Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Modelagem Estatística de Mudanças Estruturais para Simulação de Evolução de Software

Jemerson Figueiredo Damásio

Campina Grande, Paraíba, Brasil

Agosto de 2011

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Modelagem Estatística de Mudanças Estruturais para Simulação de Evolução de Software

Jemerson Figueiredo Damásio

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Evolução de Software

Jorge Figueiredo e Dalton Serey

(Orientadores)

Campina Grande, Paraíba, Brasil

D155m Damasio, Jemerson Figueiredo
Modelagem estatistica de mudancas estruturais para simulacao de evolucao de software / Jemerson Figueiredo Damasio. - Campina Grande, 2011.
66 f.

Dissertacao (Mestrado em Ciencia da Computacao) - Universidade Federal de Campina Grande, Centro de Engenharia Eletrica e Informatica.

1. Mudancas Estruturais 2. Evolucao de Software 3. Estatistica Experimental 4. Dissertacao I. Figueiredo, Jorge Cesar Abrantes de II. Guerrero, Dalton Dario Serey III. Universidade Federal de Campina Grande - Campina Grande (PB)

CDU 004.65(043)

"MODELAGEM ESTATÍSTICA DE MUDANÇAS ESTRUTURAIS PARA SIMULAÇÃO DE EVOLUÇÃO DE SOFTWARE"

JEMERSON FIGUEIREDO DAMÁSIO

DISSERTAÇÃO APROVADA EM 30.08.2011

JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador(a)

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

TIAGO LIMA MASSONI, Dr.
Examinador(a)

UIRA KULESZA, Dr.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

O cenário recente das pesquisas em Ciência da Computação está passando por uma transformação significativa. A vertente estatística, tipicamente presente em estudos científicos das mais diversas áreas do conhecimento humano, não obstante era negligenciada por pesquisadores de nossa área.

Na Engenharia de Software há, ainda, pouquíssimos resultados que apresentam validação estatística adequada. Isto se dá por diversos fatores, dentre os quais se evidencia a escassez de amostras. Contudo, este problema não é exclusivo nosso, e estratégias que visam minimizá-lo são bem conhecidas, dentre as quais se destaca a modelagem. Modelos, por definição, possuem limitações mas permitem o exercício de cenários simulados, possibilitando, eventualmente, validações estatísticas.

Em particular no campo da evolução de software, modelos de mudanças são conhecidos. Porém, quando se trata de nichos específicos que visam as mudanças de granularidade pequena (envolvendo as menores entidades e relacionamentos do software), há pouco ou nenhum aporte teórico de suporte na literatura acadêmica. Ou seja, não há hoje um modelo teórico que suporte, por exemplo, como se deve evoluir o número de classes e métodos de um software Java de forma realista. Assim, retomando o contexto das pesquisas em Engenharia de Software e sua escassez de amostras, assim como os insuficientes modelos de mudanças estruturais, temos como resultado uma série de trabalhos de nossa área que evidenciam tais mudanças sendo realizadas de forma pouco criteriosa (*ad hoc*), e com todas as limitações que isto impõe.

Frente a esta problemática, este trabalho mostra-se como o primeiro esforço em conceber modelos estatísticos formais das mudanças estruturais de software, bem como sua aplicação através de simulação. Em particular, os modelos derivam da análise estatística de uma gama de dados oriundos de mudanças estruturais reais em softwares *open source* Java. A aplicação e avaliação dos modelos dão-se através do seu uso em um simulador de versões de software, concretizado para fins desta pesquisa.

Os resultados da formalização do modelo e de sua experimentação através do simulador trazem à tona uma série de resultados novos, e demonstram boa adequação da abordagem ao problema apresentado.

Abstract

The scenario of recent research in computer science is undergoing a significant transformation. Statistical approaches, typically present in scientific studies from various fields of human knowledge, used to be neglected by researchers of our area.

In software engineering, there are still few results that have adequate statistical validation. This happens by several factors, among which we may evidence the lack of samples. However, this problem is not exclusive of us, and strategies to minimize them are well known, among which stands out modeling. Models, by definition, have limitations, but allow the exercise of simulated scenarios, allowing eventually statistical validation.

Particularly in the field of software evolution, models of change are known. But when it comes to specific niches like fine-grained changes (involving smaller entities and relationships of the software), there is little or no theoretical basis of support in the academic literature. That is, there is no theoretical model that supports, for example, how to evolve the number of classes and methods of a Java software in a realistic way. So, returning to the context of Software Engineering research and its lack of samples, as well as the insufficient structural changes models, the results emerge in a series of papers that show such changes being performed rather indiscriminate (ad hoc), and with all the limitations it imposes.

Facing with this problem, this work shows up as the first effort to devise formal statistical models of the software structural changes and its application through simulation. In particular, models derived from the statistical analysis of data from a range of real structural changes of open source Java software. The application and evaluation of models takes place through its use in a simulator of software version implemented for this research.

The results of the formal model and experimentation via simulator bring up a series of new results, and show good adequacy of the approach to the problem.

Agradecimentos

Agradeço:

À minha família, que durante este mestrado fortaleceu-se diante das agruras da vida;

À minha namorada, Roberta, minha fonte de tranqüilidade;

Aos meus orientadores, Jorge Abrantes e Dalton Serey, que me deram foco e me ajudaram a concretizar este trabalho;

Aos meus colegas de laboratório, que me permitiram trabalhar num ambiente extremamente amigável e respeitoso;

À COPIN e seus funcionários;

À população brasileira, por financiar meus estudos através do programa CAPES - Bolsas de Demanda Social.

Aos meus amigos, promotores de festas e farras, sem as quais eu seria menos feliz.

Conteúdo

Lista de Símbolos

API – *Application Programming Interface*

AST – *Abstract Syntax Trees*

CVS – *Concurrent Version System*

PL – *Power-law*

ML – *Modelo Linear*

QA – *Questão de Avaliação*

RLE – *Redes Livres de Escala*

SES – *Statistical Evolution Simulator*

SVN – *Subversion*

Lista de Figuras

pode ser considerado um expoente no assunto. Além desse, diversos outros trabalhos que visam a simulação da evolução de software com modelos baseados em resultados empíricos se destacam, como os de Pereira *et al.* [PAT2009], Travassos *et al.* [TSM+2008], Zang *et al.* [ZHK+2006], Zang *et al.* [ZZK+2008], Ajila e Alam [AA2009], Araújo [Ara2009], dentre outros.

Por razões diversas, nem sempre os modelos conhecidos atendem a necessidades específicas. Por exemplo, hoje são conhecidos nichos que demandam uma melhor noção de como se dão as mudanças das menores estruturas que compõem o software (aqui denominadas estruturas de **granularidade pequena**), a exemplos de classes e métodos em código OO. Isto se torna crítico quando não há aporte teórico que auxilie na construção de tais modelos, pois hoje é impossível responder, mesmo que de forma mínima, questões como: *se eu pretendesse simular uma atividade de* commit, *quantas classes tipicamente eu deveria adicionar? E métodos?* Ou, *Este programador removeu 10 classes de uma vez, isto é normal no projeto?*

1.2. O Problema

Motivados por um trabalho de doutorado no contexto de recuperação arquitetural realizado por um membro do nosso grupo de pesquisa, detectou-se a necessidade de simular a evolução estrutural de granularidade pequena de softwares. Levantamento bibliográfico posterior revelou que uma série de outros trabalhos acadêmicos apresentam falhas metodológicas no que se refere à forma como os softwares tem sua estrutura modificada para fins de experimentação. Nestes trabalhos, a estrutura é normalmente evoluída de forma pouco criteriosa, baseada tão somente no bom senso do pesquisador, a exemplos de [TH2000, KL2006] na área de recuperação arquitetural, e [RT2001, PR2009] em análise de impacto.

Um caso particular evidencia a relevância desta discussão. No âmbito da recuperação arquitetural, algumas métricas, tal qual a métrica de estabilidade [TH2000], são utilizadas para comparar o novo algoritmo de recuperação aos já existentes. Esta métrica é simples: dados dois algoritmos A e B de recuperação arquitetural, ela visa estabelecer qual apresenta maior estabilidade quando o **software muda**. É possível que um dos algoritmos (A, digamos) apresente desempenho excepcional para a recuperação arquitetural em uma dada versão do software, porém, após uma alteração mínima na estrutura original, uma nova recuperação arquitetural se mostre desastrosa. Esta

instabilidade do algoritmo A é indesejável. Portanto, se o algoritmo B apresentasse maior estabilidade após mudanças de versão do software, ele seria mais interessante. O mais importante desta discussão é que tanto no trabalho que define a métrica quanto no trabalho de Kazem e Lofti [KL2006] as mudanças estruturais sem critério (realizadas de forma *ad hoc*) prevalecem.

Uma experimentação adequada de um novo algoritmo (do ponto de vista desta métrica) demandaria, pelo menos: 1) Mudanças de tamanhos variados para a estrutura original; 2) Para cada tamanho de mudança, uma série de repetições com mudanças estruturais distintas; por fim, 3) a repetição dos procedimentos 1 e 2 para diversos softwares. Dada a complexidade de se obter amostras reais de mudanças que atendessem estas necessidades, far-se-ia necessária uma experimentação *in vitro* (simulação de amostras de mudanças estruturais que satisfaçam estes requisitos), o que não era possível, visto que o modelo de mudanças estruturais de granularidade pequena necessário para tal não existia até então.

1.3. A Solução

Objetiva-se modelar o comportamento típico das mudanças estruturais de granularidade pequena (*i.e.*: em termos de entidades de código e seus relacionamentos) em termos de um **modelo puramente quantitativo de evolução de software**, e aplicar tais modelos para fins de simulação, visando atender, dentro das limitações, o problema descrito.

1.4. O Método

O trabalho de modelagem das mudanças estruturais partiu de uma premissa composta: **mudanças pequenas ocorrem mais frequentemente que mudanças grandes. Além disto, quanto maior uma mudança, mais rara ela tende a ser**. Esta observação pode ser traduzida visualmente num histórico de mudanças hipotético, tal qual apresentada na Figura 1. Nessa figura o eixo das abcissas pode representar, por exemplo, a quantidade de classes adicionadas em um *commit*, enquanto o eixo das ordenadas representa o número de vezes em que esta ação ocorreu na vida do software. Assim, o ponto (40, 1) indicaria que 40 vezes houve *commits* de apenas 1 classe, o ponto (30, 2) indicaria que 30 vezes houve *commits* de apenas 2 classe, etc.

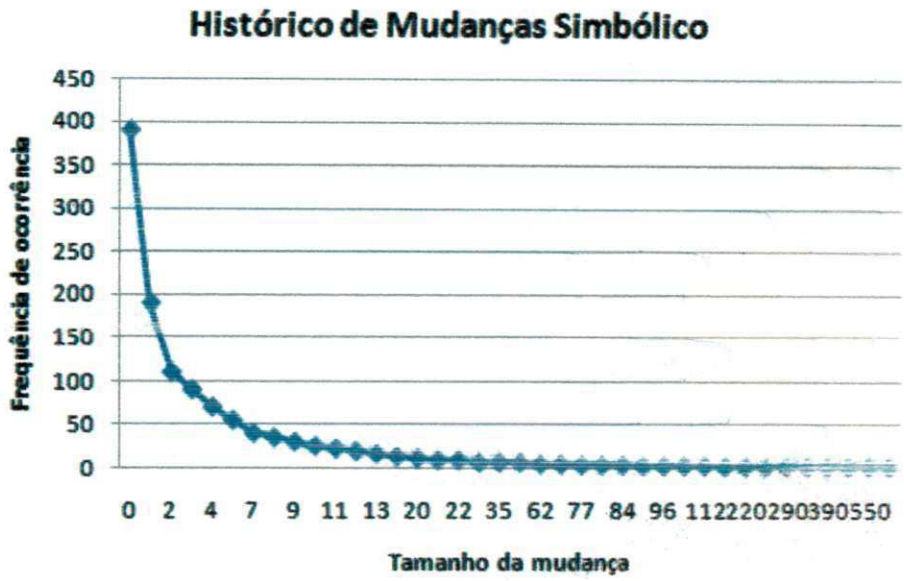


Figura 1 – Representação gráfica da intuição para mudanças estruturais.

A curva apresenta uma tendência visual. Com efeito, a esta hipótese inicial demos o nome de **hipótese de *power-law***. Este termo se refere ao fato de, ao menos intuitivamente, as mudanças apresentarem um comportamento similar ao de um conhecida classe de distribuições estatísticas denominada *power-law*, também conhecida por lei de potência.

No que se refere à avaliação dos modelos obtidos, há dois pontos de vista distintos. O primeiro trata dos modelos estatísticos em si. Por exemplo, é sabido que um dos modelos estatísticos concebidos neste trabalho incorpora 97% dos valores amostrais. Poderíamos nos ater a métricas deste tipo e afirmar que em certo aspecto o nosso simulador é 97% realista. O segundo se refere ao modelo em uso, e o impacto prático das simulações realizadas.

No que tange os modelos estatísticos, quando da modelagem os resultados de sua adequação aos dados já emergem de forma clara, tal qual apresentaremos adiante. Assim, restou-nos avaliar a simulação de forma mais ampla, com fins práticos. Em particular, consideramos três perspectivas distintas e complementares, que juntas nos darão uma **percepção** da qualidade de nossa abordagem, como segue em formatos de questões de avaliação (QA):

QA1. A evolução simulada se assemelha, em quantidade de mudanças estruturais, à evolução real?

O objetivo é confirmar que a simulação está suportada por um modelo capaz de predizer estados futuros do software, conforme esperado.

De forma complementar, pretendemos determinar se o modelo estatístico gerado se sobressai à **distribuição uniforme**, na qual todos os eventos possuem igual probabilidade de ocorrência. Esta distribuição foi escolhida para simular o ato de mudar a estrutura do software de forma *ad hoc* (sem um critério específico), tal qual argumentamos ser realizada usualmente. A intuição por trás desta ideia emerge de um cenário fictício, porém plausível, em que um pesquisador opta por simular a evolução de um software de forma *ad hoc*, e imagina seu experimento da seguinte forma: "Na primeira rodada, vou adicionar 2 classes. Na segunda vou mudar, e adicionar 4. Na terceira, vou reduzir e adicionar apenas 1". Ou seja, é provável que, a cada passo, se opte por valores não explorados. De certo, não há fundamentação teórica para esta nossa consideração. No entanto, esta abordagem pode ser interpretada apenas como um modelo quantitativo trivial a ser confrontado, dado que não há outro.

