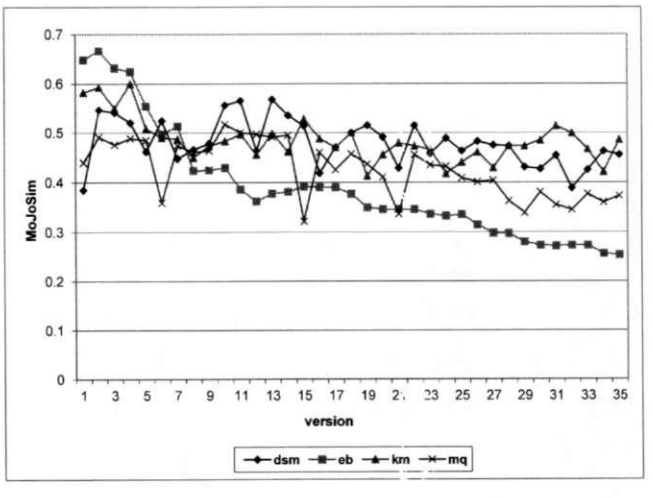


Figure B.8: $MoJoSim$ authoritativeness scores for *JabRef*

B.3 Stability

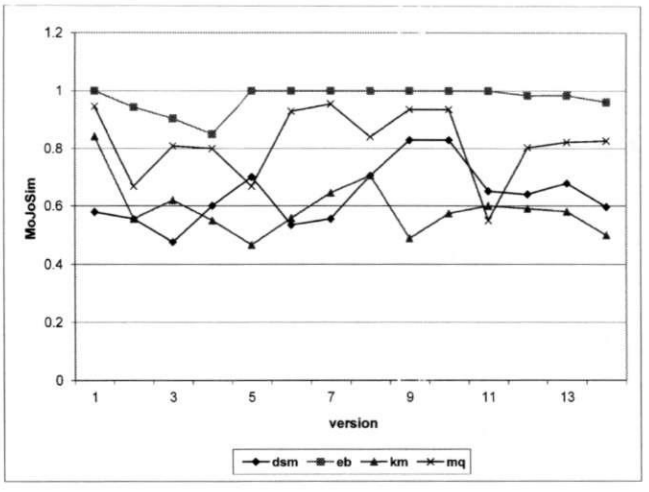


Figure B.9: $MoJoSim$ stability scores for *JUnit*

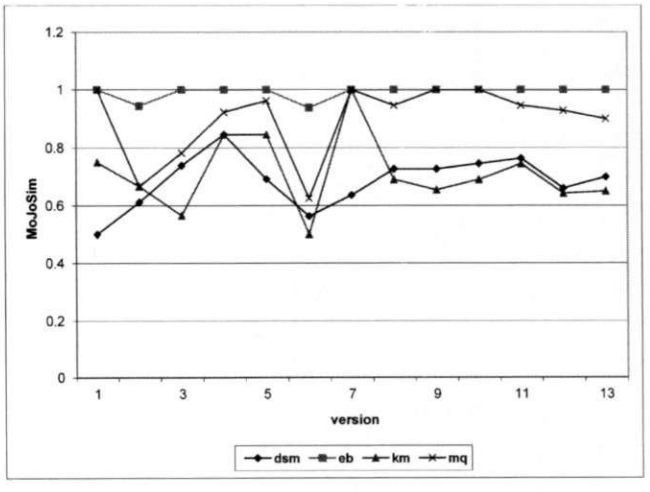


Figure B.10: $MoJoSim$ stability scores for *EasyMock*

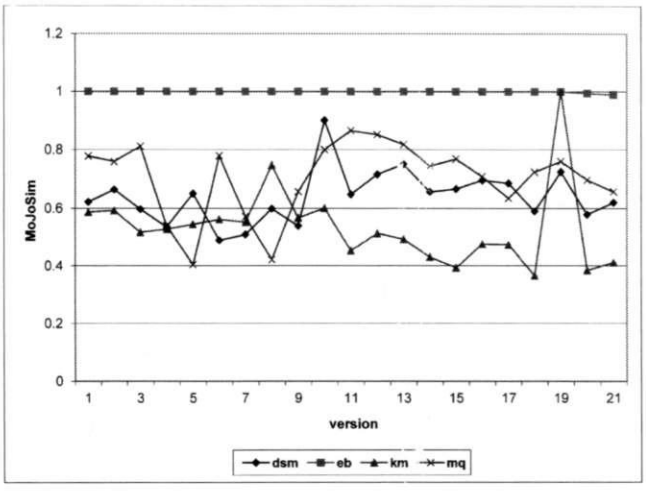


Figure B.11: $MoJoSim$ stability scores for *JEdit*

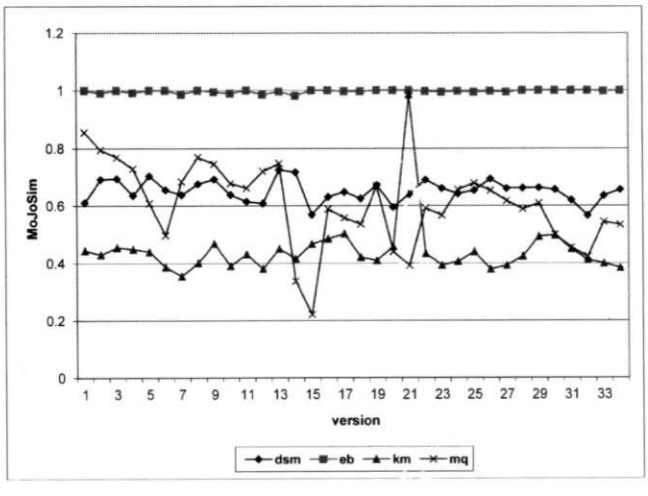


Figure B.12: $MoJoSim$ stability scores for *JabRef*

Appendix C

Models and Mapping for the Study of Mapping Techniques

Modelos e Mapeamentos para o Estudo de Técnicas de Mapeamento

Neste Apêndice, os modelos de alto nível, modelos de código-fonte e mapeamento são disponibilizados na forma de links para um web site com os dados brutos.

System data for high-level models, source code and mapping is available in a directory of a web site.[1]

This directory with raw data contains:

- files with a graphical representation of nine module views for the four subject systems;
- files with a graph representation (in GXL) of the source code entities for the four subject systems, extracted by the *Design Wizard* tool; and
- files with the mapping from source code entities for all four subject systems onto the high-level modules in their respective module views.

[1] Files can be found at `www.gmf.ufcg.edu.br/~roberto/data/wcre2010/`

The mapping files only contain the *ids* for each mapped source code entity. The *id* tag uniquely defines each entity and is represented as a node in the GXL file for the source code entities. The name of each entity is available in this same, under the *label* tag for each node. Files for each subject system are described next.

C.1 Design Wizard (DW)

High-Level Module View:

`DesignWizard_high.pdf`

Low-Level Module View:

`DesignWizard_low.pdf`

Source Code Entities:

`dw_ll.gxl`

Mapping onto High-Level Module View:

`dw_mapping_high.txt`

Mapping onto Low-Level Module View:

`dw_mapping_low.txt`

C.2 Design Suite (DS)

High-Level Module View:

`DesignSuite_high.pdf`

Low-Level Module View:

`DesignSuite_low.pdf`

Source Code Entities:

`suite_l1.gxl`

Mapping onto High-Level Module View:

`suite_mapping_high.txt`

Mapping onto Low-Level Module View:

`suite_mapping_low.txt`

C.3 OurGrid (OG)

High-Level Module View 1 (layers):

`OurGrid_high_and_low.jpg`

High-Level Module View 2 (components):

`OurGrid_comp.pdf`

Low-Level Module View:

`OurGrid_high_and_low.jpg`

Source Code Entities:

`ourgrid_l1.gxl`

Mapping onto High-Level Module View 1:

`ourgrid_mapping_high.txt`

Mapping onto High-Level Module View 2:

`ourgrid_mapping_comp.txt`

Mapping onto Low-Level Module View:

`ourgrid_mapping_low.txt`

C.4 Mylyn (ML)

High-Level Module View:

`Mylyn_high_and_low.pdf`

Low-Level Module View:

`Mylyn_high_and_low.pdf`

Source Code Entities:

`mylyn_l1.gxl`

Mapping onto High-Level Module View:

`mylyn_mapping_high.txt`

Mapping onto Low-Level Module View:

`mylyn_mapping_low.txt`

Appendix D

Results of the Study of Mapping Techniques

Resultados do Estudo de Técnicas de Mapeamento

No Capítulo 6, for apresentado o design experimental, algubns resultados e uma avaliação de técnicas automáticas de mapeamento incremental. Aqui, são apresentados os resultados completos da avaliação quantitativa.

In Chapter 6, I presented the experimental design, some results and a discussion of an evaluation of incremental automated mapping techniques. Here, I present the complete results of the quantitative evaluation.