QA2. A evolução simulada se adequa às leis de Lehman?

As propriedades matemáticas que geram os modelos não são as mesmas que definem a adequação às leis de Lehman. É possível que na versão simulada se obtenha um conjunto similar de mudanças à versão real, mas destoantes com relação às particularidades apresentadas por Lehman em suas oito leis. Realizou-se, assim, uma investigação quantitativa neste sentido.

QA3. A evolução simulada preserva as propriedades de rede livre de escala de um software?

Esta questão se refere ao trabalho de Souza [Sou2010], e remete a outra característica estrutural que explica como entidades se relacionam entre si. É possível que a aplicação de mudanças estruturais simuladas divirja das mudanças reais, corrompendo esta organização.

No que se refere à hipótese de modelagem (**hipótese *power-law***), elaborou-se um estudo experimental com 8 softwares *open source* (código aberto) escritos em Java, com vasto histórico de versões. Destes, extraímos os modelos estatísticos de evolução, e o mais preciso foi escolhido como candidato para uma avaliação do processo de

simulação de evolução de software, através da qual as três questões de avaliação foram confrontadas. Esta experimentação foi promovida sob duas abordagens distintas: a **Simulação Numérica** e a **Simulação Aplicada**, conforme explicaremos a seguir.

Na **Simulação Numérica** o objetivo é gerar toda a evolução de um software através de ordens de mudanças estruturais, versão após versão, sem a preocupação de aplicá-las de fato na estrutura pré-existente. A simulação numérica propõe conceber uma evolução completa do software, através da qual seja possível, por exemplo, estimar um crescimento de número de classes, métodos, etc. Para esta abordagem foram realizadas três análises quantitativas que almejavam responder as duas primeiras questões de pesquisa (**QA1**, **QA2**).

No que se refere à **Simulação Aplicada**, visou-se incorporar a evolução simulada, versão após versão, à estrutura pré-existente do software. Esta abordagem aplica o modelo em sua totalidade: as mudanças de versão geradas na Simulação Numérica são lidas uma a uma e concretizadas de fato na estrutura. Neste processo, deparamo-nos com um problema que denominamos **não-aceitação estrutural da mudança**, uma limitação no modelo. De fato, veremos neste trabalho que foi necessário criar um algoritmo de ajuste estrutural para que as mudanças pudessem ser incorporadas, dando-nos subsídios para averiguar a terceira questão de avaliação (**QA3**).

Percebe-se que as avaliações proposta não visam explicitamente verificar se o modelo e o simulador se aplicam de fato dentro do contexto das pesquisas em Engenharia de Software, conforme citado. Porém, realizando os estudos que respondem às QA's, estamos criando um arcabouço teórico para conclusões apropriadas sobre a aplicabilidade do modelo e da simulação no contexto das pesquisas em Engenharia de Software.

1.5. Resultados

No que tange o modelo que suporta a simulação, dentre outras observações, verificou-se que a **hipótese *power-law*** não pode ser descartada como modelo para as mudanças estruturais, mas também não se confirma. Além disto, demonstraremos que esta hipótese, mesmo que validada, por si só não é suficiente para fins de simulação, sendo necessário um modelo que correlacione as unidades estruturais.

Quanto à simulação, a Simulação Numérica apresentou resultados satisfatórios. As versões geradas apresentaram quantidades de estruturas e relacionamentos superiores às reais, mas com curvas de crescimento extremamente similares, e com resultados muito mais satisfatórios que a distribuição uniforme para mudanças *ad hoc*. Quanto às leis de Lehman, as métricas retiradas sugerem concordância parcial de ambas as evoluções (reais e simulada) com seus princípios. Houve discordância, em especial, com a penúltima lei (VII – Qualidade Decrescente), difícil de serem mensuradas adequadamente.

No que se refere às propriedades de redes livres de escala, a evolução simulada tendeu a deteriorar a estrutura do software à medida que o número de versões simuladas cresce. Há uma série de possibilidades para explicar tais resultados, em especial, relacionadas a limitações na amostra e no modelo. Uma discussão apropriada é apresentada ao final deste trabalho.

1.6. Estrutura do Documento

O Capítulo 2 trata da fundamentação teórica, contemplando os trabalhos que suportam esta pesquisa. Sua leitura se faz necessária para uma melhor compreensão deste trabalho.

O Capítulo 3 é dedicado aos mecanismos necessários para a execução dos modelos estatísticos da Simulação Numérica, Simulação Aplicada e o Algoritmo de Acomodação de Mudanças.

O Capítulo 4 explicita a metodologia e os resultados do experimento completo.

O Capítulo 5 explora o estado da arte, evidenciando uma perspectiva histórica da concepção de modelos de evolução de software e seu recente viés de aplicação em simulação.

No Capítulo 6 concluímos o trabalho, evidenciamos as principais contribuições, as limitações e linhas de pesquisas que se abrem a partir deste estudo.

Capítulo 2

Fundamentação Teórica

Este trabalho caracteriza-se por dois aspectos principais: o primeiro se refere à utilização de técnicas estatísticas para fins de modelagem do domínio da evolução de software; o segundo trata da avaliação do simulador que exercita um dos modelos encontrados. A seguir, é apresentada a fundamentação teórica que suporta uma compreensão apropriada destas duas fases. Em particular, no que concerne a modelagem, descrevemos o conjunto de variáveis de interesse e o conjunto de técnicas aplicadas para a modelagem, das quais o trabalho de Clauset *et al.* [CSN2009] se destaca. No que se refere à avaliação da simulação, faz-se necessário detalhar dois trabalhos: os postulados de Lehman, conhecidos como as Leis de Lehman [Leh1996], e as propriedades de redes livres de escala de software proposto por Souza [Sou2010].

2.1. Modelos em Evolução de Software

Um modelo estatístico é definido em termos das variáveis observadas. No domínio da evolução de software queremos definir quais estruturas de um software serão consideradas e sob qual aspecto de mudança. É comum representar a estrutura do software em termos de entidades e relacionamentos. Uma vez que nosso escopo está limitado à linguagem Java, nos interessam seus componentes estruturais mais usuais: **pacotes**, **tipos**, **métodos** e **atributos** como entidades; e ***throws***, ***implements***, ***extends*** e ***method calls*** como relacionamentos. A Figura 2 apresenta um exemplo de código Java contemplando as entidades e relacionamentos apresentados, bem como a visão estrutural equivalente.

O conjunto de entidades e relacionamentos escolhidos possui algumas particularidades que merecem discussão apropriada. Primeiro porque nem todos os

possíveis tipos de entidades em Java foram contemplados. Neste estudo, **tipo** incorpora três abstrações de Java: **classes**, **interfaces** e **classes abstratas**; já **método** incorpora métodos e construtores. Além disso, blocos estáticos não foram considerados. A respeito dos relacionamentos, ***return*** e ***contains*** não foram contabilizados, mas são considerados, pois sua ocorrência possui relação direta com certas entidades (*i.e.*: a adição de um método implica, obrigatoriamente, na adição de um relacionamento do tipo ***return***). Assim, em termos numéricos, a simulação incorpora estes relacionamentos sem a necessidade de um modelo próprio.

Conhecidas as unidades estruturais de interesse, faz-se necessário definir quais operações de mudanças queremos modelar. Considere classes Java quaisquer. Uma série de diferentes operações pode ocorrer sobre tais estruturas, a exemplo de adições, remoções, *renames, moves, joins e splits*. Sendo as mais simples as **adições** e **remoções** (*e.g.*: uma nova classe é adicionada para acrescentar funcionalidades, ou um método obsoleto é puramente removido); as demais são decorrentes de situações de refatoramento, mais complexas, e interpretadas como descontinuidades da evolução [ADPM2004].

```
interface A {
    public B m1();
}
```

```
class B implements A{
    public B m1() {
        return new B();
    }
}
```

```
class C extends Exception{
    static String a = "";
    public C() {
        super(a);
    }
}
```

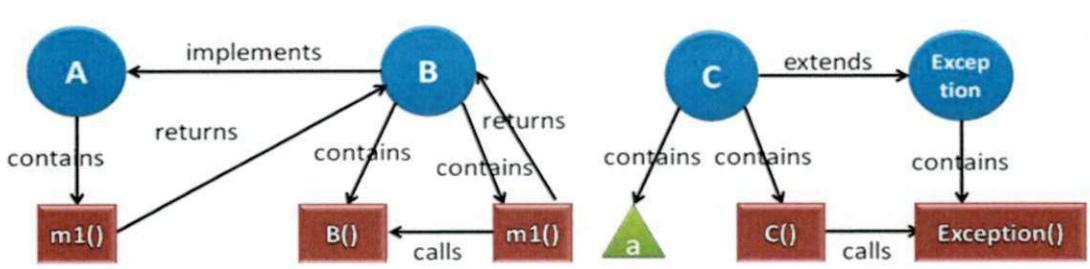


Figura 2 – Exemplo de código Java que exercita as estruturas consideradas e sua representação estrutural.

Os softwares de controle de versões, a exemplo de CVS e SVN, não suportam uma detecção adequada deste segundo tipo de mudanças, representando-as puramente como remoções de estruturas pré-existentes e adição de novas estruturas. Apesar de técnicas para detecção de descontinuidades serem conhecidas [ADPM2004, GT2002,

GT2005], optou-se aqui por modelar as mudanças tal quais suportadas pelos softwares de controle, em termos de adições e remoções. Definimos, portanto, que a forma de mensurar a ocorrência de mudanças estruturais se dá através das seguintes variáveis presentes na **Erro! Fonte de referência não encontrada.**.

Tabela 1 – O conjunto de variáveis que define o modelo de evolução de software. Em (a) as variáveis relativas às entidades. Em (b) as variáveis relativas aos relacionamentos.

Entidades
Adição de Pacotes
Remoção de Pacotes
Adição de Tipos
Remoção de Tipos
Adição de Métodos
Remoção de Métodos
Adição de Atributos
Remoção de Atributos

(a)

Relacionamentos
Adição de *Throws*
Remoção de *Throws*
Adição de *Implements*
Remoção de *Implements*
Adição de *Extends*
Remoção de *Extends*
(Líquido) *Method Calls*

(b)

Como pode ser observado, diferentemente das demais variáveis, as *method calls* não são modeladas em termos de adição e remoção, mas do seu número liquido, versão após versão. Assim, quando há uma mudança de versão, podemos afirmar que houve, por exemplo, a remoção de dois tipos pré-existentes, e adição de três novos tipos. Porém, no que se refere às *method calls*, podemos apenas afirmar que o número total de *method calls* aumentou ou decresceu de certo valor (na sessão 2.2 explicaremos a razão).

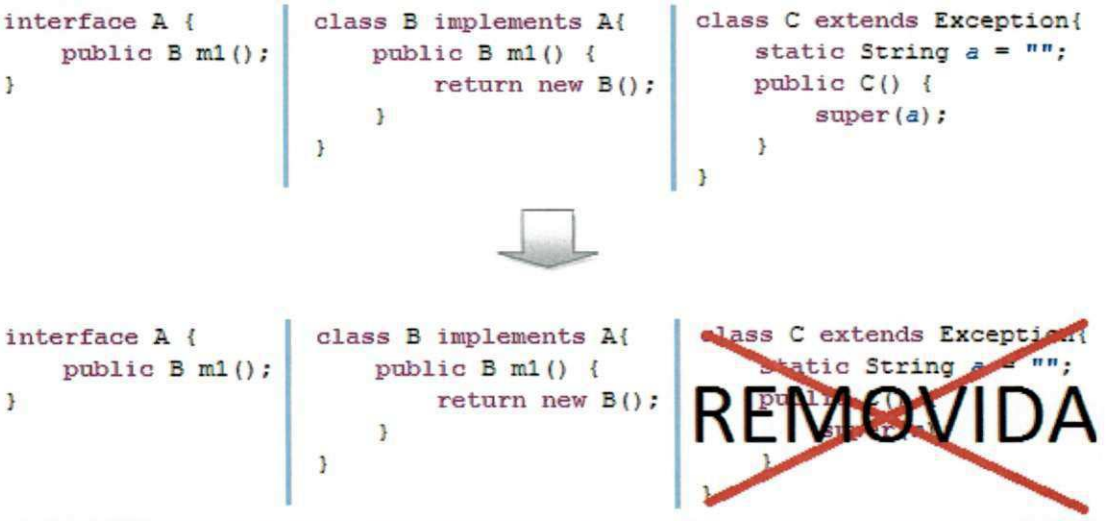
A estratégia de reduzir o número de variáveis a serem observadas deu-se pela dificuldade envolvida em conceber e manipular um modelo mais complexo, especialmente quando se trata de variáveis que apresentam dependência entre si. Uma discussão apropriada em cima deste e outros possíveis caminhos para aprimorar o modelo proposto é apresentada no Capítulo 6.

2.2. A Extração de Dados

Consideremos o conjunto de quinze variáveis que definimos e que regem a modelagem de um software. De forma simples, podemos representar uma mudança de versão como tupla composta de quinze valores, um para cada variável.

Para fins de exemplo, consideraremos uma mudança de versão na qual uma classe e suas estruturas internas foram removidas. A Figura 3 apresenta tal mudança e a tupla que a representa. A extração da informação no formato de tuplas depende de alguns passos anteriores. Primeiro, faz-se necessário transformar o código fonte em um grafo, onde os nós são as entidades, e as arestas, relações. Compará-los e armazenar o resultado desta comparação numa tupla como a apresentada.