D.1 Case Study 1: Mapping onto High-Level Views

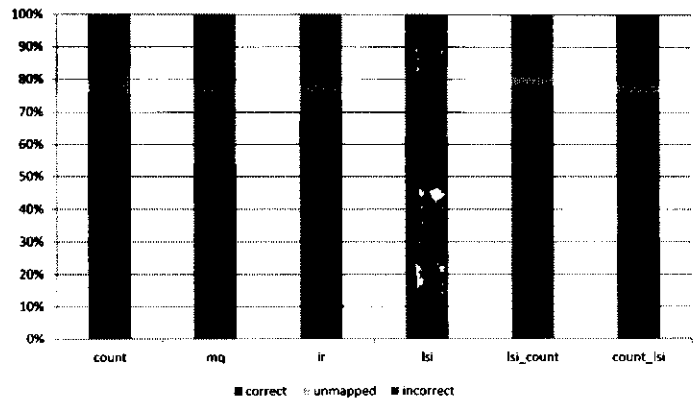


Figure D.1: Mass function for singleton changes: *Design Wizard* high-level view

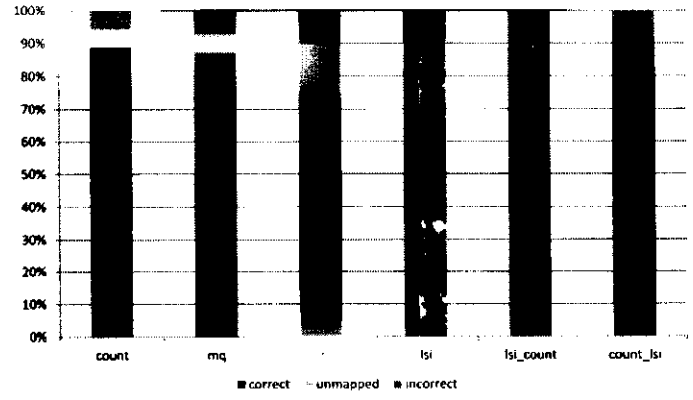


Figure D.2: Mass function for singleton changes: *Design Suite* high-level view

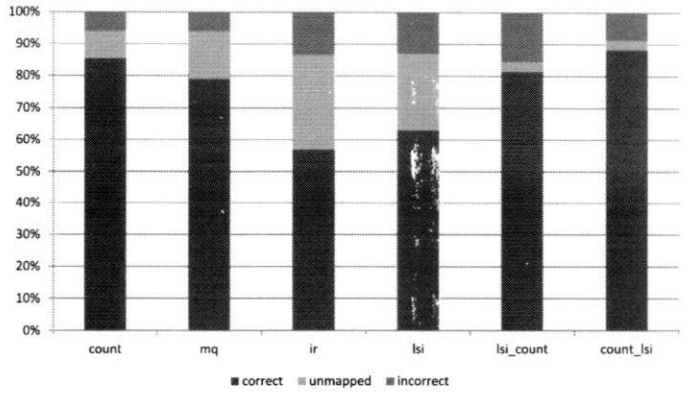


Figure D.3: Mass function for singleton changes: *Mylyn* high-level view

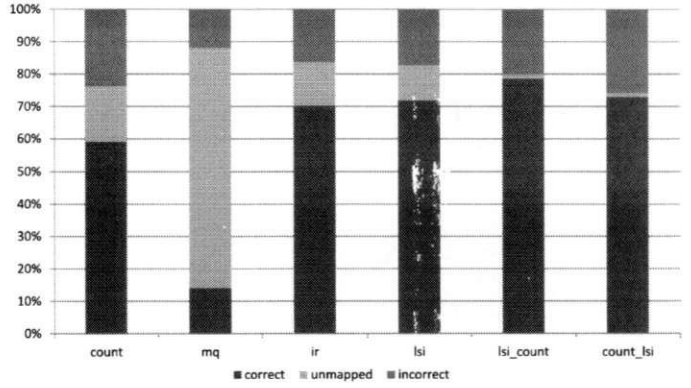


Figure D.4: Mass function for singleton changes: *OurGrid* high-level layered view

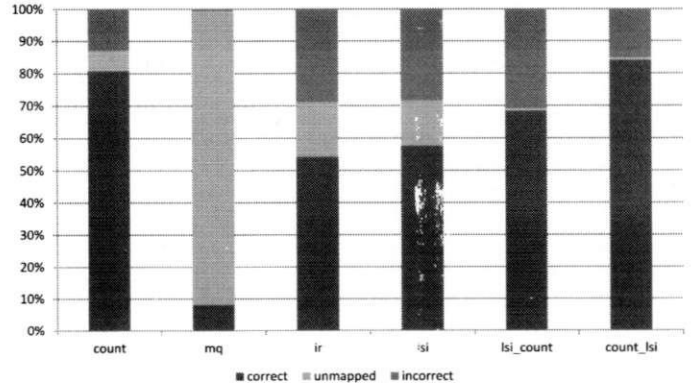


Figure D.5: Mass function for singleton changes: *OurGrid* high-level component view