Para a atividade de comparação, foi desenvolvido um extrator[1] que faz uso do *plugin* de construção de ASTs (*Abstract Syntax Trees*) do Eclipse[2] e reconstrói design em forma de um grafo. Internamente, o extrator cria um contexto para cada unidade estrutural, tornando-a única. Por exemplo, para criar uma assinatura única para um método basta associá-lo a uma classe e evidenciar seus parâmetros e tipo de retorno. De forma similar, uma relação do tipo *throws* pode ser identificada pela união da assinatura do método que a lança e a do tipo lançado. Esta forma de estruturar os dados nos permitiu realizar a comparação entre duas versões de forma efetiva, havendo a presença de alguma unidade estrutural na primeira versão e não na segunda, há a remoção. Se for o contrário, há a adição. E se estão presentes em ambas as versões, nada é contabilizado.



ADD. PAC.	REM. PAC.	ADD. TIPO	REM. TIPO	ADD. MÉT.	REM. MÉT.	ADD. ATRIB.	REMO. ATRIB.	ADD. *IMPL.*	REM. *IMPL.*	ADD. *EXTE.*	REM. *EXTE.*	ADD. *THRO.*	REMO. *THRO.*	*METH. CALLS*
0	0	0	1	0	1	0	1	0	0	0	1	0	0	-1

Figura 3 – Exemplo de mudança de versão, do qual uma classe completa foi removida. Embaixo, a tupla de mudança que a caracteriza através das variáveis definidas.

[1] Ferramenta disponível no site desta pesquisa [Dam2011], na sessão Extractor.

[2] A IDE Eclipse (http://www.eclipse.org) é um ambiente que provê vasto suporte ao desenvolvimento de aplicações Java.

Este formato mostrou-se adequado para os fins propostos, porém, introduziu um problema no que se refere às *method calls*. A criação de contextos que explicitem métodos chamadores e chamados de uma *method call* mostrou-se uma tarefa árdua e infrutífera. O maior problema decorre das formas diferenciadas que se pode fazer uma chamada de método em Java (e.g.: direta entre métodos via *cast*, reflexão, polimorfismo e tipos implícitos). Algumas técnicas foram experimentadas, mas no fim optou-se por uma forma alternativa de contabilizar as *method calls*, apenas contabilizando quantas há em cada versão, esse sim, fácil de extrair a partir dos números totais de cada versão. Ou seja, temos apenas como saber o número de *method calls* em dada versão. Isto significa que temos apenas como modelar o número líquido de *method calls* (adições menos remoções). Existe a percepção que todas as variáveis poderiam ser consideradas de forma similar (em termos de seu número líquido). Porém, para sustentar a riqueza do modelo, optou-se por manter apenas as *method calls* neste formato e as demais como as apresentamos.

No decorrer do estudo, observou-se que os dados extraídos tipicamente continham *outliers* que enviesavam a modelagem e, posteriormente, introduziam anomalias à simulação. Ao que percebemos, estes pontos excepcionais (caracterizados por tuplas com números bem maiores que o convencional) são frutos de atividades atípicas no repositório de versões, tal qual uma migração entre repositório, a entrada de uma grande versão desenvolvida paralelamente, ou um erro de outra natureza. Desta forma, aplicou-se um filtro de *outliers* para a **remoção de valores que ultrapassassem duas vezes o desvio padrão**. A extração de *outliers* possui um caráter subjetivo que depende da amostra. Neste caso, o filtro aplicado mostrou-se adequado para extração dos pontos excepcionais, o que se refletiu numa melhor simulação do software em questão.

Por fim, no que se refere ao formato da extração, o formato de tuplas de mudanças foi adotado tanto para a sumarização dos dados extraídos dos softwares reais, quanto para o processo de simulação fim deste trabalho. Assim, na extração, cada um dos oito softwares da amostra teve todo seu histórico de mudança sintetizado em um arquivo neste formato. E na simulação, um arquivo similar foi gerado.

α e o *x-min*. O primeiro indica a inclinação da curva, e o segundo, o valor inicial da amostra a partir do qual a distribuição se aplica. Para os *fitting tests*, obedecemos a metodologia proposta por Clauset *et al* [CSN2009] para a investigação de *power-laws*, e posterior análise dos dados frente a outras distribuições para fins de comparação de resultados com a *power-law*. O conjunto de ferramentas necessário a toda esta avaliação é também fornecido pelos autores em formatos de scripts R [R2010] e Matlab [Mat2010] e consiste de três atividades:

1) Primeiro, estima-se os parâmetros da curva de *power-law* (*i.e.*: α e *x-min*) que mais se aproxima dos dados amostrais.

2) Estima-se um valor denominado *p-value* para mensurar quantitativamente a hipótese de *power-law*, ou seja, dizer quão aceitável ela é. Para tal, são gerados 1000 conjuntos de dados aleatórios obedecendo aos parâmetros do passo 1. Sobre cada conjunto, comparam-se os dados aleatórios e os dados amostrais, checando quais se aproximam mais da *power-law* estimada. A cada rodada, se a amostragem se sobressai, o *p-value* é incrementado. De forma complementar, se os dados aleatórios se sobressaem, o *p-value* é decrementado. Assim, um valor entre 0 e 1 é obtido. Segundo os autores, uma abordagem conservadora considera que **valores acima de 0.1 indicam que a hipótese de Power-law não pode ser descartada, e valores abaixo deste indicam que a hipótese de Power-law não se verifica**.

3) O último passo desta abordagem implica em verificar se hipóteses de aderência a outras distribuições similares também se aplicam e qual distribuição se mostra mais adequada para modelar os dados. As seguintes distribuições são comparadas: pareto, exponencial discreta, lognormal e weibull. Também aqui é gerado um valor de *p-value*. **Aqui, quão menor este *p-value*, mais improvável se torna distinguir se alguma distribuição possui vantagem sobre outra**.

Observa-se que na linguagem estatística os posicionamentos não devem ser finais, no sentido que não cabe afirmar que os dados obedecem de fato ao conjunto das *power-laws*, mas apenas que tal hipótese não pode ser descartada.

Modelos de Regressão Linear – Os modelos de regressão linear objetivam modelar uma variável desconhecida em função da ocorrência isolada ou combinada de outras variáveis conhecidas. Diz-se, neste caso, que a variável a ser modelada é a variável resposta, e as demais são suas variáveis preditoras.

Tomemos duas variáveis bastante representativas: o número de classes adicionadas como preditora e o número de métodos adicionados em um *commit* como variável resposta. A razão para aplicação de regressão linear na concepção dos modelos é que não se pode omitir o fato das variáveis apresentarem forte dependência linear entre si. Se o ignorássemos e tratássemos cada variável como independente seria possível conceber simulação com operações incoerentes como, por exemplo, adição de 30 classes e 0 métodos, uma evolução claramente irreal. Desta forma, apesar do esforço inicial para confirmar a hipótese *power-law* e a expectativa de que estes resultados fossem suficientes para fins de simulação, a evidência maior é dada aos modelos de regressão linear, através dos quais, por exemplo, a adição de certo número de classes deriva a adição de um número proporcional de outras entidades e relacionamentos.

A construção do modelo de regressão linear é incremental. Para fins de exemplificação, considere que a adição de classes obedece a uma *power-law* com parâmetros $\alpha = 2.2$ e *x-min* = 1. Com este conhecimento é possível simular sua ocorrência, simplesmente gerando valores aleatórios que obedeçam esta distribuição. Isto permite que modelos lineares para outras variáveis sejam determinados como função desta. Passo após passo, para todas as variáveis. A cada modelo de variável concebido, essa se torna apta a ser utilizada como preditora, de forma isolada ou combinada.

Um pouco mais apropriadamente, temos:

1) É escolhida uma variável cujo comportamento seja conhecido (por exemplo, através de um modelo de distribuição) para ser a primeira variável preditora;

2) O segundo passo consiste em procurar variáveis resposta que apresentem alta correlação linear (valor de R^2) com a primeira variável preditora (adições de classes e de métodos, por exemplo, normalmente apresentam alta correlação). Para tal fim, utiliza-se também a ferramenta estatística R. Através do comando *corr* calcula-se a correlação de Pearson entre duas variáveis, uma métrica amplamente utilizada, cujo valor varia entre -1 e 1 e indica se duas variáveis são linearmente dependentes (1), inversamente dependentes (-1), ou se não há relação entre ambas (0). Afirma-se que valores superiores a 0.7 e inferiores a -0.7 configuram relações de forte dependência linear;

3) Uma vez modeladas duas ou mais entidades, repete-se o processo de predição combinando-as, excluindo da predição variáveis que apresentem dependência linear entre si (problema conhecido como multicolinearidade) visando maximizar o valor de R^2.

Uma observação adicional se refere aos modelos com dependência cíclica. Ou seja, se modelamos uma variável *v1* como função de *v2*, e *v2* como função de *v1* não poderemos executar a simulação das variáveis de forma adequada.

Assim, o método acima descrito culmina com o conjunto de modelos para as 15 variáveis de um software.

2.3. As Leis de Lehman

Em mais de uma década de pesquisas, Lehman e colaboradores observaram uma série de propriedades que se preservaram mediante diversas observações. A estas se deu o nome de Leis de Lehman. Resumiremos abaixo uma breve conceituação sobre estes princípios. Apesar de algumas leis serem trivialmente verificáveis (*e.g.*: Crescimento Contínuo), outras possuem definição pouco intuitiva (*e.g.*: Sistema de Feedback). Trabalhos anteriores estabelecem métricas ou evidências extraídas do código estático que servem de base para avaliação das Leis de Lehman [XCN2009, Leh1998]. A seguir descrevemos as Leis de Lehman e as métricas que utilizaremos.

I) **Mudança Contínua** – O software muda continuamente para adaptar-se ao ambiente, ou tenderá a se tornar menos útil ao longo do tempo.

   Métricas: Quaisquer evidências numéricas ou gráficas de que mudanças estão sendo aplicadas ao código, tais quais gráficos que apresentem as mudanças no tempo, o número acumulativo de mudanças, ou evidências de crescimento contínuo (que também confirma a Lei 6).

II) **Complexidade Crescente** – O software tenderá a tornar-se mais complexo de manter à medida que evolui.

   Métricas: Aumento da complexidade ciclomática, aumento do número de relacionamentos como *method calls*, crescimento de outras métricas de acoplamento.

III) **Auto-regulação** – Os processos de evolução de softwares são auto-regulatórios. Regulando, inclusive, seu tamanho no tempo.

Métricas: Oscilações positivas e negativas no gráfico de crescimento do software. Ou seja, ora o software cresce, ora decresce.

IV) **Conservação da Estabilidade Organizacional** – A carga de trabalho necessária para manter o software permanece constante com o tempo.

Métrica: Lehman sugere a quantidade de mudanças por *release* como métrica. No entanto, Xie *et al.* [XNC2009] revela certa inadequação desta métrica para mensurar carga de trabalho. Em sua pesquisa com softwares *open source*, utilizou-se esta métrica e a quarta lei não foi confirmada. Como o intuito deste trabalho não é questionar as leis de Lehman, mas apenas explorá-las como referencial, utilizou-se a métrica original sugerida por Lehman. No entanto, como não exercitamos o conceito clássico de releases, o relaxamos considerando que conjuntos de 10 versões de *commit* configuram uma pseudo-*release*. A justificativa para estabelecermos pontos de observações a cada 10 *commits* dar-se-á no Capítulo 5, quando discutimos o *design* e resultados do experimento. Ainda com relação à métrica, utilizaremos a soma de todas as operações sugeridas entre as versões de pseudo-release para mensurar a quantidade de mudanças aplicadas.

V) **Conservação da Familiaridade** – Toda a equipe relacionada ao desenvolvimento de um software precisa compreendê-lo, implicando em taxa de crescimento constante ou declinante. Um corolário adicional sugere que mudanças grandes são seguidas de mudanças pequenas.

Métrica: Evidência visual através da curva de crescimento de módulos estruturais [XNC2009]. Para fins desta pesquisa, foi considerado apenas o crescimento do número de tipos como métrica.

VI) **Crescimento Contínuo** – O software tenderá a crescer continuamente.

Métricas: Quaisquer evidências no aumento de entidades do código, como tipos ou métodos.

VII) **Qualidade Decrescente** – A qualidade do software tenderá a degradar com o tempo, a não ser que medidas para adaptá-lo sejam tomadas.

Métrica: A abordagem tradicional sugestiona considerar o número de defeitos por release do software, no entanto esta abordagem é inviável para versões simuladas. Uma métrica que pode ser extraída a partir do código estático encontra-se no trabalho de Palson *et al.*[PSE2004]: a quantidade percentual de funções alteradas por *release*, que deve aumentar.

VIII) **Sistema de Feedback** – Trata-se da lei mais complexa de ser compreendida. Lehman [Leh1998] descreveu a evolução de software como um processo multi-*loop*, multi-*level* e um sistema de *feedback* multi-agente. Por ser de *feedback*, afeta o ambiente, e deste receber estímulos que o alterarão. Também é multi-agente, porque múltiplos agentes atuam neste processo, até mesmo externos ao desenvolvimento, mas ligados ao ambiente organizacional no qual o software está inserido. E é multi-*level* porque as mudanças ocorrem em diversos níveis lógicos ou físicos. Multi-*loop* porque *loops* de interação distintos ocorrem em partições lógicas ou físicas diferentes do software.

Métrica: Uma demonstração do sistema de *feedback* dá-se pela possibilidade de aproximar o número de módulos num *release* a partir do número do *release*. Ou seja, assume-se que a compreensão de parte do sistema de *feedback* permitiu equacionar este aspecto da evolução. O modelo exponencial proposto por Lehman [Leh1998] que relaciona estas variáveis é dado abaixo:

$$S = a \times \sqrt[3]{\#R} + b$$

Onde: S é o novo número de módulos do software;
#R é o número do *release* atual;
a e *b* são os parâmetros a serem estimados.

Esta mesma métrica é utilizada neste estudo para verificar a lei. No nosso caso, a regressão linear visa modelar o número de tipos em detrimento do número da versão. Há uma diferença considerável entre as duas abordagens. No entanto, uma vez que tipos são nossas unidades mais representativas de um módulo, qualquer outra estratégia parece-nos mais inadequada.