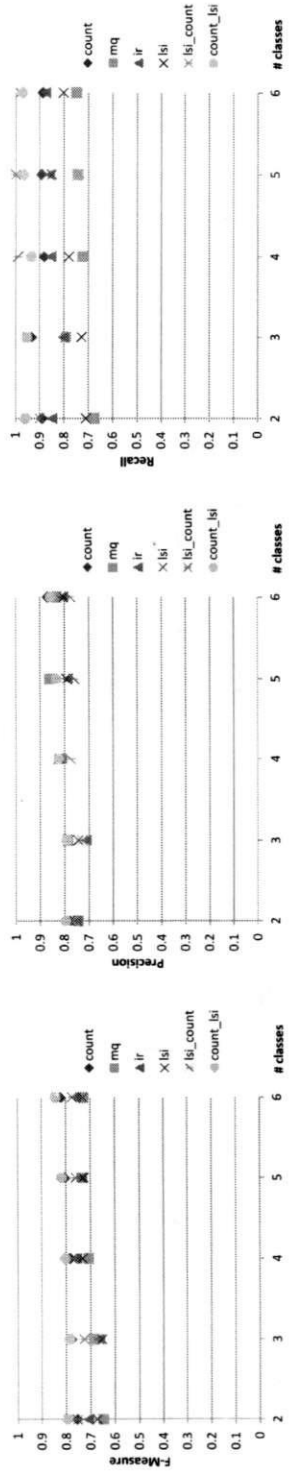


Figure D.6: Measures for small changes: *Design Wizard* high-level view

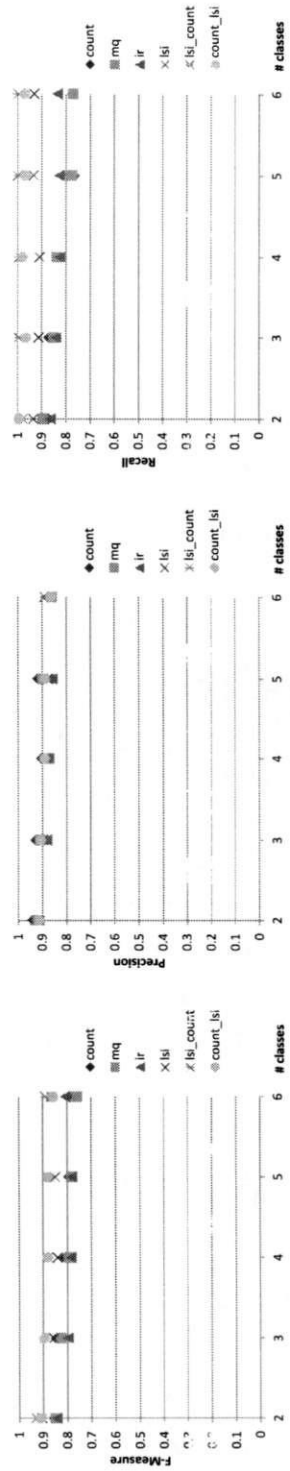


Figure D.7: Measures for small changes: *Design Suite* high-level view

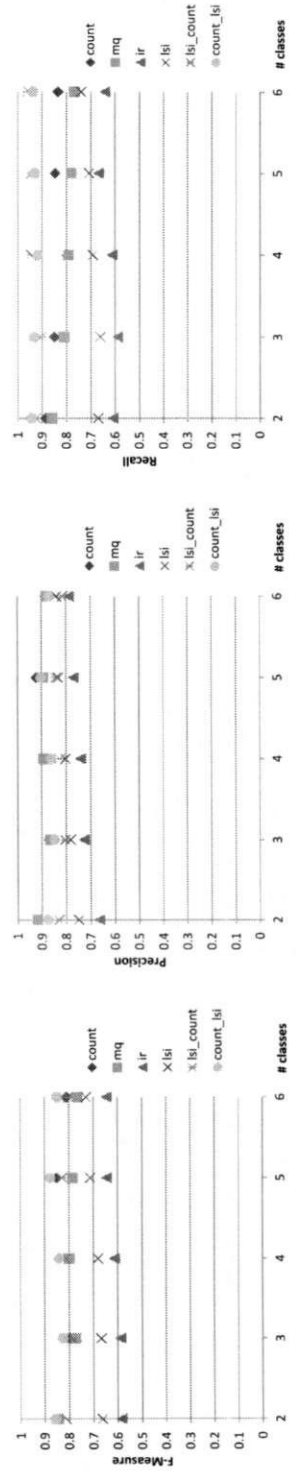


Figure D.8: Measures for small changes: *Mylyn* high-level view

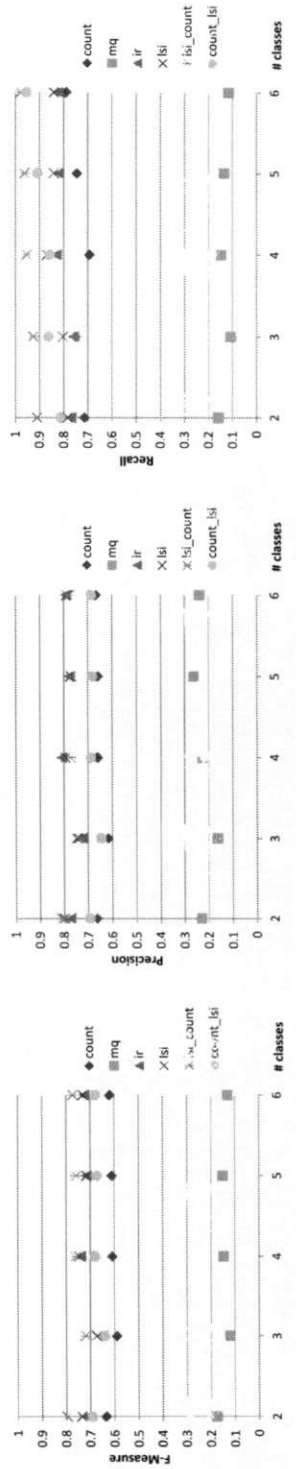


Figure D.9: Measures for small changes: *OurGrid* high-level layered view

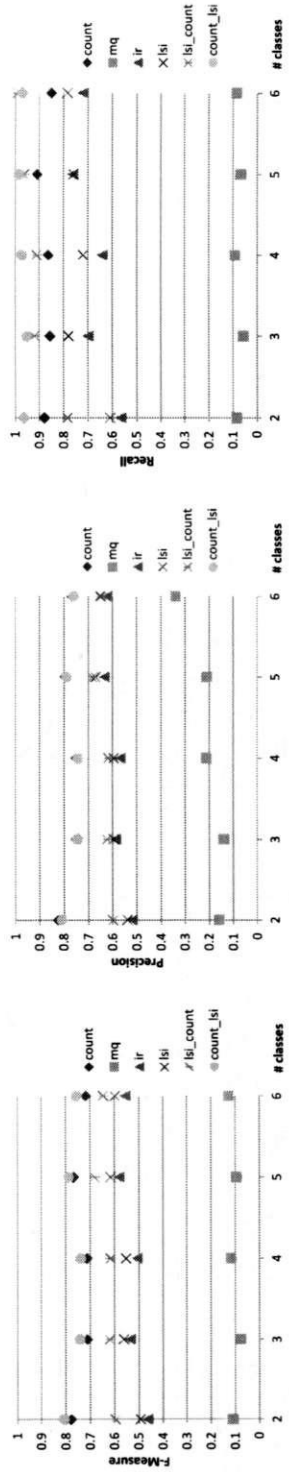


Figure D.10: Measures for small changes: *OurGrid* high-level component view

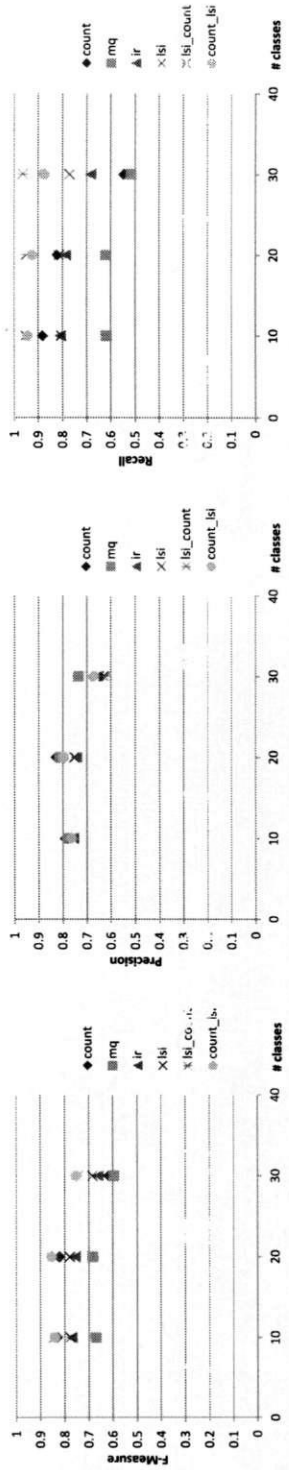


Figure D.11: Measures for large changes: *Design Wizard* high-level view

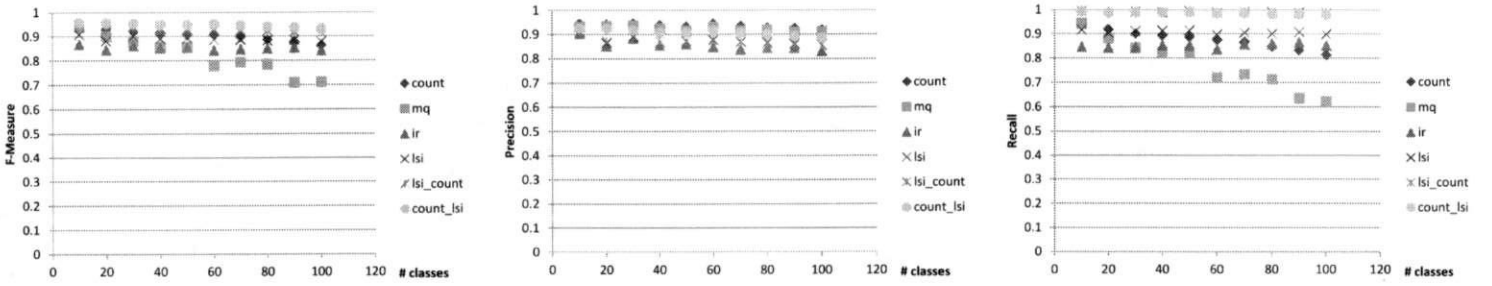


Figure D.12: Measures for large changes:*Design Suite* high-level view

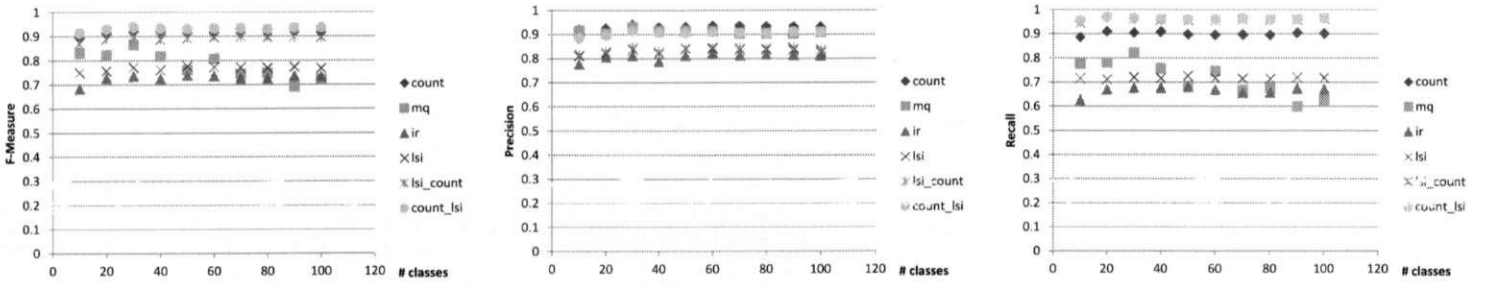


Figure D.13: Measures for large changes: *Mylyn* high-level view

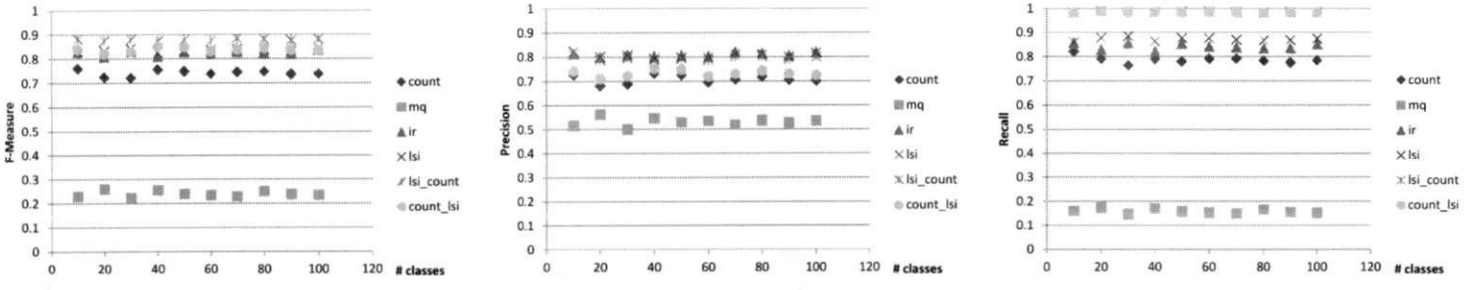


Figure D.14: Measures for large changes: *OurGrid* high-level layered view

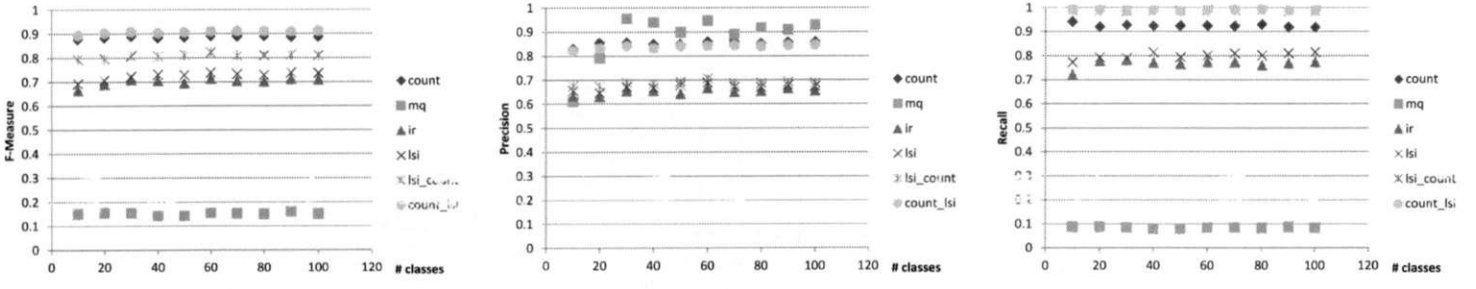


Figure D.15: Measures for large changes: *OurGrid* high-level component view