Terminamos, assim, a explanação acerca das 8 Leis de Lehman e como pretendemos abordá-las. Segue uma breve discussão sobre as propriedades das redes livres de escala de software e o que pretendemos extrair da simulação neste sentido.

2.4. Redes Livres de Escala de Software.

Uma rede livre de escala (RLE) é uma rede onde nós estão interligados por arestas que obedecem a uma lei de potência (*power-law*). Ou seja, há um comportamento de poucos nós fortemente conectados, e muitos nós poucos conectados. Adicionalmente, novos nós conectam-se preferencialmente a nós com muitas conexões.

Muitas redes do mundo real mostram-se redes livres de escala, tal qual citações em artigos científicos, redes sociais, conexões entre aeroportos, e o relacionamento entre entidades de software. Em seu trabalho, Souza [Sou2010] demonstrou ser possível sintetizar redes livres de escala similares às redes de software, de tal forma que esta se torna perfeitamente distinguível de outras RLEs. Em seu trabalho, o autor considerou apenas as classes Java como nós, e as arestas representam qualquer relacionamento entre classes (como *extends* e *implements*) ou entre métodos (como *method calls*).

Agrupa-se os nós em conjuntos de tamanho 3, denominado tríades. Considerando uma tríade qualquer, com nós representando classes, e arestas dirigidas representando seus relacionamentos, temos o conjunto de treze possíveis combinações, conforme apresentado na Figura 4.

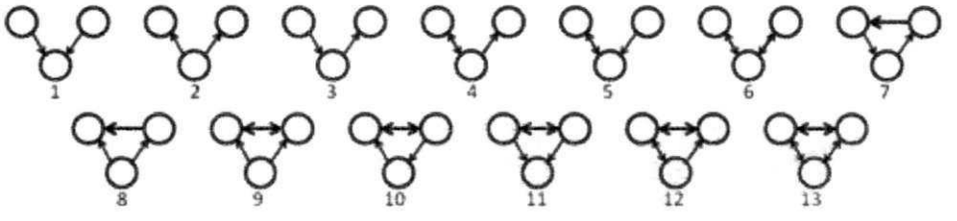
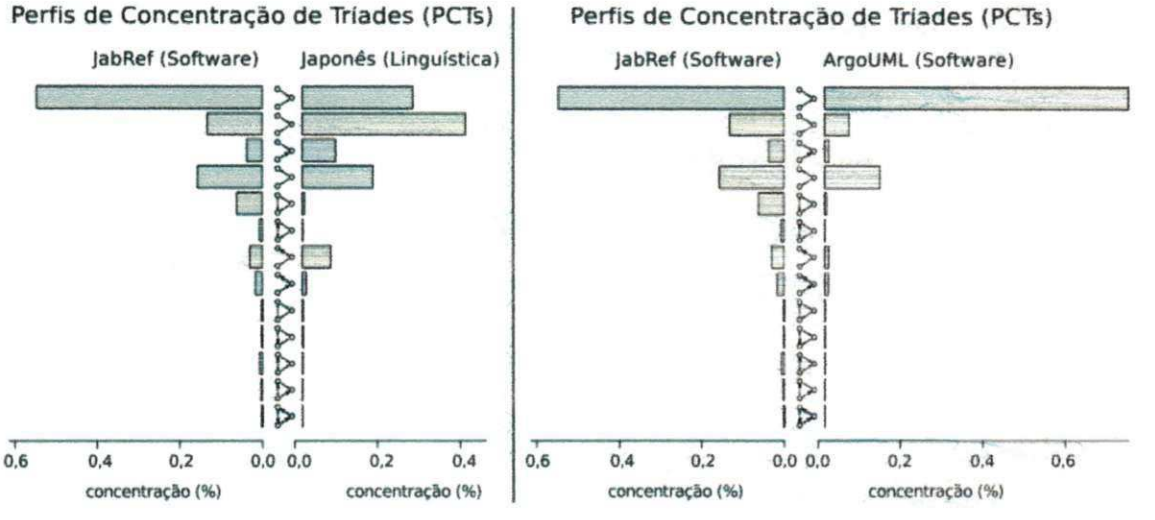


Figura 4 – O conjunto de tríades derivado da combinação de três nós e suas arestas.

A seguir, o software inteiro é analisado e a ocorrência de cada uma destas treze tríades é contabilizada. Obviamente, este processo refere-se apenas à abordagem para mensurar tríades de software, uma vez que outras RLEs já possuem seu conjunto de tríades conhecido. Este conjunto de frequência de tríades caracterizam unicamente diversas redes livre de escala, inclusive a de software, conforme apresentado na Figura 5. Em (a), se tomarmos as duas primeiras tríades de cima para baixo perceberemos que já há uma diferenciação entre os dois tipos de rede. No software, o primeiro tipo de tríade é mais frequente que o segundo, enquanto que na linguística é o inverso. Já em

(b), percebe-se clara similaridade entre ambas as redes provenientes de softwares diferentes. Apesar de haver diferença nos valores de cada tríade, há muito mais respeito à ordem de frequência.

Neste trabalho, utilizaremos o ferramental fornecido por Souza [Sou2010] para verificar se o processo simulatório aqui proposto gera sucessivas versões que preservam a propriedade de uma rede livre de típica de software segundo o critério de concentração de tríades. Conforme apresentaremos no capítulo a seguir, nossa modelagem considera as menores entidades de software (atributos e métodos), no entanto, pelo fato de utilizarmos o ferramental proposto por Souza, apenas as classes são consideradas na avaliação frente às propriedades das RLEs.



(a) (b)

Figura 5 – Perfil de distribuição de tríades. (a) Uma comparação entre duas redes distintas que evidência as diferenças de concentração de algumas tríades. (b) Comparação entre duas redes de software onde se observam as similaridades entre as concentrações[3].

[3] Figura original extraída de Souza [Sou2010], pág. 10.

Capítulo 3

Simulações Numérica e Aplicada

As simulações visam aplicar os modelos anteriormente citados de duas formas distintas: a Numérica e a Aplicada. A simulação numérica realiza a geração de mudanças de forma simplista, propiciando uma série de investigações diretas sobre o modelo. No entanto, apenas introduzindo as mudanças à estrutura podemos investigar quão suficiente o modelo é para fins práticos de sua aplicação no contexto das pesquisas em Engenharia de Software, bem como para uma investigação adequada das propriedades de redes livres de escala.

No centro desta atividade está uma ferramenta desenvolvida pelo autor, denominada *Statistical Evolution Simulator*[4] (SES). A Figura 6 descreve seu papel deste processo.

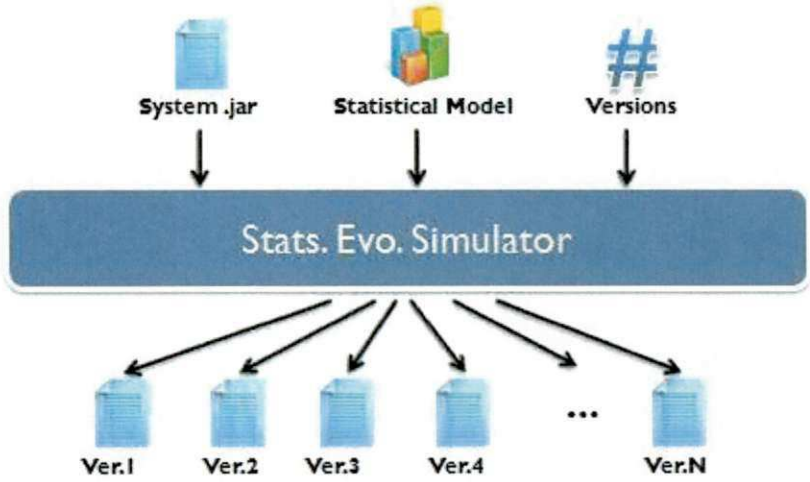


Figura 6 – Uma visão geral do processo de simulação proposto.

[4] A implementação completa do simulador em Java encontra-se disponível no site desta pesquisa [Dam2011], na seção **Applied Simulation.**

A entrada do simulador consiste em um arquivo *.jar* (código Java compilado), um outro arquivo que descreve um modelo de evolução para o software; além do número de versões a ser gerado. Sua execução se dá em 4 passos: (1) cria uma visão lógica do design a partir do *.jar*. Para tal, utiliza-se um extrator do *bytecode* que dá acesso fácil à estrutura de entidades e seus relacionamentos, o Design Wizard [DW2010]; (2) aplica a Simulação Numérica por N vezes, onde N é o número de versões a ser gerado; (3) aplica a Simulação Aplicada, caso seja solicitado, em cada uma das ordens de mudança provenientes da Simulação Numérica; (4) Geração do arquivo de saída. A saída pode ser de duas formas: se foi executada uma simulação numérica, a saída é um arquivo constituído por descritores (tuplas) de mudanças, onde cada tupla contém as mudanças que gerarão uma nova versão do software a partir da versão anterior; do contrário, se a execução foi a simulação aplicada, a saída consiste de um arquivo textual contendo N versões modificadas em formato *.txt* compatível com o trabalho de Souza [Sou2010], que essencialmente armazena pares contendo o nome da primeira classe, o nome da segunda classe. A saída não representa o design completo (com atributos, métodos e seus relacionamentos) por mera questão de escassez de tempo para concretizar a implementação.

3.1. A Simulação Numérica

A simulação numérica é o nome dado ao processo interno ao SES que interpreta o modelo e gera valores para cada uma das variáveis a partir desse. Internamente o simulador numérico possui uma API[5] para geração de números aleatórios baseado em uma distribuição desejada e seus parâmetros. Além disto, ele é capaz de interpretar modelos lineares simples, com dois termos e um operador matemático. Por fim, seu trabalho se resume a unir os 15 valores em uma única tupla e persistir estes dados em arquivo ao final do processo. A simplicidade da Simulação Numérica é contrastada pela Simulação Aplicada.

3.2. A Simulação Aplicada

A simulação aplicada recebe como entrada os resultados da simulação numérica, e tenta introduzir cada tupla de mudanças à estrutura pré-existente. Apesar de conceitualmente simples, na prática, ambas a adição e a remoção demandam o uso de

[5] http://acs.lbl.gov/software/colt/ - The Colt Project

heurísticas que introduzem limitações ao modelo, especialmente no que se refere às remoções[6].

Algumas informações se fazem necessárias para uma compreensão do algoritmo.

I) O algoritmo recebe a estrutura do software em forma de um grafo.

II) Uma `tupla` é um mapa entre uma descrição de mudança e o seu valor.

Ex.: `tupla.ADD_TIPO` poderia retornar 5. Descrevendo uma adição de 5 novos tipos.

III) Num `grafo`, as entidades estão organizadas hierarquicamente e de forma direta em termos de seus pacotes, classes, métodos e atributos.

a. `grafo.packages` – Uma lista de dados `package`, que possui os pacotes Java que o software possui.
b. `package.tipos` – Uma lista de dados `tipo`, que possui as classes e interfaces Java que o pacote possui.
c. `tipo.metodos` – Uma lista de dados `metodo` que possui os métodos pertencentes ao tipo.
d. `metodo.tipo` – O tipo ao qual o método pertence.
e. `atributo.tipo` – O tipo ao qual o atributo pertence.
f. `tipo.pacote` – O pacote ao qual o tipo pertence.
g. `tipos.atribs` – Uma lista de dados `atributo` que possui os atributos pertencentes a um tipo.
h. `grafo.tipos` – Uma lista de todos os nós `tipo` do grafo.
i. `grafo.metodos` – Uma lista de todos os nós `metodo` do grafo.
j. `grafo.atributos` – Uma lista de todos os nós `atributo` do grafo.

IV) Cada entidade possui seu conjunto de suas relações.

a. `tipo.implements` – Uma lista de dados do tipo `aresta` que especificam os tipos implementados por `tipo`.
b. `tipo.extends` – Uma lista de dados do tipo `aresta` que especifica o tipo estendido por `tipo`.
c. `metodo.throws` – Uma lista de dados do tipo `aresta` que especificam os tipos lançados por `metodo`.

[6] A implementação completa do algoritmo em Java encontra-se disponível no site desta pesquisa [Dam2011], na seção **Applied Simulation.**

V) Em um `grafo` é também possível acessar diretamente um conjunto de `arestas` pelo seu tipo.

Ex.: `grafo.implements`: retorna um conjunto com todas as arestas que representam relações do tipo *implements* entre dois tipos.

Uma visão geral do algoritmo de simulação é apresentada na Figura 7 em pseudocódigo. A linha 2 se refere à função para as operações que envolvem adições, que são relativamente triviais; a linha 3 trata das operações de acomodações de mudanças relacionadas às remoções; por fim, na linha 4, a função para as *method calls*, que podem ser adições ou remoções e, por isto, tratadas de forma diferenciada.

```
1  simulacao_aplicada(grafo, tupla)
2      aplica_adicoes(grafo, tupla)
3      acomoda_remocoes(grafo, tupla)
4      aplica_methcalls(grafo, tupla)
```

Figura 7 – O pseudo-código da simulação aplicada.

3.3. A Aplicação das Adições

Quanto às adições, o pseudocódigo está apresentado na Figura 8. Novos nós são adicionados diretamente ao grafo de forma direta (linhas 5 a 20). A adição de arestas, no entanto, visa preservar a lei da rede livre de escala da seguinte forma: se a estrutura inteira de um software possui *m* arestas no total, um nó com *n* arestas possui chance de *n/m* de receber uma nova aresta. Esta operação é executada pela função `roleta(entidades, aresta)`, que promove um sorteio utilizando uma roleta probabilística para definir um nó a receber a nova aresta. Nas linhas 21 a 27 evidencia-se a adição de aresta. Ao adicionarmos um novo nó, faz-se necessário explicitar qual outro nó o conterá (e.g.: um tipo demanda uma aresta do tipo *contains*, pois um pacote deverá conter aquele tipo). Para tal, a roleta é utilizada entre os tipos que podem conter a nova entidade (linhas 22 e 26); criam-se as novas arestas (linhas 23 e 27), e estas são adicionadas ao grafo (linhas 24 e 28). Outro detalhe apresenta-se nas linhas de 29 a 32, onde se adiciona uma relação obrigatória do tipo *return* se o novo nó é um método.

A adição de relações (linhas 34 a 52) evidencia uma característica inerente ao nosso modelo. Neste, os novos relacionamentos **não são alocados** aos novos nós sugeridos em uma mesma tupla. Ou seja, se a tupla sugere a adição de 5 novos tipos e 2 novas relações do tipo *extends*, não se delega estas arestas (necessariamente) aos novos

nós. Solicitam-se os tipos envolvidos no relacionamento através do uso da roleta que preserva a propriedade de redes livre de escala de software, e entre estes tipos aplica-se a aresta. Se optássemos por adicionar os novos relacionamentos às novas entidades estaríamos assumindo um comportamento que nosso modelo não cobre, nem foi estudado até o momento, possivelmente introduzindo erros fora de controle à simulação.

3.4. A Concretização das Remoções

Quando se trata de remoções, emergem diversos problemas de uma mesma natureza: a **não-aceitação estrutural da mudança**. Em um pequeno exemplo a problemática se torna clara. Consideremos um tupla que recomenda a remoção de 2 tipos e 5 métodos[7]. Se o menor tipo do software (em número de métodos) tiver tamanho 6, a mudança já se torna inaplicável. A situação se agrava quando a tupla sugere remoções mais complexas como a seguinte, retirada de uma simulação do Spring Framework (destaque para os valores diferentes de zero):

ADIC. PAC.	REM. PAC.	ADIC. TIPO	REM. TIPO	ADIC. MÉT.	REM. MÉT.	ADIC. ATRIB.	REMO. ATRIB.	ADIC. *IMPL.*	REM. *IMPL.*	ADIC. *EXTE.*	REM. *EXTE.*	ADIC. *THRO.*	REMO. *THRO.*	*METH. CALLS*
0	1	0	5	0	17	0	8	0	0	0	1	0	0	-61

Decorre um problema: mesmo que seja possível incorporar esta mudança, encontrar a combinação perfeita de pacotes, tipos, métodos, etc. para tal é uma tarefa de tempo exponencial, tomando em conta o número de unidades estruturais que o software possui. Para fins de conhecimento, um algoritmo de *backtracking* com diversas podas foi criado para este fim, mas continuamente mudanças pequenas se mostravam inaplicáveis, mesmo após execuções que duravam vários minutos. Com efeito, não obtivemos nenhum sucesso com tuplas que sugeriam remoções superiores a dois tipos. Assim, seja por limitação do algoritmo de busca ou por inadequação da sugestão de mudança em si, a questão é que introduzir a mudança na estrutura sem alterá-la mostrou-se inviável. Partiu-se então para uma abordagem inovadora, denominada **Algoritmo de Acomodação de Remoções**.

[7] Este problema decorre da utilização dos modelos estatísticos puramente baseados no histórico sem conhecimento prévio da estrutura. Soluções para esta lacuna serão discutidas no Capítulo 6.

```
1  aplica_adicoes(grafo, tupla)
2      adiciona_entidades(grafo, tupla)
3      adiciona_relacoes(grafo, tupla)
4
5  adiciona_entidades(grafo, tupla)
6      if (tupla.ADD_PACOTE > 0)
7          for (i = 1 to tupla.ADD_PACOTE)
8              adic_entidade(grafo, new pacote())
9      if tupla.ADD_TIPO > 0
10         for (i = 1 to tupla.ADD_TIPO)
11             adic_entidade(grafo, new_tipo())
12     if tupla.ADD_METODO > 0
13         for (i = 1 to tupla.ADD_METODO)
14             adic_entidade(grafo, new_metodo())
15     if tupla.ADD_ATRIBUTO > 0
16         for (i = 1 to tupla.ADD_ATRIBUTO)
17             adic_entidade(grafo, new_atributo())
18
19 adic_entidade(grafo, entidade)
20     grafo.add_no(entidade)
21     if (entidade instanceof type)
22         pacote_selec = roleta(grafo.pacotes)
23         aresta_contains = new aresta(CONTAINS, pacote_selec, entidade)
24         grafo.add_aresta(aresta_contains)
25     else if (entidade instanceof metodo or entidade instanceof atributo)
26         tipo_selec = roleta(grafo.tipos)
27         aresta_contains = new aresta(CONTAINS, tipo_selec, entidade)
28         grafo.add_aresta(aresta_contains)
29         if (entidade instanceof metodo)
30             tipo_selec = roleta(grafo.tipos)
31             aresta_contains = new aresta(RETURN, tipo_selec, entidade)
32             grafo.add_aresta(aresta_contains)
33
34 adiciona_relacoes(grafo, tupla)
35     if tupla.ADD_IMPLEMENTS > 0
36         for (i = 1 to tupla.ADD_IMPLEMENTS)
37             adic_relacao(grafo, IMPLEMENTS)
38     if tupla.ADD_EXTENDS > 0
39         for (i = 1 to tupla.ADD_EXTENDS)
40             adic_relacao(grafo, EXTENDS)
41     if tupla.ADD_THROWS > 0
42         for (i = 1 to tupla.ADD_THROWS)
43             adic_relacao(grafo, THROWS)
44
45 adiciona_relacao(grafo, titulo)
46     if (titulo == IMPLEMENTS or titulo == EXTENDS)
47         tipo1 = roleta(grafo.tipos)
48     else if (titulo == THROWS)
49         tipo1 = roleta(grafos.metodos)
50     tipo2 = roleta(grafo.tipos, aresta)
51     aresta = new aresta(titulo, tipo1, tipo2)
52     grafo.add_aresta(aresta)
```

Figura 8 – O pseudocódigo das operações de adição de entidades e relações.

O algoritmo objetiva **esvaziar** entidades candidatas, tornando sua remoção possível. Suponha uma classe Java qualquer; esvaziar a classe sugere remover ou realocar seus métodos, atributos e relacionamentos. Consideremos o exemplo de mudança simulada a seguir, onde os valores diferentes de zero estão evidenciados:

ADIC. PAC.	REM. PAC.	ADIC. TIPO	REM. TIPO	ADIC. MÉT.	REM. MÉT.	ADIC. ATRIB.	REMO. ATRIB.	ADIC. *IMPL.*	REM. *IMPL.*	ADIC. *EXTE.*	REM. *EXTE.*	ADIC. *THRO.*	REMO. *THRO.*	*METH. CALLS*
0	0	0	2	0	17	0	8	0	1	0	1	0	4	-48

Deste exemplo, evidencia-se que o maior problema é remover os dois tipos. Ou seja, faz-se necessário encontrar, dentre todos os tipos existentes, um par candidato a remoção. Tal par deve se aproximar ao máximo das restrições impostas pelas demais remoções. Como mencionamos anteriormente, se fosse possível encontrar sempre uma combinação perfeita o trabalho estaria resolvido, mas esta premissa além de improvável demanda alto custo computacional. Isto sugere que mesmo tentando esvaziar as entidades candidatas baseado nas demais remoções, é muito provável que sobrem sub-entidades e relacionamentos.

A solução encontrada para este problema é **mover** estas unidades estruturais para outras entidades do software, a fim de esvaziar as entidades candidatas e removê-las. Dados a tupla de mudanças anterior e uma estrutura qualquer, se os dois melhores tipos candidatos possuem juntos 19 métodos, sobrarão pelo menos 2 métodos que precisarão ser movidos para que os tipos possam ser removidos vazios.

É possível inferir que para as demais unidades estruturais, tais quais tipos e métodos, o problema pode ser resolvido de forma análoga. É esta a heurística do algoritmo, uma solução *bottom-up* onde a remoção de pacotes depende da remoção de seus tipos, a remoção dos tipos depende da remoção de seus métodos, atributos e relacionamentos específicos.

Apresentamos, na Figura 9, uma visão geral do Algoritmo de Acomodação de Remoções. Inicialmente (linhas de 1 a 6), o algoritmo inicia as listas de entidades e relacionamento vazia. Na linha 9, é chamada a função que construirá o conjunto de candidatos a remoção. Na linha 10, aplica-se as remoções possíveis de serem realizadas dentro dos conjuntos de candidatas, visando uma remoção natural das entidades, sem a necessidade de alocar subestruturas para outras entidades. A linha 11 se refere ao esvaziamento artificial das entidades candidatas; seu conteúdo restante é alocado para outras entidades não-candidatas, a fim de deixá-la vazia, pronta para remoção.

Esta solução mostra-se adequada para realizar concretizar a mudança sugerida, mas introduz mudanças adicionais, além do escopo da tupla original.

```
1   tipos_cand_remocao = {}
2   metodos_cand_remocao = {}
3   atribs_cand_remocao = {}
4   ext_cand_remocao = {}
5   impl_cand_remocao = {}
6   throws_cand_remocao = {}
7
8   acomoda_remocoes(grafo, tupla)
9       prepara(grafo, tupla)
10      remove(tupla)
11      esvazia(grafo)
```

Figura 9 – Pseudocódigo com a visão geral do algoritmo de acomodação de remoções.

O pseudocódigo da função `prepara(grafo, tupla)` é apresentado a seguir na Figura 10. Na linha 14, verifica-se o número de pacotes a serem removidos. Caso seja necessário, na linha 15 ordenam-se os pacotes em ordem crescente pelo seu número de tipos. Na linha 16, observa-se uma chamada para a função que vai alocar todas as subentidades de cada pacote que deve ser removido como candidatos à remoção (comandos nas linhas 21 a 25). Na linha 17, verifica-se se o número de tipos candidatos já é suficiente face o número total de tipos que devem ser removidos, caso não seja, abordagem similar à remoção de pacotes é adotada.

Conforme veremos a seguir, as demais estruturas demandam abordagem mais direta, pois não possuem subestruturas complexas tais quais pacotes e tipos. Nestes casos, retomando a Figura 9 como referência, a resolução se dá através da função `remove(tupla)`, cujo pseudocódigo é apresentado na Figura 11. As linhas de 47 a 55 demonstram a remoção dos relacionamentos baseado na lista de candidatos, e quando esta se encontra vazia, de forma direta no grafo. O mesmo se aplica à remoção de entidades do tipo atributo[8]. A remoção da aresta no grafo é dada de forma aleatória. Ao contrário da adição que possui uma heurística para preservar a lei de rede complexa, não há, neste caso, nenhuma referência para degradação da estrutura de uma rede complexa. Portanto, optou-se pela estratégia mais simples.

Ainda na Figura 11, nas linhas de 56 a 63 e 65 a 73, encontram-se dois códigos extremamente similares entre si. Referem-se à remoção de entidades complexas se

[8] As linhas que sugerem estas operações foram omitidas para simplificar a quantidade de informação apresentada.

estiverem vazias. E, caso não haja entidades candidatas, uma nova, retirada aleatoriamente do grafo, é alocada como candidata para sofrer o processo seguinte de esvaziamento. Com relação aos pacotes, não existe tal preocupação (linhas 75 a 77), pois, se há na tupla ordem de remoção de certo número de pacotes, há obrigatoriamente o mesmo número de pacote na lista de candidatos, ao contrário das demais entidades e relacionamentos, cujo número de candidatos pode ser definido pelos pacotes escolhidos.

```
13  prepara(grafo, tupla)
14      if (tupla.REM_PACOTES > 0)
15          pacotes_ordenados = sort(grafo.pacotes, #_DE_TIPOS)
16          prepara_pacotes(pacotes_ordenados, tupla.REM_PACOTES)
17      if (tupla.REM_TIPOS > tipos_cand_remocao.size())
18          tipos_ordenados = sort(grafo.tipos, #_DE_METODOS)
19          prepara_tipos(tipos_ordenados, tupla.REM_TIPOS)
20
21  prepara_pacotes(pacotes, num_pacs_rem)
22      for (i = 1 to num_pacs_rem)
23          tipos_cand_remocao.addAll(pacotes.get(i).tipos)
24      for (i = 1 to tipos_cand_remocao.size())
25          tipo = tipos_cand_remocao.get(i)
26          prepara_tipo(tipo)
27
28  prepara_tipos(tipos, num_tipos_rem)
29      for (i = 1 to (num_tipos_rem - tipos_cand_remocao.size()))
30          tipo = tipos.get(i)
31          prepara_tipo(tipo)
32
33  prepara_tipo(tipo)
34      atribs_cand_remocao.addAll(tipo.atributos)
35      ext_cand_remocao.addAll(tipo.extends)
36      impl_cand_remocao.addAll(tipo.implements)
37      metodos_cand_remocao.addAll(tipo.metodos)
38      for (j = 1 to tipos.metodos.size())
39          metodo = tipos.metodos.get(j)
40          throws_cand_remocao = metodo.throws
```

Figura 10 – A função prepara e suas funções auxiliares que estabelecem os conjuntos de entidades e relacionamento candidatos à remoção baseado na tupla original de remoção.

O fim do processo de remoção ocorre com a função `esvazia(grafo)`, que trata de realocar subestruturas que impedem a concretização de uma remoção. Sua idéia é simples: para cada entidade candidata C não vazia, move-se sua subestrutura para outras entidades do grafo não-candidata e remove-se C vazia. No intuito de preservar as propriedades de redes livres de escala, a alocação acontece como uma remoção seguida de uma adição. A adição se dá de forma idêntica à adição de novos nós e arestas

conforme já apresentado. O pseudocódigo apresentado na Figura 12 descreve esta estratégia.

```
45  remove(tupla)
46      while (tupla.REM_IMPLEMENTS > 0)
47          if (impl_cand_remocao.size() > 0)
48              impl_cand_remocao.remova_primeiro()
49          else grafo.implements.remova_primeiro()
50          tupla.REM_IMPLEMENTS -= 1
51
52      ... # Código similar para as demais relações
53      ... # e a entidade atributo.
54
55
56      for (i = 1 to tupla.REM_METODOS)
57          if (metodos_cand_remocao.size() == 0)
58              metodo = grafo.metodos.remova_primeiro()
59              metodos_cand_remocao.add(metodo)
60              throws_cand_remocao = metodo.throws
61              mcalls_cand_remocao = metodo.methcalls
62          if (metodos_cand_remocao.proximo().esta_vazio())
63              metodos_cand_remocao.remova_primeiro()
64
65      for (i = 1 to tupla.REM_TIPOS)
66          if (tipos_cand_remocao.size() == 0)
67              tipo = grafo.tipos.remova_primeiro()
68              tipos_cand_remocao.add(tipo)
69              metodos_cand_remocao.add(tipo.metodos)
70              impl_cand_remocao = tipo.implements
71              ext_cand_remocao = tipo.extends
72          if (tipos_cand_remocao.proximo().esta_vazio())
73              tipos_cand_remocao.remova_primeiro()
74
75      for (i = 1 to tupla.REM_PACOTES)
76          if (pacotes.proximo().esta_vazio)
77              pacotes.remova_primeiro()
```

Figura 11 – O algoritmo de remoção baseado na tupla. Neste primeiro momento objetiva-se remover as unidades estruturais de forma direta, minimizando a necessidade de esvaziamento por alocação de subentidades.