D.2 Case Study 2: Mapping onto Low-Level Views

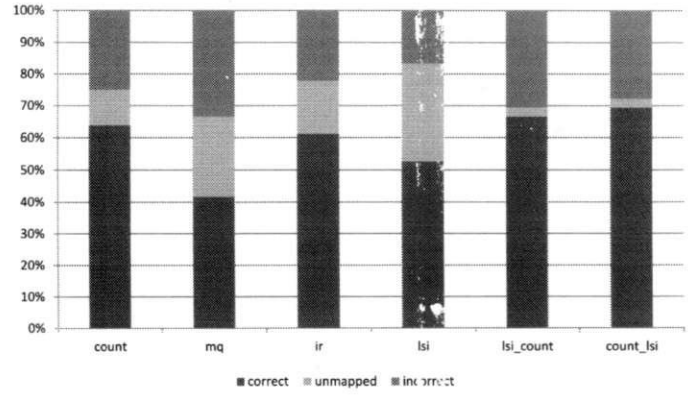


Figure D.16: Mass function for singleton changes: *Design Wizard* low-level view

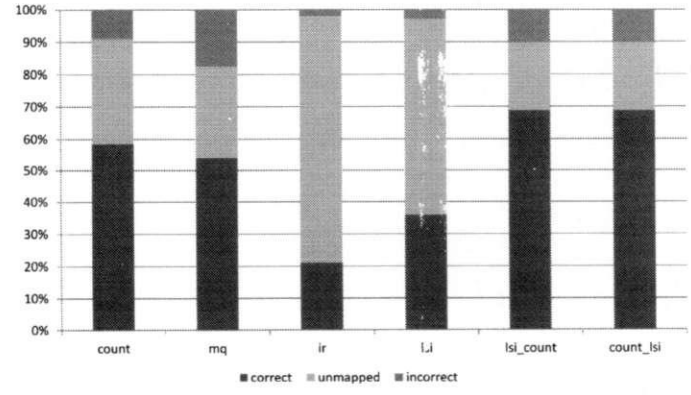


Figure D.17: Mass function for singleton changes: *Design Suite* low-level view

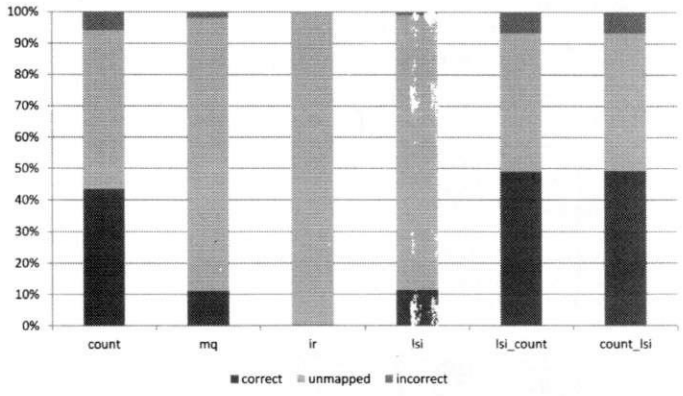


Figure D.18: Mass function for singleton changes: *Mylyn* low-level view

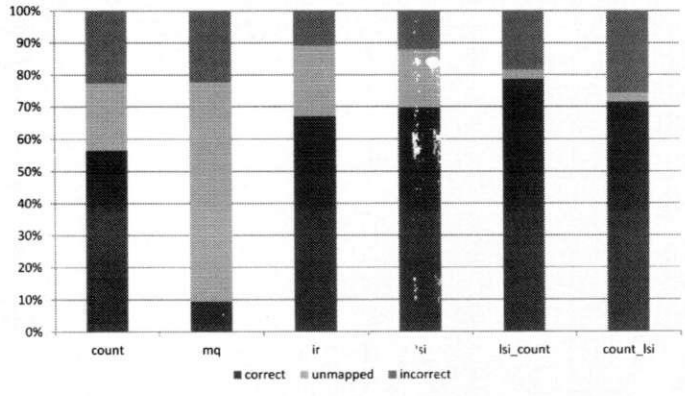


Figure D.19: Mass function for singleton changes: *OurGrid* low-level view

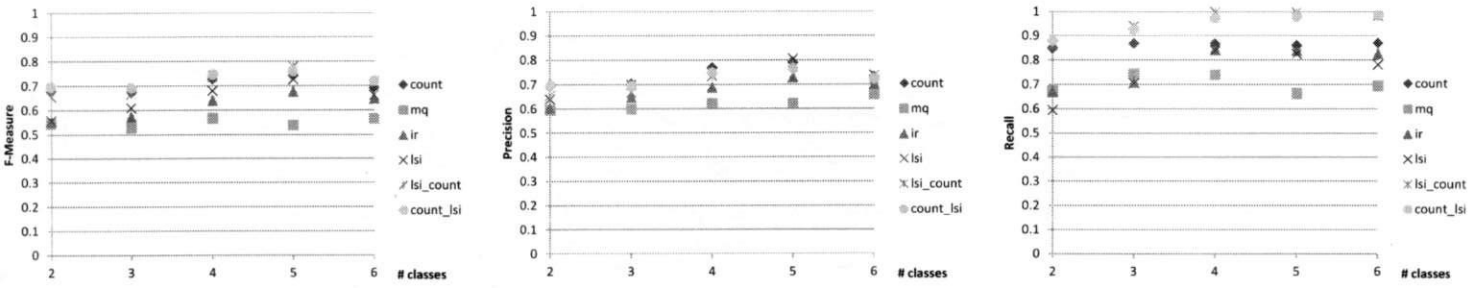


Figure D.20: Measures for small changes: *Design Wizard* low-level view

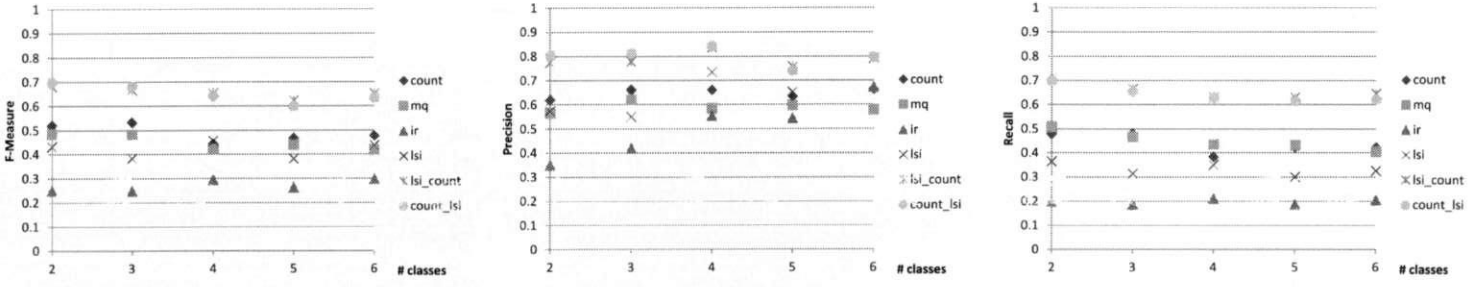


Figure D.21: Measures for small changes: *Design Suite* low-level view

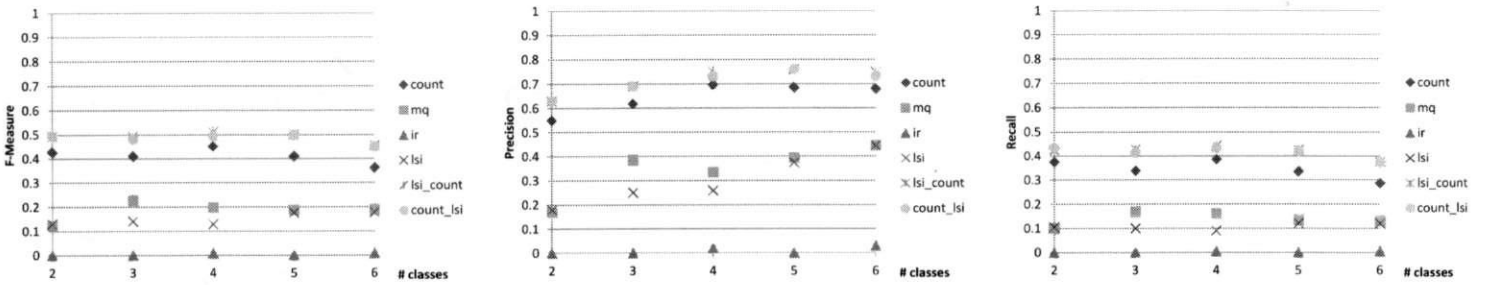


Figure D.22: Measures for small changes: *Mylyn* low-level view

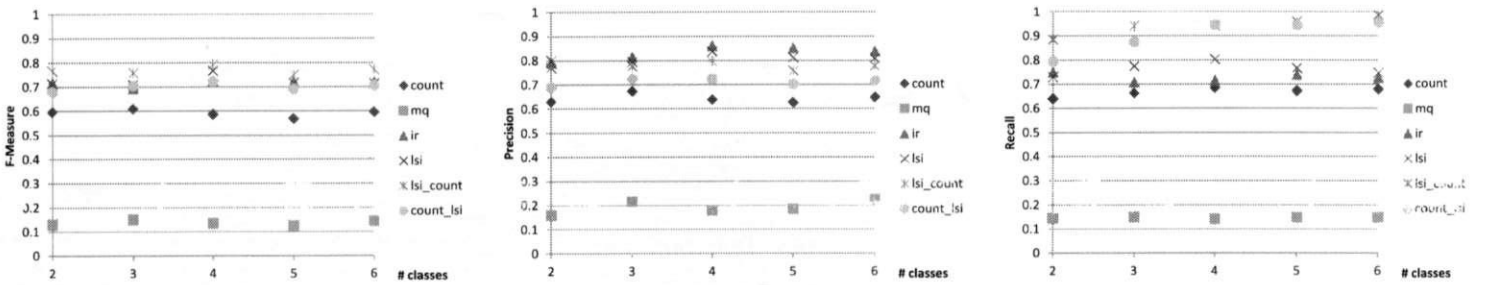


Figure D.23: Measures for small changes: *OurGrid* low-level view

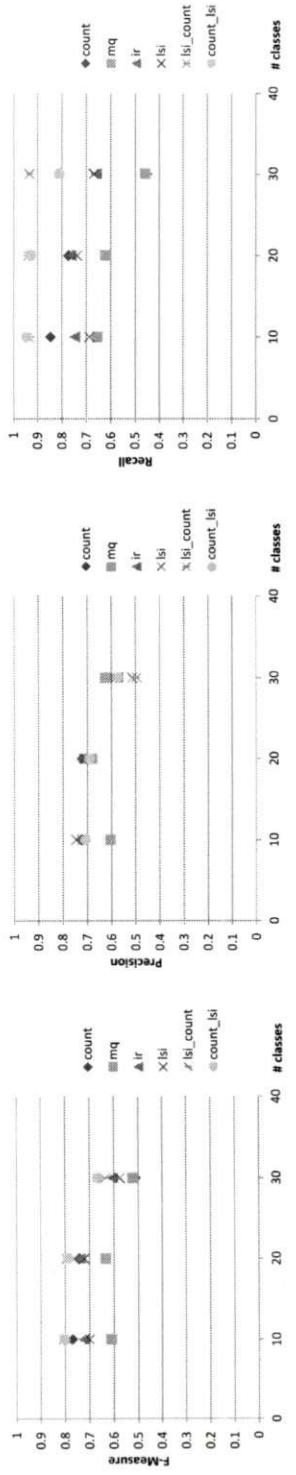


Figure D.24: Measures for large changes: *Design Wizard* low-level view

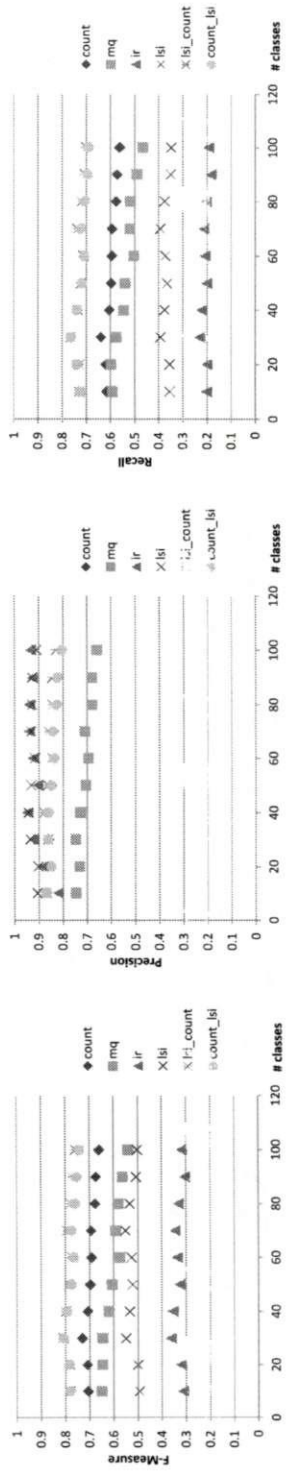


Figure D.25: Measures for large changes: *Design Suite* low-level view

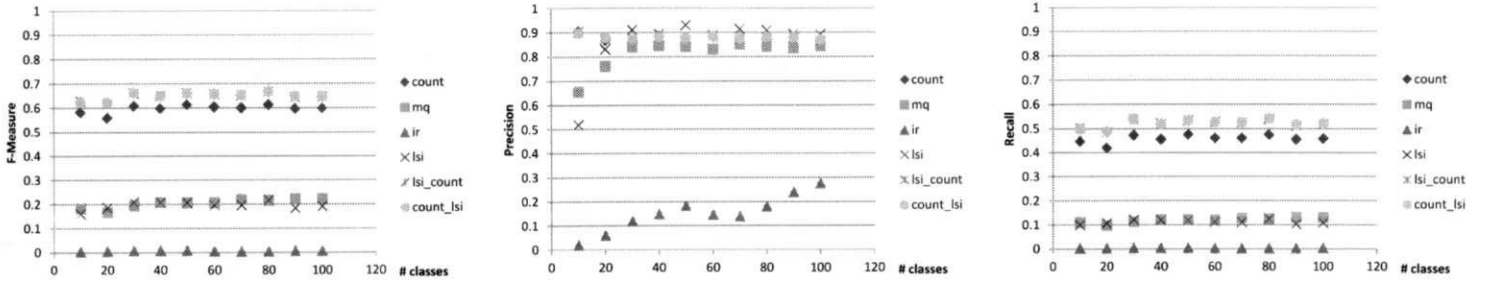


Figure D.26: Measures for large changes: *Mylyn* low-level view

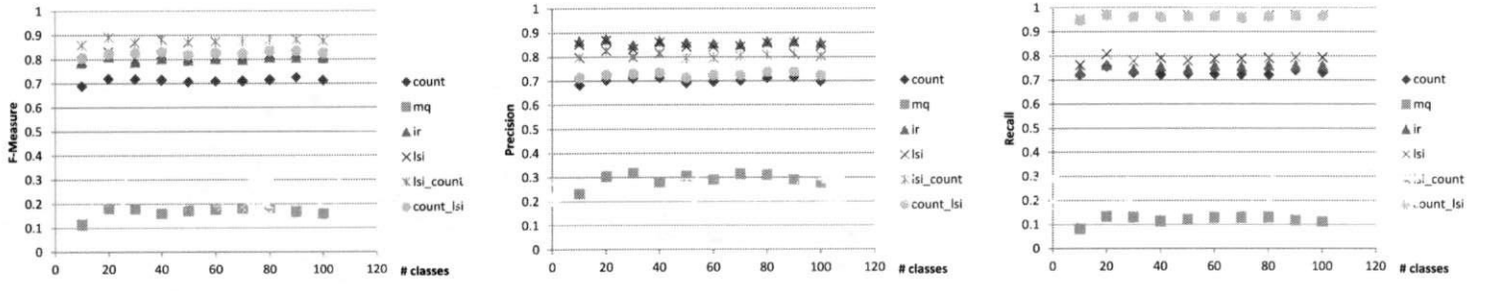


Figure D.27: Measures for large changes: *OurGrid* low-level view

Appendix E

Models and Mapping for the Study on Prioritizing Violations

Modelos e Mapeamentos para o Estudo de Priorização de Violações

Neste Apêndice, as visões modulares de alto nível e o mapeamento das entidades de código-fonte para os módulos nestas visões são detalhados. Para cada sistema, os módulos são descritos, seguidos pelo mapeamento na forma de expressões regulares. Finalmente, as relações autorizadas entre módulos são expressas como relações binárias entre os módulos fonte e destino.

The high-level module views and the mapping from source code entities onto the modules in these views are detailed in the following pages. For each system, modules are first described. Then, a mapping from source code entities onto the modules is described by regular expressions. Finally, the authorized relations between modules are expressed as binary relations between a source module and a target module.