Se retornarmos à Figura 7, veremos que o último passo de todo o algoritmo se resume à adição ou remoção do número líquido de relacionamentos do tipo *method calls*. Nota-se que passos anteriores implicam na remoção de certo número de *method calls*. Por exemplo, digamos que foram removidos 10 métodos e esta remoção implicou na remoção de 30 *method calls*. Se este número é superior ao número sugerido pelo simulador para remoção, simplesmente adiciona-se a diferença de volta ao grafo com a roleta probabilística. Se é inferior, remove-se o excesso de arestas *method call* de forma aleatória, com igual possibilidade para todas as entidades.

```
79  esvazia(grafo)
80      for (i = 1 to tipos_cand_remocao.size())
81          tipo = tipos_cand_remova.proximo()
82          metodos = tipos.metodos
83          if (metodos.size() > 0)
84              for (j = 1 to metodos.size())
85                  metodo = metodos.proximo();
86                  tipo_selec = roleta(grafo.tipos)
87                  aresta_contains = new aresta(CONTAINS, tipo_selec, metodo)
88                  grafo.add_aresta(aresta_contains)
89          grafo.tipos.remova(tipo)
90          if (tipo.pacote.esta_vazio())
91              grafo.pacotes.remova(tipo.pacote)
```

Figura 12 – Pseudocódigo da operação de esvaziamento de entidades.

3.5. Análise de Complexidade da Concretização de Mudanças

Anteriormente apresentamos o algoritmo de concretização de mudanças, com especial atenção nas concretizações das remoções. Este último visa confrontar a solução força-bruta, que obrigatoriamente analisa todas as combinações entre entidades e relacionamentos, a fim de encontrar aqueles que satisfazem as ordens de remoção.

Consideremos um software $S(E,R)$. O seu conjunto de entidades E é dado como $E(p,c,m,a)$ e seus relacionamentos como $R(e,i,t,mc)$, onde: p = número de pacotes do software; c = número de classes do software; m = número de métodos do software; a = número de atributos do software. E, no que refere aos relacionamentos: e = número de *extends* do software. i = número de *implements* do software. t = número de *throws* do software. mc = número de *method calls* do software.

Considere uma tupla genérica **apenas de remoções** T(*rp*, *rc*, *rm*, *ra*, *re*, *ri*, *rt*, *rmc*), onde cada elemento representa o número de remoções (*r*) de cada uma das variáveis anteriormente descritas do software $S(E,R)$. *Method calls* só aparecem na tupla se o número líquido for negativo, conforme limitações do modelo explicitadas anteriormente.

Um algoritmo força-bruta para solução do problema é hierárquico no sentido que para remover *rp*, e necessário combinar os pacotes existentes condicionados às restrições impostas por *rc*. Para remover *rc* há outras diversas restrições, e assim por diante. Mais apropriadamente temos:

(1) Remoção de *rp* condicionada à remoção de *rc*;

(2) Remoção de *rc* condicionada à remoção de *rm*, *ra*, *re* e *ri*.

(3) Remoção de *rm* condicionada à remoção de *rt* e *rmc*, quando se aplicar.

Tomando por base (1), o número de combinações é dado por $C_c^p = \frac{n! \times (n-s)!}{s!}$.

O mesmo se aplica para (2) e (3). Deste, se evidencia a natureza fatorial do algoritmo. Em notação *Big-Oh* podemos assumir, desprezando constantes, que o algoritmo força bruta apresenta crescimento fatorial.

Para a análise do algoritmo de acomodação de mudanças, retomaremos o código anteriormente apresentado na Figura 9, a seguir:

```
1   tipos_cand_remocao = {}
2   metodos_cand_remocao = {}
3   atribs_cand_remocao = {}
4   ext_cand_remocao = {}
5   impl_cand_remocao = {}
6   throws_cand_remocao = {}
7
8   acomoda_remocoes(grafo, tupla)
9       prepara(grafo, tupla)
10      remove(tupla)
11      esvazia(grafo)
```

As inicializações (linhas 1 a 6) apresentam tempo $O(1)$. Quanto à função `acomoda_remocoes(grafo, tupla)`, esta se subdivide em 3 funções relativamente simples de analisar: `prepara(grafo, tupla)`; `remove(tupla)` e `esvazia(grafo)`

Na Figura 10 observa-se que a função `prepara(grafo, tupla)` possui condições iniciais (linhas 14 a 19) onde há uma ordenação $O(nlogn)$ e chamadas às `prepara_pacotes(pacotes, num_pacs_rem)` e `prepara_tipos(tipos, num_tipos_rem)`. Realizando uma análise *bottom-up* temos que a função `prepara_tipo(tipo)` (linhas 33 a 40) é $O(e + i + a + 2 \times m)$. A função `prepara_tipos(tipos, num_tipos_rem)` faz uso dessa e possui tempo $O((rt - t) \times (e + i + a + 2 \times m))$. Por fim, a função `prepara_pacotes(pacotes, num_pacs_rem)` apresenta tempo $O((rp \times t) + t^2)$. O termo t^2 se refere à multiplicação de t pelo tempo de execução da função `prepara_tipos(tipos, num_tipos_rem)` (linhas 24 e 26). De fato, o maior expoente que se apresenta neste algoritmo é 2, e deste podemos concluir sem demasiada preocupação que o tempo do da função `prepara` é polinomial em $O(t^2)$.

A Figura 11 explicita a função `remove(tupla)`. Neste, o maior tempo se dá na linha 69, onde para cada tipo são adicionados todos os seus métodos como candidatos; podendo descrever a função como $O(t \times m_i)$, onde m_i se refere aos métodos do i-ésimo tipo.

Por fim, através da Figura 12 é possível perceber que o pior caso da função `esvazia(grafo)` se dá na linha 69, numa construção $O(t^2)$. Esta última permite afirmar que o algoritmo de acomodação de mudanças possui tempo polinomial com tempo $O(t^2)$, desprezadas as constantes.

Em contraste à solução força-bruta, que apresenta solução fatorial, a solução polinomial apresenta tempo inferior. Para fins de exemplificação, um experimento realizado com o algoritmo força-bruta fez com que sua execução não terminasse mesmo após cerca de 10 minutos de espera. Após isto, a mesma tupla de remoção submetida ao nosso algoritmo foi finalizada em cerca de 3 segundos. Apesar de pouco representativo, podemos afirmar, associando este dado à análise de complexidade, que o algoritmo proposto é satisfatório em seu tempo de execução.

Capítulo 4

Experimentação

Anteriormente, no Capítulo 2, apresentamos a fundamentação teórica que precede e se faz necessária para a compreensão deste trabalho. No Capítulo 3, introduzimos as técnicas de simulação que fazem uso dos modelos estatísticos que nos propomos a conceber. Este capítulo descreve o experimento de forma completa, desde a verificação da hipótese *power-law* à concepção dos modelos e sua utilização nas simulações numérica e aplicada. Adicionalmente, apresentamos os resultados obtidos.

4.1. Aspectos Metodológicos Formais

Esta pesquisa caracteriza-se como um estudo experimental e, como tal, propõe-se a observar o objeto de estudo (o processo de evolução de software), elaborar um modelo ou teoria comportamental e avaliá-lo mediante hipóteses implícitas pré-definidas (similaridades com o processo real). O modelo concebido neste trabalho é avaliado em forma de um estudo *in virtuo* de forma quantitativa e qualitativa.

As etapas que definem este experimento são:

- Seleção dos softwares de amostra para modelagem e simulação.
- Extração dos dados a partir das amostras.
- Geração dos modelos e distribuição e verificação da hipótese *power-law*.
- Geração dos modelos de regressão linear para simulação.
- Estudo de caso com simulação do software que apresenta os melhores modelos de distribuição e regressão linear.
- Avaliação da simulação frente à evolução real e a distribuição normal (quantitativa).

- Avaliação da simulação frente às Leis de Lehman (quantitativa).
- Avaliação da simulação frente às propriedades das redes livres de escala (qualitativa).

4.2. Amostragem

Processos de modelagem começam, obrigatoriamente, da coleta de informações sobre o domínio através de observações de amostras. Dada a enormidade de softwares existentes, bem como suas características totalmente diversificadas, emergem quatro questionamentos: *1) Quantos softwares são necessários para caracterizar adequadamente a população de software existente?*; *2) Se sabemos quantos, então quais softwares são apropriados como amostra para a modelagem?*; *3) Qual um número de versões mínimo de ser trabalhado para cada software?*; *4) Quais repositórios de versões utilizar?*

De fato, não há resposta precisas para tais perguntas, especialmente porque as **populações** em questão (conjuntos de software e evoluções existentes) são desconhecidas em tamanho e características. Estratégias que nos permitam escolher amostras aceitáveis se fazem necessárias, como descrevemos a seguir.

No que se refere à quantidade de softwares a serem trabalhadas, observou-se, desde o início desta pesquisa, o alto custo de tempo necessário para se processar o conjunto de versões que cada software possuía, ficando claro que o tempo desta pesquisa seria o fator limitante da amostra, conseguimos processar oito softwares para a pesquisa.

Com relação à segunda pergunta, seu apelo é quanto à qualidade da amostra escolhida. O primeiro passo é caracterizar um pouco melhor a população. Assim, foram aplicadas algumas restrições e definimos a população deste estudo como o conjunto de **software orientado a objeto,** escrito em **Java** e ***open source***. Nesta direção, optou-se pelos softwares hospedados no SourceForge[9], ordenados pelo critério de relevância do site que leva em consideração dados de atualizações e downloads dos usuários. Assim, o

[9] O SourceForge (http://www.sourceforge.net) é um dos mais populares servidores de código *open source* do mundo. Diversos sistemas de ampla utilização tal qual JUnit, Azureus e Emule possuem *branches* de desenvolvimento lá hospedados.

conjunto de oito softwares aqui explorados pode ser considerado representativo desta população.

No que tange os repositórios utilizados (Questões 3 e 4), os tipos de repositórios tradicionais (CVS e SVN) foram escolhidos em detrimento de soluções mais atuais, tal qual o Git[10]. A nosso favor temos que os repositórios mais antigos tendem a apresentar históricos com maior número de versões disponíveis. Assim, neste contexto elegemos os softwares que apresentavam histórico de versões relativamente grandes (sendo o menor o EasyMock com 183 versões), mas não demasiado grandes (sendo o maior o Spring Framework com 1524 versões). Um contraponto ao uso de grandes repositórios é que o volume de dados lá contidos, apesar de ricas fontes para a análise estatística, são relativamente difícel de manipular, pois contêm um número enorme de arquivos. De fato, não há um número adequado de versões, ou sequer um estudo similar com o qual pudéssemos comparar, uma vez que este, ao que sabemos, é o primeiro estudo que considera análise estrutural em cima de todo o histórico de versões de repositórios em código fonte. Por fim, resumimos a amostra deste estudo na Tabela 2 a seguir, ordenada pelo número de versões.

Tabela 2 – Resumo dos dados dos softwares de amostra.

Software	Tipo Rep.	#Versões	Data Início	Data Fim	Tamanho
EasyMock	SVN	183	08/2005	06/2010	180MB
JHotDraw	SVN	221	08/2006	06/2010	1.6GB
JUnit	CVS	332	02/2000	06/2010	350MB
Freemind	SVN	378	08/2000	06/2010	2.1GB
TuxGuitar	SVN	504	03/2008	08/2010	1.5GB
Jomic	SVN	923	01/2007	06/2010	3GB
JEdit	SVN	1087	11/2007	06/2010	5.1GB
Spring Framework	SVN	1524	03/2002	06/2010	6.6GB

O JUnit é o único software extraído de CVS pois foi o pioneiro. A experiência nos levou a adotar o SVN como padrão para os demais, uma vez que o CVS, ao contrário do SVN não mantém um registro único de versão para todos os arquivos, mas apenas registros individuais de cada arquivo. Isto se torna um problema quando é necessário definir onde inicia e encerra uma versão completa do software. Em particular, no caso do JUnit, a abordagem utilizada foi a proposta por Zimmermann e Weißgerber [ZW2004] .

[10] http://git-scm.com – The Fast Version Control System.

4.3. Extração dos Dados

Os dados extraídos apresentam-se em formato de uma tupla com 15 valores. A fim de exemplificação, parte do histórico do Spring Framework no modelo de tuplas é apresentado na Figura 13[11].

A partir dos dados extraídos foi possível visualizar evidências da hipótese *power-law*. A Figura 14 apresenta um exemplo desta curva de tendência. Para uma melhor visualização dos dados removemos o ponto 0, que contém um grande número de ocorrências, inviabilizando uma visualização adequada dos demais pontos.

ADD TIPO	REM TIPO	ADD METODO	REM METODO	ADD ATRIB.	REM ATRIB.	ADD THROWS	REM THROWS	ADD EXTENDS	REM EXTENDS	ADD IMPLEMENTS	REM IMPLEMENTS	METHOD CALLS
0	0	2	3	0	0	6	0	0	0	0	0	-1
1	0	8	2	2	0	6	0	2	0	0	0	2
2	0	2	2	0	0	0	0	2	0	0	0	0
1	4	7	7	0	0	2	0	8	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	1
1	4	9	13	0	2	0	0	0	0	0	0	-2
7	0	9	0	0	0	18	0	2	0	0	0	9
0	0	0	0	0	0	2	0	0	0	0	0	0

Figura 13 – O número líquido de *method calls*, em contraponto às demais variáveis.

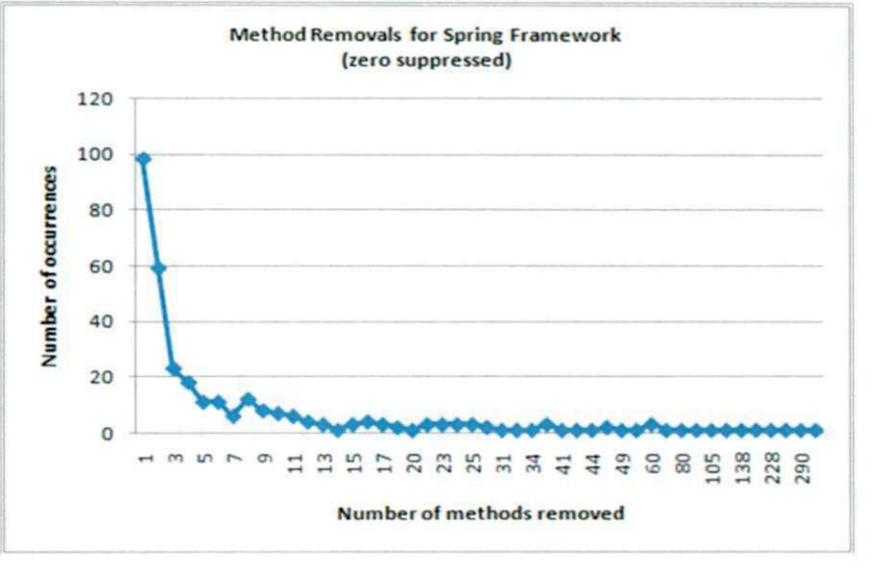


Figura 14 – Evidência visual da hipótese *power-law* para a remoção de métodos do software Spring Framework com o ponto zero omitido para melhor visualização dos demais pontos.

4.4. Modelo de Distribuições e Hipótese *Power-law*

[11] Todos os dados extraídos estão disponíveis em no site desta pesquisa [Dam2011] em formato *.csv* na sessão **Extracted Data.**

Anteriormente, afirmamos que procuramos estabelecer um modelo de distribuições para as entidades do software, a fim de validar a hipótese *power-law*, bem como prover um aporte inicial para a simulação. Para tal, utilizou-se a abordagem proposta por Clauset *et al.* [CSN2009]. Para cada uma das 8 variáveis relativas às entidades do código, objetivamos (através de *fitting tests*) definir os parâmetros da power-law que a modela: α e *x-min*, além do *p-value*, que verifica a hipótese de *power-law*. A **Erro! Fonte de referência não encontrada.** apresenta um resumo da análise das oito variáveis, para os oito softwares da amostra, totalizando 64 variáveis. Destas, 62 variáveis possuíam amostragem suficiente para o *fitting test*; as variáveis que não puderam ser modeladas são relativas aos pacotes do EasyMock, evidenciadas com fundo negro.

Percebe-se que não há um padrão evidente, seja para os valores de α ou de *x-min*, entre variáveis iguais em softwares diferentes. Esta falta de padronização sugere o óbvio: softwares distintos mudam de formas distintas. Por exemplo, a diferença da curvatura para a variável Add. Tipo entre o JUnit e Jomic (1.87 e 2.21), afirma que a curvatura do primeiro é menos acentuada, implicando em maior ocorrência de valores intermediários e um crescimento mais rápido deste.

Tabela 3 – Resultado dos *fitting tests* para cada uma das variáveis. Evidencia-se que 62 das 64 variáveis puderem ser modeladas.

	JUnit	Freemind	TuxGuitar	JHotDraw	Jomic	Spring Framework	JEdit	Easy Mock
Add. Pac.	1.83 – 1	1.69 – 1	1.62 – 2	2.05 – 1	3.35 – 1	2.76 – 1	1.92 – 14	-
Rem. Pac.	1.93 – 1	1.53 – 1	3.50 – 32	1.93 – 1	1.50 – 1	2.36 – 1	1.92 – 9	-
Add. Tipo	1.87 – 3	1.56 – 3	1.70 - 22	1.92 – 14	2.21 – 2	2.17 – 2	1.74 – 62	2.17 – 2
Rem. Tipo	1.59 – 1	1.50 – 2	1.67 – 22	1.91 – 8	2.21 – 1	2.23 – 3	1.71 – 46	1.95 – 1
Add. Atrib.	2.05 – 21	1.68 – 55	1.53 – 23	1.50 – 4	3.37 – 35	2.22 – 4	1.50 – 4	2.33 – 25
Rem. Atrib.	2.05 – 22	1.50 – 8	3.50 – 10	1.90 – 28	2.00 – 6	1.96 – 2	1.88 – 34	1.61 – 2
Add. Méto.	1.97 – 16	1.52 – 12	1.66 – 13	1.74 – 62	2.52 – 5	2.02 – 8	1.74 – 62	2.25 – 9
Rem. Méto.	1.65 – 5	1.51 – 15	3.50 – 20	1.73 – 48	2.18 – 3	2.10 – 20	1.76 – 46	1.50 – 1

Dando continuidade aos resultados do *fitting test*, evidencia-se que o fato de podermos aproximar curvas para os dados da amostra não implica que há um modelo para cada variável. Faz-se necessário verificar o *p-value* de cada modelagem, determinante para a confirmação, ou não, da hipótese *power-law*. A **Erro! Fonte de referência não encontrada.** sumariza estes dados, evidenciando os valores abaixo de 0,01, que descartam a hipótese. Assim, observa-se que a hipótese *power-law* se verifica em 52 das 64 variáveis exploradas. O passo seguinte desta abordagem consiste em confrontar estes resultados com outras distribuições – em particular, a exponencial, a

logarítmica e a Weibull. Apesar dos resultados destes testes não terem sido completamente satisfatórios, a ponto de descartar as outras distribuições, também não foi possível invalidar a hipótese original de *power-law* para nenhuma das 52 variáveis modeladas com sucesso[12].

Tabela 4 – Resumo do p-value gerado pelos fitting tests para as variáveis de cada um dos softwares da amostra. Fundo escuro identifica p-values inferiores a 0.1.

	JUnit	Freemind	TuxGuitar	JHotDraw	Jomic	Spring Framework	JEdit	Easy Mock
Add. Pac.	0.281	0.046	0.710	0.564	0.875	0.866	0.927	-
Rem. Pac.	0.711	0.000	0.908	0.412	0.000	0.806	0.911	-
Add. Tipo	0.800	0.221	0.022	0.914	0.433	0.107	0.721	0.143
Rem. Tipo	0.453	0.358	0.002	0.929	0.124	0.920	0.750	0.339
Add. Atrib.	0.180	0.432	0.075	0.526	0.895	0.795	0.530	0.373
Rem. Atrib.	0.623	0.263	0.796	0.926	0.040	0.587	0.814	0.810
Add. Méto.	0.978	0.306	0.020	0.709	0.786	0.176	0.721	0.246
Rem. Méto.	0.019	0.166	0.682	0.768	0.685	0.675	0.750	0.027

Observa-se que as distribuições *power-law* são uma boa hipótese para a maioria das variáveis observadas. E quanto às demais? Este resultado sugere que há diferentes distribuições para diferentes softwares? Em parte, a resposta é não.

No campo da análise estatística, o maior problema se refere à falta de dados amostrais para suportar os resultados. Clauset *et al.* [CSN2009] argumenta que amostras com número de observações menores que 100 podem enviesar os resultados dos *fitting tests*, seja encontrando modelos para variáveis que não o obedecem, ou não o encontrando quando de fato eles se aplicam. Estes problemas com pequenas amostras ocorrem porque dentro de suas limitações, poucos pontos ainda são capazes de descrever tendências, que podem ser confirmadas ou não com a adesão de novos pontos.

Traçando um paralelo entre a contagem de observações para cada variável e seus *p-values*, conforme apresentado na **Erro! Fonte de referência não encontrada.**, emergem evidências que 6 destas variáveis não foram modeladas devido a escassez da amostra, são elas: Freemind – Adição e Remoção de Pacotes; Jomic – Remoção de Pacotes; EasyMock – Adição e Remoção de Pacotes; EasyMock – Remoção de Métodos.

Tabela 5 – Paralelo entre o número de observações amostrais de cada variável e p-value.

	JUnit	Freemind	TuxGuitar	JhotDraw	Jomic	Spring Framework	JEdit	EasyMock
Add. Pac.	8 - 0.281	4 - 0.046	29 - 0.710	6 - 0.564	3 - 0.875	6 - 0.459	6 - 0.927	0 - ?
Rem. Pac.	6 - 0.711	4 - 0.000	29 - 0.908	5 - 0.412	2 - 0.000	5 - 0.809	5 - 0.911	0 - ?
Add. Tipo	27 - 0.800	25 - 0.221	63 - 0.022	15 - 0.914	14 - 0.433	22 - 0.265	15 - 0.721	3 - 0.143
Rem. Tipo	23 - 0.453	17 - 0.358	54 - 0.002	14 - 0.929	7 - 0.124	17 - 0.908	12 - 0.750	7 - 0.339

[12] Os resultados completos da modelagem e testes de hipótese frente a outras distribuições para as 54 variáveis estão disponíveis no site da pesquisa [Dam 2011], na seção **Models**.

Add. Atrib.	57 - 0.180	55 - 0.432	77 - 0.075	19 - 0.526	46 - 0.895	25 - 0.606	19 - 0.530	15 - 0.373
Rem. Atrib.	49 - 0.623	38 - 0.263	64 - 0.796	17 - 0.926	39 - 0.040	21 - 0.616	16 - 0.814	13 - 0.810
Add. Méto.	51 - 0.978	16 - 0.306	93 - 0.020	13 - 0.709	21 - 0.786	59 - 0.433	13 - 0.721	15 - 0.246
Rem. Méto.	45 - 0.019	18 - 0.166	73 - 0.682	14 - 0.768	15 - 0.685	47 - 0.686	**13 - 0.750**	5 - 0.027

No que se refere ao TuxGuitar, suas 4 variáveis possuem amostragens relativamente altas, mas podem ser explicadas sob outra perspectiva. O TuxGuitar apresenta um histórico de desenvolvimento atípico, contendo vários *commits* grandes. Posteriormente foi observado que isto acontece devido ao desenvolvimento paralelo em *branches* (linhas paralelas de desenvolvimento), cujos resultados são continuamente transferidos para o *main trunk* (linha principal de desenvolvimento), enviesando a distribuição das mudanças, e invalidando, assim, a hipótese *power-law*.

As últimas duas variáveis são a Remoção de Métodos do JUnit e a Remoção de Atributos do Jomic. Estas duas variáveis, em particular, não puderam ser explicadas. Foram explorados *fittings tests* com as distribuições relacionadas (Exponencial e Logaritmica) sem sucesso. Uma vez que o número mínimo de amostras para uma avaliação adequada é 100, é possível que pontos de amostra adicionais permitam uma modelagem.

4.5. Modelo de Regressão Linear

A série de resultados a respeito do modelo de distribuições *power-law* acima vai ao encontro de vários trabalhos que visam detectar padrões na evolução de software [BKS2003, MHPB2009, XCN2009, WWY+2009]. No entanto, para fins de simulação, conforme discutimos brevemente no Capítulo 3, os modelos lineares são, de fato, cruciais, uma vez que há forte dependência linear entre as variáveis de interesse.

Cada software possui um conjunto de modelos lineares para as suas variáveis, e a definição de tais modelos demanda um esforço em combinar variáveis conhecidas como preditoras de variáveis resposta ainda sem modelo. É possível presumir que combinar 15 variáveis 2 a 2, depois 3 a 3, etc. é inviável, e algumas heurísticas foram utilizadas. Por exemplo, assume-se que uma variável de adição é boa preditora de outras variáveis de adição, a exemplo da adição de classes como preditora da adição de métodos (o mesmo se aplica para remoções). Assim, o processo inicia-se com duas variáveis de comportamento conhecido: a **adição de tipos** e a **remoção de tipos** (ambas *power-laws* com modelo de distribuição previamente definidos). Intuitivamente, as entidades **tipo** são as mais relevantes no contexto estrutural de qualquer software, além de que, apesar de não ter havido um critério formal para tal, foi realizada uma breve

análise utilizando adição/remoção de métodos e atributos como variáveis iniciais (lembrando que temos o modelo de *power-law* de todas estas as variáveis) e os resultados não foram tão satisfatórios.

A Figura 15 apresenta a modelagem completa para o Spring Framework; seus modelos são os mais satisfatórios dentre os oito softwares, pois seus valores de R^2 (que definem o quanto dos dados amostrais são explicados pelo modelo) estão, em média, acima dos demais[13]. Cada linha descreve o modelo estatístico de uma variável.

```
packadd           : ml : typeadd * 0.0965359 - 0.0693298
packrem           : ml : typerem * 0.052921 + 0
typeadd           : pl : 2.17; 2; 41
typerem           : pl : 2.23; 3; 41
methodadd         : ml : fieldadd * 4.04226 + 1.37362
methodrem         : ml : fieldrem * 3.64045 + 1.54624
fieldadd          : ml : typeadd * 1.1646 + 0
fieldrem          : ml : typerem * 1.70719 + 0
relthrowsadd      : ml : methodadd * 0.4792 + 1.3672
relthrowsrem      : ml : methodrem * 0.2352 + 1.941
relimplementsadd  : ml : methodadd * 0.04526 + 0
relimplementsrem  : ml : typerem * 0.191 + 2.41
relextendsadd     : ml : typeadd * 0.634013 + 0.412462
relextendsrem     : ml : typerem * 0.1412 - 0.088
relmethodcalls.   : ml : packadd * 24.98468 + 0
```

Figura 15 – O conjunto de modelos das variáveis do Spring Framework, onde se evidencia os modelos de regressão linear (*ml*) e os modelos de power-law (*pl*).