E.1 SweetHome3D

```
# modules
sweetHome3DModel
sweetHome3DTools
sweetHome3DPlugin
sweetHome3DViewController
sweetHome3DSwing
sweetHome3DJava3D
sweetHome3DIO
sweetHome3DApplet
sweetHome3DApplication

# mapping
# <high_level_module> <regular_expression>
sweetHome3DModel com.eteks.sweethome3d.model.*
sweetHome3DTools com.eteks.sweethome3d.tools.*
sweetHome3DPlugin com.eteks.sweethome3d.plugin.*
sweetHome3DViewController com.eteks.sweethome3d.viewcontroller.*
sweetHome3DSwing com.eteks.sweethome3d.swing.*
sweetHome3DJava3D com.eteks.sweethome3d.j3d.*
sweetHome3DIO com.eteks.sweethome3d.io.*
sweetHome3DApplet com.eteks.sweethome3d.applet.*
sweetHome3DApplication com.eteks.sweethome3d(?!(.model|.tools|.plugin|.viewcontroller|.j3d|.io|.applet)).*

# relations
# <source_module> <target_module>
sweetHome3DTools sweetHome3DModel
sweetHome3DPlugin sweetHome3DModel
sweetHome3DPlugin sweetHome3DTools
sweetHome3DViewController sweetHome3DModel
sweetHome3DViewController sweetHome3DTools
sweetHome3DViewController sweetHome3DPlugin
sweetHome3DJava3D sweetHome3DModel
sweetHome3DJava3D sweetHome3DTools
sweetHome3DSwing sweetHome3DModel
sweetHome3DSwing sweetHome3DTools
sweetHome3DSwing sweetHome3DPlugin
sweetHome3DSwing sweetHome3DViewController
sweetHome3DSwing sweetHome3DJava3D
sweetHome3DIO sweetHome3DModel
sweetHome3DIO sweetHome3DTools
sweetHome3DApplet sweetHome3DModel
sweetHome3DApplet sweetHome3DTools
sweetHome3DApplet sweetHome3DPlugin
sweetHome3DApplet sweetHome3DViewController
sweetHome3DApplet sweetHome3DJava3D
sweetHome3DApplet sweetHome3DSwing
sweetHome3DApplet sweetHome3DIO
sweetHome3DApplication sweetHome3DModel
sweetHome3DApplication sweetHome3DTools
sweetHome3DApplication sweetHome3DPlugin
sweetHome3DApplication sweetHome3DViewController
sweetHome3DApplication sweetHome3DJava3D
sweetHome3DApplication sweetHome3DSwing
sweetHome3DApplication sweetHome3DIO
```

E.2 Ant

```
# modules
optional
compilers
condition
rmic
cvslib
email
repository
taskdefs
listener
types
ant
ant.util
zip
tar
mail
bzip2

# mapping
# <high_level_module> <regular_expression>
optional org.apache.tools.ant.taskdefs.optional.*
compilers org.apache.tools.ant.taskdefs.compilers.*
condition org.apache.tools.ant.taskdefs.condition.*
rmic org.apache.tools.ant.taskdefs.rmic.*
cvslib org.apache.tools.ant.taskdefs.cvslib.*
email org.apache.tools.ant.taskdefs.email.*
repository org.apache.tools.ant.taskdefs.repository.*
taskdefs org.apache.tools.ant.taskdefs(?!(.optional|.compilers|.condition|.rmic|.cvslib|.email|.repository)).*
listener org.apache.tools.ant.listener.*
types org.apache.tools.ant.types.*
ant org.apache.tools.ant(?!(.taskdefs|.listener|.types|.util|.filters|.helper|.input|.launch|.loader|.dispatch|.property)).*
ant.util org.apache.tools.ant.util.*
zip org.apache.tools.zip.*
tar org.apache.tools.tar.*
mail org.apache.tools.mail.*
bzip2 org.apache.tools.bzip2.*

# relations
# <source_module> <target_module>
optional taskdefs
optional listener
optional types
optional ant
optional ant.util
optional zip
optional tar
optional mail
optional bzip2
compilers taskdefs
compilers listener
compilers types
compilers ant
compilers ant.util
compilers zip
compilers tar
compilers mail
compilers bzip2
condition taskdefs
condition listener
condition types
condition ant
condition ant.util
condition zip
condition tar
condition mail
condition bzip2
rmic taskdefs
rmic listener
rmic types
rmic ant
rmic ant.util
rmic zip
rmic tar
rmic mail
rmic bzip2
cvslib taskdefs
cvslib listener
cvslib types
cvslib ant
cvslib ant.util
cvslib zip
cvslib tar
cvslib mail
cvslib bzip2
email taskdefs
email listener
email types
email ant
email ant.util
email zip
email tar
email mail
email bzip2
```

```
repository taskdefs
repository listener
repository types
repository ant
repository ant.util
repository zip
repository tar
repository mail
repository bzip2
taskdefs listener
taskdefs types
taskdefs ant
taskdefs ant.util
taskdefs zip
taskdefs tar
taskdefs mail
taskdefs bzip2
listener ant
listener ant.util
listener zip
listener tar
listener mail
listener bzip2
types ant
types ant.util
types zip
types tar
types mail
types bzip2
ant zip
ant tar
ant mail
ant bzip2
ant.util zip
ant.util tar
ant.util mail
ant.util bzip2
```

E.3 Lucene

```
# modules
queryparser
search
index
store
analysis
util
document

# mapping
# <high_level_module> <regular_expression>
queryparser org.apache.lucene.queryParser.*
search org.apache.lucene.search.*
index org.apache.lucene.index.*
store org.apache.lucene.store.*
analysis org.apache.lucene.(analysis|collation).*
util org.apache.lucene.(util|message).*
document org.apache.lucene.document.*

# relations
# <source_module> <target_module>
queryparser search
queryparser index
queryparser document
queryparser util
search index
search analysis
search document
search util
index store
index analysis
index document
index util
analysis document
analysis util
document util
util document
```

E.4 ArgoUML

```
# modules
application
diagrams
notation
explorer
codeGeneration
reverseEngineering
persistence
profile
help
moduleLoader
gui
model
internationalization
taskManagement
configuration
swingExtensions
ocl
critics
javaCodeGeneration

# mapping
# <high_level_module> <regular_expression>
application org.argouml.application.*
diagrams org.argouml.uml.diagram.*
notation org.argouml.notation.*
explorer org.argouml.ui.explorer.*
codeGeneration org.argouml.language.*
reverseEngineering org.argouml.uml.reveng.*
persistence org.argouml.persistence.*
profile org.argouml.profile.*
help org.argouml.help.*
moduleLoader org.argouml.moduleloader|org.argouml.application.modules|org.argouml.application.api
gui org.argouml.ui.*
model org.argouml.model.*
internationalization org.argouml.i18n.*
taskManagement org.argouml.taskmgmt.*
configuration org.argouml.configuration.*
swingExtensions org.argouml.swingext.*
ocl org.argouml.ocl.*
critics org.argouml.cognitive.*
javaCodeGeneration org.argouml.language.java.*

# relations
# <source_module> <target_module>
application diagrams
application notation
application explorer
application codeGeneration
application reverseEngineering
application persistence
application profile
application help
application moduleLoader
application gui
application model
application internationalization
application taskManagement
application configuration
application swingExtensions
application ocl
application critics
application javaCodeGeneration
diagrams notation
diagrams gui
diagrams model
diagrams internationalization
diagrams taskManagement
diagrams configuration
diagrams swingExtensions
notation model
notation internationalization
notation taskManagement
notation configuration
notation swingExtensions
explorer gui
explorer model
explorer internationalization
explorer taskManagement
explorer configuration
explorer swingExtensions
codeGeneration moduleLoader
codeGeneration model
codeGeneration internationalization
codeGeneration taskManagement
codeGeneration configuration
codeGeneration swingExtensions
reverseEngineering model
reverseEngineering internationalization
reverseEngineering taskManagement
reverseEngineering configuration
reverseEngineering swingExtensions
```

```
persistence model
persistence internationalization
persistence taskManagement
persistence configuration
persistence swingExtensions
profile model
profile internationalization
profile taskManagement
profile configuration
profile swingExtensions
help model
help internationalization
help taskManagement
help configuration
help swingExtensions
moduleLoader model
moduleLoader internationalization
moduleLoader taskManagement
moduleLoader configuration
moduleLoader swingExtensions
gui internationalization
gui taskManagement
gui configuration
gui swingExtensions
javaCodeGeneration codeGeneration
javaCodeGeneration reverseEngineering
javaCodeGeneration moduleLoader
javaCodeGeneration model
ocl moduleLoader
ocl model
critics moduleLoader
critics model
```

Bibliography

[Aldrich et al. 2002] Aldrich, J., Chambers, C., and Notkin, D. (2002). Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA. ACM.

[Anquetil et al. 1999] Anquetil, N., Fourrier, C., and Lethbridge, T. C. (1999). Experiments with clustering as a software remodularization method. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 235, Washington, DC, USA. IEEE Computer Society.

[Antoniol et al. 1999] Antoniol, G., Potrich, A., Tonella, P., and Fiutem, R. (1999). Evolving object oriented design to improve code traceability. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, page 151, Washington, DC, USA. IEEE Computer Society.

[Anvik et al. 2006] Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370, New York, NY, USA. ACM.

[Armstrong and Trudeau 1998] Armstrong, M. N. and Trudeau, C. (1998). Evaluating architectural extractors. In *WCRE '98: Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, page 30, Washington, DC, USA. IEEE Computer Society.

[Bass et al. 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Bird et al. 2006] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A. (2006). Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 137–143, New York, NY, USA. ACM.

[Bittencourt et al. 2009] Bittencourt, R. A., Damásio, J. F., Santos, G. J. S., de Almeida Filho, A. T., da Nóbrega, J. M. F., de Figueiredo, J. C. A., and Guer-

rero, D. D. S. (2009). Design suite: Towards an open scientific investigation environment for software architecture recovery. In *SBQS 2009: Anais do VIII Simpósio Brasileiro de Qualidade de Software*.

[Bittencourt and Guerrero 2009] Bittencourt, R. A. and Guerrero, D. D. S. (2009). Comparison of graph clustering algorithms for recovering software architecture module views. In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 251–254, Washington, DC, USA. IEEE Computer Society.

[Bittencourt et al. 2012] Bittencourt, R. A., Guerrero, D. D. S., and Murphy, G. C. (2012). Prioritizing violation warnings in architecture checkers from software history. Submitted for publication.

[Bittencourt et al. 2010] Bittencourt, R. A., Santos, G. J. S., Guerrero, D. D. S., and Murphy, G. C. (2010). Improving automated mapping in reflexion models using information retrieval techniques. In *WCRE 10: Proceedings of the 2010 Working Conference on Reverse Engineering*, pages 163–172, Los Alamitos, CA, USA. IEEE Computer Society.

[Boogerd and Moonen 2006] Boogerd, C. and Moonen, L. (2006). Prioritizing software inspection results using static profiling. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 149–160, Washington, DC, USA. IEEE Computer Society.

[Bourquin and Keller 2007] Bourquin, F. and Keller, R. K. (2007). High-impact refactoring based on architecture violations. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 149–158, Washington, DC, USA. IEEE Computer Society.

[Brooks 1995] Brooks, Jr., F. P. (1995). The mythical man-month: After 20 years. *IEEE Softw.*, 12(5):57–60.

[Browning 2001] Browning, T. (2001). Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *Engineering Management, IEEE Transactions on*, 48(3):292–306.

[Brunet et al. 2009] Brunet, J., Guerrero, D., and Figueiredo, J. (2009). Design tests: An approach to programmatically check your code against design rules. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 255–258.

[Brunet et al. 2011] Brunet, J., Serey, D., and Figueiredo, J. (2011). Structural conformance checking with design tests: An evaluation of usability and scalability. In

ICSM '11: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'11).

[Chen et al. 1990] Chen, Y.-F., Nishimoto, M. Y., and Ramamoorthy, C. V. (1990). The c information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334.

[Christl et al. 2005] Christl, A., Koschke, R., and Storey, M.-A. (2005). Equipping the reflexion method with automated clustering. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 89–98, Washington, DC, USA.

[Christl et al. 2007] Christl, A., Koschke, R., and Storey, M.-A. (2007). Automated clustering to support the reflexion method. *Inf. Softw. Technol.*, 49(3):255–274.

[Clements et al. 2002] Clements, P., Garlan, D. Bass, L., Stafford, J., Nord, R., Ivers, J., and Little, R. (2002). *Documenting Software Architectures: Views and Beyond.* Pearson Education.

[Cohen 1988] Cohen, J. (1988). *Statistical power analysis for the behavioral sciences.* Lawrence Erlbaum.

[da Silva et al. 2006] da Silva, I. A., Chen, P. H., Van der Westhuizen, C., Ripley, R. M., and van der Hoek, A. (2006). Lighthouse: coordination through emerging design. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 11–15, New York, NY, USA. ACM.

[de Silva and Balasubramaniam 2012] de Silva, L. and Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132 – 151. <ce:title>Dynamic Analysis and Testing of Embedded Software</ce:title>.

[Deerwester et al. 1990] Deerwester, S., Dumais, S., Furnas, G., Landauer, T., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407.

[Diehl 2007] Diehl, S. (2007). *Software visualization: visualizing the structure, behaviour, and evolution of software.* Springer Verlag.

[Doval et al. 1999] Doval, D., Mancoridis, S., and Mitchell, B. S. (1999). Automatic clustering of software systems using a genetic algorithm. In *STEP '99: Proceedings of the Software Technology and Engineering Practice*, page 73, Washington, DC, USA. IEEE Computer Society.

[Ebert et al. 2002] Ebert, J., Kullbach, B., Riediger, V., and Winter, A. (2002). Gupro - generic understanding of programs: An overview. *Electronic Notes in Theoretical Computer Science*, 72(2):47–56.

[Eichberg et al. 2008] Eichberg, M., Kloppenburg, S., Klose, K., and Mezini, M. (2008). Defining and continuous checking of structural program dependencies. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 391–400, New York, NY, USA. ACM.

[Eick et al. 2001] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A. (2001). Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12.

[Feilkas et al. 2009] Feilkas, M., Ratiu, D., and Jurgens, E. (2009). The loss of architectural knowledge during system evolution: An industrial case study. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 188 –197.

[Fiutem and Antoniol 1998] Fiutem, R. and Antoniol, G. (1998). Identifying design-code inconsistencies in object-oriented software: a case study. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 94, Washington, DC, USA. IEEE Computer Society.

[Frenzel et al. 2007] Frenzel, P., Koschke, R., Breu, A. P. J., and Angstmann, K. (2007). Extending the reflexion method for consolidating software variants into product lines. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 160–169, Washington, DC, USA. IEEE Computer Society.

[Fritz et al. 2010] Fritz, T., Ou, J., Murphy, G. C., and Murphy-Hill, E. (2010). A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 385–394, New York, NY, USA. ACM.

[Gansner and North 2000] Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233.

[Garlan 2000] Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 91–101, New York, NY, USA. ACM.

[Garlan and Perry 1995] Garlan, D. and Perry, D. E. (1995). Introduction to the special issue on software architecture. *IEEE Trans. Softw. Eng.*, 21(4):269–274.

[Girvan and Newman 2002] Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *Proceedings Of The National Academy Of Sciences Of The United States Of America*, 99(12):7821–7826.

[Gutierrez Fernandez 1998] Gutierrez Fernandez, C. I. (1998). Integration analysis of product architecture to support effective team co-location. Master's thesis, Massachusetts Institute of Technology.

[Hartigan and Wong 1979] Hartigan, J. A. and Wong, M. A. (1979). A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108.

[Hattori and Lanza 2008] Hattori, L. and Lanza, M. (2008). On the nature of commits. In *23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops, 2008. ASE Workshops 2008*, pages 63–71.

[Hochstein and Lindvall 2003] Hochstein, L. and Lindvall, M. (2003). Diagnosing architectural degeneration. In *28th Annual NASA Goddard Software Engineering Workshop*.

[Hochstein and Lindvall 2005] Hochstein, L. and Lindvall, M. (2005). Combating architectural degeneration: a survey. *Information And Software Technology*, 47(10):643–656.

[Hofmeister et al. 2000] Hofmeister, C., Nord, R., and Soni, D. (2000). *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Holt et al. 2006] Holt, R. C., Schürr, A., Sim, S. E., and Winter, A. (2006). Gxl: a graph-based standard exchange format for reengineering. *Sci. Comput. Program.*, 60(2):149–170.

[Holt et al. 2000] Holt, R. C., Winter, A., and Schürr, A. (2000). Gxl: Toward a standard exchange format. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 162, Washington, DC, USA. IEEE Computer Society.

[Huynh et al. 2008] Huynh, S., Cai, Y., Song, Y., and Sullivan, K. (2008). Automatic modularity conformance checking. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 411–420, New York, NY, USA. ACM.

[Jain et al. 1999] Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323.

[Kazman et al. 2001] Kazman, R., O'Brien, L., and Verhoef, C. (2001). Architecture reconstruction guidelines. Technical Report CMU-SEI-2001-TR-026, Carnegie Mellon Software Engineering Institute.

[Kim and Ernst 2007a] Kim, S. and Ernst, M. D. (2007a). Prioritizing warning cate-

gories by analyzing software history. In *Proc. of Int'l Workshop on Mining Software Repositories (MSR 2007*, page 27.

[Kim and Ernst 2007b] Kim, S. and Ernst, M. D. (2007b). Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 45–54, New York, NY, USA. ACM.

[Kim et al. 2006] Kim, S., Pan, K., and Whitehead, Jr., E. E. J. (2006). Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 35–45, New York, NY, USA. ACM.

[Knodel et al. 2008a] Knodel, J., Muthig, D., Haury, U., and Meier, G. (2008a). Architecture compliance checking - experiences from successful technology transfer to industry. In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 43–52, Washington, DC, USA. IEEE Computer Society.

[Knodel et al. 2006] Knodel, J., Muthig, D., Naab, M., and Lindvall, M. (2006). Static evaluation of software architectures. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 279–294, Washington, DC, USA. IEEE Computer Society.

[Knodel et al. 2008b] Knodel, J., Muthig, D., and Rost, D. (2008b). Constructive architecture compliance checking - an experiment on support by live feedback. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 287–296.

[Knodel and Popescu 2007] Knodel, J. and Popescu, D. (2007). A comparison of static architecture compliance checking approaches. In *WICSA '07: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, page 12, Washington, DC, USA. IEEE Computer Society.

[Korn et al. 1999] Korn, J., Chen, Y.-F., and Koutsofios, E. (1999). Chava: Reverse engineering and tracking of java applets. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 314, Washington, DC, USA. IEEE Computer Society.

[Koschke and Eisenbarth 2000] Koschke, R. and Eisenbarth, T. (2000). A framework for experimental evaluation of clustering techniques. In *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, page 201, Washington, DC, USA. IEEE Computer Society.

[Koschke and Simon 2003] Koschke, R. and Simon, D. (2003). Hierarchical reflexion models. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 36, Washington, DC, USA. IEEE Computer Society.

[Kremenek and Engler 2003] Kremenek, T. and Engler, D. (2003). Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 295–315, Berlin, Heidelberg. Springer-Verlag.

[Lattix, Inc. 2008] Lattix, Inc. (2008). Lattix, Inc. Website. `http://www.lattix.com/`.

[Le Gear et al. 2005] Le Gear, A., Buckley, J., Collins, J., and O'Dea, K. (2005). Software reconnexion: understanding software using a variation on software reconnaissance and reflexion modelling. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.–.

[Lehman et al. 1997] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA. IEEE Computer Society.

[Lilienthal 2009] Lilienthal, C. (2009). Architectural complexity of large-scale software systems. In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 17–26, Washington, DC, USA. IEEE Computer Society.

[Lindvall and Muthig 2008] Lindvall, M. and Muthig, D. (2008). Bridging the software architecture gap. *Computer*, 41(6):98–101.

[Lopez-Fernandez et al. 2004] Lopez-Fernandez, L., Robles, G., Gonzalez-Barahona, J., et al. (2004). Applying social network analysis to the information in cvs repositories. In *International Workshop on Mining Software Repositories*. Citeseer.

[Lundberg and Löwe 2003] Lundberg, J. and Löwe, W. (2003). Architecture recovery by semi-automatic component identification. *Electronic Notes in Theoretical Computer Science*, 82(5):98–114.

[Madhavji et al. 2006] Madhavji, N. H., Fernandez-Ramil, J., and Perry, D. (2006). *Software Evolution and Feedback: Theory and Practice*. John Wiley & Sons.

[Mancoridis et al. 1999] Mancoridis, S., Mitchell, B., Chen, Y., and Gansner, E. (1999). Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 50–59.

[Mancoridis et al. 1998] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., and Gansner, E. R. (1998). Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 45, Washington, DC, USA. IEEE Computer Society.

[Manning et al. 2008] Manning, C. D., Raghavan, P., and Schtze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

[Maqbool and Babri 2004] Maqbool, O. and Babri, H. A. (2004). The weighted combined algorithm: A linkage algorithm for software clustering. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 15, Washington, DC, USA. IEEE Computer Society.

[Medvidovic et al. 2007] Medvidovic, N., Dashofy, E. M., and Taylor, R. N. (2007). Moving architectural description from under the technology lamppost. *Inf. Softw. Technol.*, 49(1):12–31.

[Medvidovic and Taylor 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93.

[Miller 1956] Miller, G. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81.

[Mitchell and Mancoridis 2001a] Mitchell, B. and Mancoridis, S. (2001a). Craft: a framework for evaluating software clustering results in the absence of benchmark decompositions [clustering results analysis framework and tools]. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 93–102.

[Mitchell and Mancoridis 2006] Mitchell, B. and Mancoridis, S. (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.*, 32(3):193–208.

[Mitchell and Mancoridis 2001b] Mitchell, B. S. and Mancoridis, S. (2001b). Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 744, Washington, DC, USA. IEEE Computer Society.

[Müller et al. 1993] Müller, H. A., Orgun, M. A., Tilley, S. R., and Uhl, J. S. (1993). A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5(4):181–204.

[Murphy 1996] Murphy, G. C. (1996). *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington.

[Murphy and Notkin 1997] Murphy, G. C. and Notkin, D. (1997). Reengineering with reflexion models: A case study. *Computer*, 30(8):29–36.

[Murphy et al. 1995] Murphy, G. C., Notkin, D., and Sullivan, K. (1995). Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, New York, NY, USA. ACM.

[Murphy et al. 2001] Murphy, G. C., Notkin, D., and Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380.

[Murta et al. 2006] Murta, L. G. P., van der Hoek, A., and Werner, C. M. L. (2006). Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 135–144, Washington, DC, USA. IEEE Computer Society.

[Myers 2003] Myers, C. R. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E*, 68:046116.

[Nagappan and Ball 2005] Nagappan, N. and Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering* ICSE '05, pages 284–292, New York, NY, USA. ACM.

[Nistor et al. 2005] Nistor, E. C., Erenkrantz, J. R., Hendrickson, S. A., and van der Hoek, A. (2005). Archevol: versioning architectural-implementation relationships. In *SCM '05: Proceedings of the 12th international workshop on Software configuration management*, pages 99–111, New York, NY, USA. ACM.

[OSGi 2008] OSGi (2008). The OSGi Architecture. `http://www.osgi.org/About/WhatIsOSGi`.

[Parnas 1994] Parnas, D. L. (1994). Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Passos et al. 2010] Passos, L., Terra, R., Valente, M. T., Diniz, R., and das Chagas Mendonca, N. (2010). Static architecture-conformance checking: An illustrative overview. *IEEE Softw.*, 27:82–89.

[Perry and Wolf 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52.

[Pires et al. 2008] Pires, W., Brunet, Jo a., and Ramalho, F. (2008). Uml-based design test generation. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 735–740, New York, NY, USA. ACM.

[Pollet et al. 2007] Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cimpan, S., and Verjus, H. (2007). Towards a process-oriented software architecture reconstruction taxonomy. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 137–148, Washington, DC.

[Postma 2003] Postma, A. (2003). A method for module architecture verification and its application on a large component-based system. *Inf. Softw. Technol.*, 45(4):171–194.

[Raza et al. 2006] Raza, A., Vogel, G., and Plodereder, E. (2006). Bauhaus - a tool suite for program analysis and reverse engineering. *Reliable Software Technologies - Ada - Europe 2006, Proceedings*, 4006:71–82.

[Rosik et al. 2008] Rosik, J., Le Gear, A., Buckley, J., and Ali Babar, M. (2008). An industrial case study of architecture conformance. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 80–89, New York, NY, USA. ACM.

[Ruthruff et al. 2008] Ruthruff, J. R., Penix, J., Morgenthaler, J. D., Elbaum, S., and Rothermel, G. (2008). Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 341–350, New York, NY, USA. ACM.

[Salton et al. 1975] Salton, G., Wong, A., and Yang, C. S. (1975). A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620.

[Sangal et al. 2005] Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. *SIGPLAN Not.*, 40(10):167–176.

[Schwanke 1991] Schwanke, R. W. (1991). An intelligent tool for re-engineering software modularity. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 83–92, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Siff and Reps 1997] Siff, M. and Reps, T. W. (1997). Identifying modules via concept analysis. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 170–179, Washington, DC, USA. IEEE Computer Society.

[Sim et al. 2003] Sim, S., Easterbrook, S., and Holt, R. (2003). Using benchmarking to advance research: a challenge to software engineering. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 74–83.

[Software Architecture Group 2009] Software Architecture Group (2009). SWAG Tools. `http://www.swag.uwaterloo.ca/tools.htm`.

[SourceForge 2010] SourceForge (2010). Find and Develop Open Source Software. `http://sourceforge.net/`.

[Souza et al. 2010] Souza, R., Guerrero, D., and Figueiredo, J. (2010). Modular network models for class dependencies in software. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 278 –281.

[Storey and Muller 1995] Storey, M.-A. D. and Muller, H. A. (1995). Manipulating and documenting software structures using shrimp views. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 275, Washington, DC, USA. IEEE Computer Society.

[Sullivan et al. 2001] Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA. ACM.

[Sutton 2008] Sutton, I. (2008). Software erosion in pictures - Findbugs. `http://www.headwaysoftware.com/blog/2008/11/software-erosion-findbugs/`.

[Terra and Valente 2009] Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Softw. Pract. Exper.*, 39(12):1073–1094.

[The Open Group 2008] The Open Group (2008). Architecture compliance. `http://www.opengroup.org/architecture/togaf8-doc/arch/chap24.html`.

[Thebeau 2001] Thebeau, R. E. (2001). Knowledge management of system interfaces and interactions from product development processes. Master's thesis, Massachusetts Institute of Technology.

[Tvedt et al. 2002] Tvedt, R. T., Costa, P., and Lindvall, M. (2002). Does the code match the design? a process for architecture evaluation. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 393, Washington, DC, USA. IEEE Computer Society.

[Tzerpos 1997] Tzerpos, V. (1997). The orphan adoption problem in architecture maintenance. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 76, Washington, DC, USA. IEEE Computer Society.

[Tzerpos and Holt 1999] Tzerpos, V. and Holt, R. (1999). Mojo: a distance metric for software clusterings. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 187–193.

[Tzerpos and Holt 2000] Tzerpos, V. and Holt, R. C. (2000). On the stability of software clustering algorithms. In *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, page 211, Washington, DC, USA. IEEE Computer Society.

[van der Westhuizen et al. 2006] van der Westhuizen, C., Chen, P. H., and van der Hoek, A. (2006). Emerging design: new roles and uses for abstraction. In *ROA '06: Proceedings of the 2006 international workshop on Role of abstraction in software engineering*, pages 23–28, New York, NY, USA. ACM.

[van Gurp and Bosch 2002] van Gurp, J. and Bosch, J. (2002). Design erosion: problems and causes. *J. Syst. Softw.*, 61(2):105–119.

[van Ommering et al. 2001] van Ommering, R., Krikhaar, R., and Feijs, L. (2001). Languages for formalizing, visualizing and verifying software architectures. *Computer Languages*, 27(1-3):3–18.

[Wen and Tzerpos 2003] Wen, Z. and Tzerpos. V. (2003). An optimal algorithm for mojo distance. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 227, Washington, DC, USA. IEEE Computer Society.

[Wiggerts 1997] Wiggerts, T. A. (1997). Using clustering algorithms in legacy systems remodularization. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, USA. IEEE Computer Society.

[Williams and Hollingsworth 2005] Williams, C. C. and Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31:466–480.

[Wu et al. 2005] Wu, J., Hassan, A. E., and Holt, R. C. (2005). Comparison of clustering algorithms in the context of software evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 525–535, Washington, DC, USA. IEEE Computer Society.

[Zhang et al. 2004] Zhang, X., Young, M., and Lasseter, J. H. E. F. (2004). Refining code-design mapping with flow analysis. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 231–240, New York, NY, USA. ACM.