O formato apresentado é o mesmo utilizado como entrada para o simulador. A semântica das linhas é a seguinte: o primeiro valor explicita a variável que está sendo modelada. O termo seguinte refere-se ao modelo de distribuição *power-law* (*pl*) ou o modelo de regressão linear (*ml*). Nos modelos de *power-law* os 3 parâmetros seguintes são, nesta ordem: α; *x-min*; e *x-max*, um limite superior para a simulação dos valores de uma dada variável. Seu valor é definido como o maior valor da amostra daquela variável no histórico do software, excluídos os *outliers*. Há um dado omitido por questões de espaço. Valores abaixo do *x-min* são descritos em termos de sua ocorrência percentual, assim:

```
typeadd : 0=>68%; 1=>12%; pl : 2.17; 2; 41
```

Indicando que zero tipos devem ser adicionados 68% das vezes, um tipo 12% das vezes, e o resto da simulação deve obedecer uma *power-law* com *x-min* igual a 2 e seus demais parâmetros.

[13] Os modelos lineares para todos os sistemas e seus valores de R^2 então disponível na página de resultados da pesquisa.

Por fim, as linhas que se referem aos modelos lineares discriminam as variáveis preditoras, e os dois parâmetros que definem o modelo.

4.6. A Simulação da Evolução de Software

Para fins de avaliação dos modelos deste trabalho conduzimos dois experimentos distintos para a simulação da evolução do software Spring Framework: a Simulação Numérica e a Simulação Aplicada. A simulação Numérica possibilitou duas análises distintas de seus resultados. Primeiro, comparamos os dados gerados através dos modelos de *power-law* e regressão linear com um modelo de distribuição uniforme e com os dados reais. Na segunda análise, os dados de simulação são confrontados com as Leis de Lehman. Com relação à Simulação Aplicada, analisamos as implicações de incluir as mudanças simuladas na estrutura real do software frente às propriedades de redes livres de escala de software.

O momento de partida para a simulação foi um instante do Spring Framework 500 versões antes da última versão estudada (datada de Junho de 2010). O objetivo é simular e comparar 500 versões do modelo aqui proposto, com 500 versões simuladas para o modelo uniforme e as 500 mudanças reais. Este número de versões é suficientemente grande, pois, apesar deste software possuir mais de 1500 versões, apenas 770 destas incluem alguma das mudanças estruturais que observamos, assim, quando geramos 500 versões estamos, na prática, simulando 64,9% do histórico de mudanças do Spring Framework, um número bastante representativo. Para minimizar anomalias provenientes de uma única geração, a geração de 500 versões foi repetida 50 vezes, e a média dos valores encontrados foi utilizada como resultado da Simulação Numérica em ambos os modelos. Cada uma das execuções do experimento em suas 500 versões geradas tomou em torno de 30 segundos do simulador.

Ainda, optamos por tomar 50 dos 500 pontos de observação, cada ponto sumariza os dados de 10 versões. Esta particularidade visa diminuir o número de pontos de análise por questões visuais e ainda garantir um número de amostras suficientes para algumas validações estatísticas. Na prática, 30 pontos é um limiar inferior aceitável, mas 50 é um valor melhor, pois é estatisticamente adequado e não é demasiado grande de observações a ponto de complicar a análise. Ainda, esta abordagem adéqua-se à análise de ao menos uma lei de Lehman, que trata de versões de *release*, conceitos até então ignorado. A seguir, descrevemos os resultados deste esforço.

4.6.1. Modelo Estatístico × Modelo Uniforme × Mudanças Reais.

A comparação entre o modelo fruto deste trabalho, o modelo uniforme e mudanças reais possui dois objetivos: (i) apresentar a fraqueza da abordagem *ad hoc* para mudanças estruturais, mesmo que de forma qualitativa; e (ii) demonstrar a capacidade de previsão dos modelos de distribuição *power-law* associados às regressões lineares. Esta comparação dar-se-á através do número total de tipos e relacionamentos em cada ponto de observação que contempla a soma de 10 versões. A Figura 16 apresenta o primeiro resultado da simulação, uma comparação entre a simulação utilizando os modelos estatísticos aqui descritos, e os dados reais onde fica evidente a proximidade entre as curvas. Apesar de o número final de módulos diferir em 25,9% do original (306 da simulada contra 243 da real), fica evidente a similaridade entre as curvas. Curiosamente, a maior diferenciação ocorreu nas últimas 50 versões, pois até a versão 450 a diferença era de apenas 2,95% (244 da simulada contra 237 da real). No entanto, não há evidências que simulações adicionais produzirão resultados melhores, pois o comportamento da simulação é relativamente previsível, ao contrário da evolução real, que praticamente estagnou o crescimento nas últimas 50 versões.

A Figura 17 apresenta a visão completa deste processo, onde incluímos a geração utilizando a distribuição uniforme com *x-max* igual ao nosso modelo estatístico. Optamos por apresentar uma figura separada, pois os valores da distribuição uniforme impossibilitam visualização adequada dos demais. Nota-se que esta abordagem produz resultados totalmente destoantes do que se espera, com uma explosão de novos módulos que supera o número real na ordem de 159 vezes. É pouco provável que um pesquisador produzisse tão desastrosa sequência de mudanças estruturais, mas que optasse por parar o experimento após os primeiros passos.No entanto, esta abordagem somente sugere que realizar mudanças de forma discriminada, sem um critério formal tal qual um modelo estatístico, pode desencadear fenômenos bizarros, tal qual este crescimento desonerado.

O passo complementar a análise dos módulos trata do número total de novos relacionamentos gerados no processo simulatório em face ao real. Optou-se, na Figura 18, por não apresentar os dados referentes ao modelo uniforme que são, novamente, totalmente irreais, até mesmo sem mérito para serem discutidos. Conforme se pode observar, também neste caso, percebe-se a similaridade e proximidade entre as curvas.

Para as relações, o valor total simulado supera o real em 31,5% (5574 da simulada contra 4237 da real).

Uma vez que a adição e remoção de relações são funções das adições e remoções de entidades, e sabendo que os tipos simulados superaram os reais em mais de 25,9%, torna-se perfeitamente compreensível que este resultado se reflita nas relações. Uma vez que se adicionam mais tipos, mais métodos e atributos são também adicionados, bem como todas as relações que os organizam.

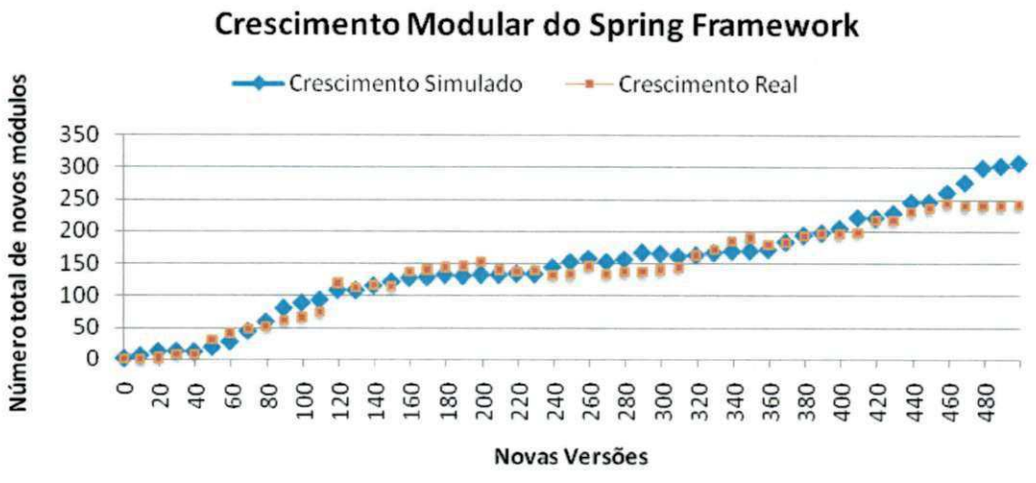


Figura 16 – O crescimento do total de módulos do Spring Framework simulado com o modelo estatístico e o real, após 500 versões. Não é considerado o número de módulos anteriormente existentes.

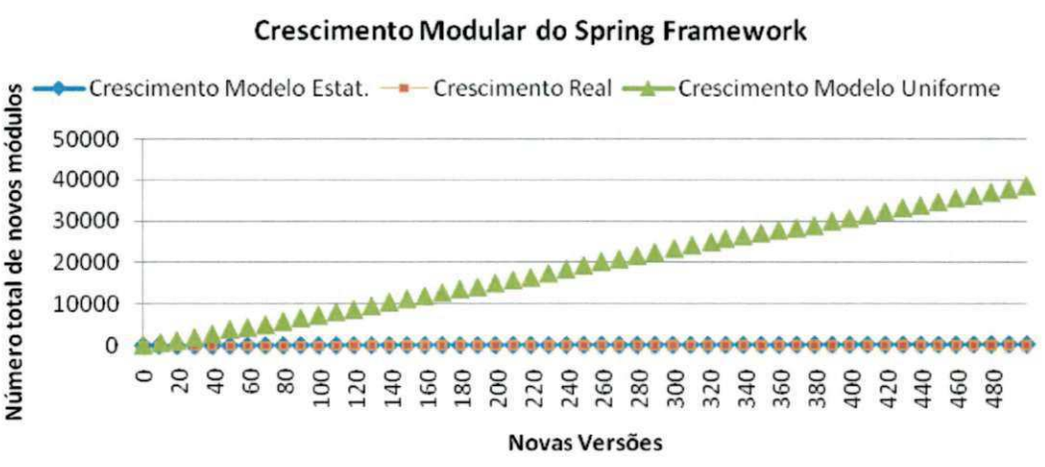


Figura 17 – O crescimento do total de módulos do Spring Framework. Nesta, apresenta-se a explosão de módulos gerados pela distribuição uniforme.

A técnica mais simples para aprimorar a simulação seria fortalecer a regra de *outliers* na amostra. Teríamos, assim, limites inferiores (*x-max*) menores e o simulador

geraria valores mais próximos aos reais. Porém, esta abordagem é enormemente tendenciosa, e optou-se por manter o critério original de eliminação de *outliers*.

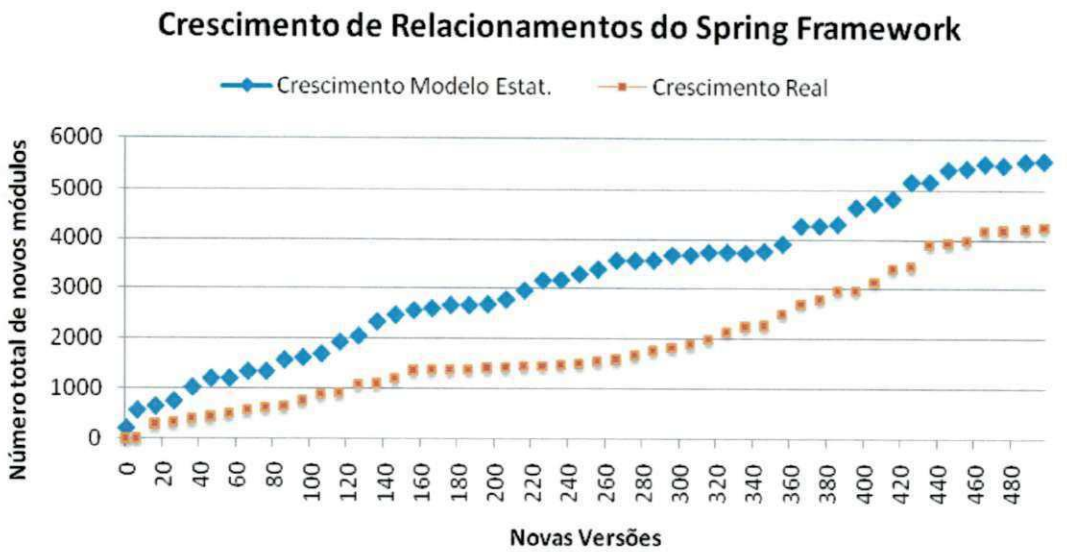


Figura 18 – O crescimento do total de relações do Spring Framework simulado com o modelo estatístico e o real, após 500 versões. Não é considerado o número de relações anteriormente existentes.

4.6.2. Modelo Estatístico × Leis de Lehman

As métricas utilizadas para a análise do Spring Framework foram discutidas no Capítulo 2. A partir dos resultados previamente apresentados nas Figura 16 e Figura 18, três leis foram observadas: **I) Mudança Contínua** – pois o software está em constante processo de mudança e não há evidência contrária a isto; **II) Complexidade Crescente** – pois, ao menos do ponto de vista estrutural, o software está incorporando módulos e mais relações entre eles; e **VI) Crescimento contínuo** – as curvas apresentam, claramente, o crescimento do número de módulos e seus relacionamentos. Discutiremos a seguir os resultados para as demais leis.

III) Auto-Regulação – Espera-se que curvas de crescimento do software apresentem oscilações positivas e negativas. As figuras anteriores não são adequadas para verificar esta lei, pois se trata do resultado acumulativo. Este experimento é um pouco diferente do anterior, pois os dados aqui utilizados demandam certo cuidado: (1) Aqui tomaremos o número líquido de módulos adicionados a cada versão (número de adições subtraído do número de remoções); (2) Consideraremos as 500 microversões de *commit*, pois o somatório de 10 versões, tal qual apresentado anteriormente, tende a ser

positivo pela sexta lei; (3) Por fim, não é tomado o resultado da média das 50 execuções da simulação, mas o resultado de uma rodada qualquer do simulador (em particular a de número 50), pois na média o número líquido tende a ser positivo em todos os pontos.

Na Figura 19 apresentamos as variações no número líquido de módulos adicionados retiradas do histórico real do Spring Framework. Estas constituem as oscilações sugeridas por Lehman como indícios da auto-regulação. A Figura 20 confirma que o mesmo comportamento está presente nos dados simulados, e de forma mais acentuada, pois os picos positivos e negativos são mais representativos que no original. Além disto, percebe-se a predominância de picos positivos, o que também justifica o maior crescimento em número de módulos quando da simulação. Optou-se por duas figuras para não sobrepor os dados, mas a escala foi mantida para melhor apreciação visual.

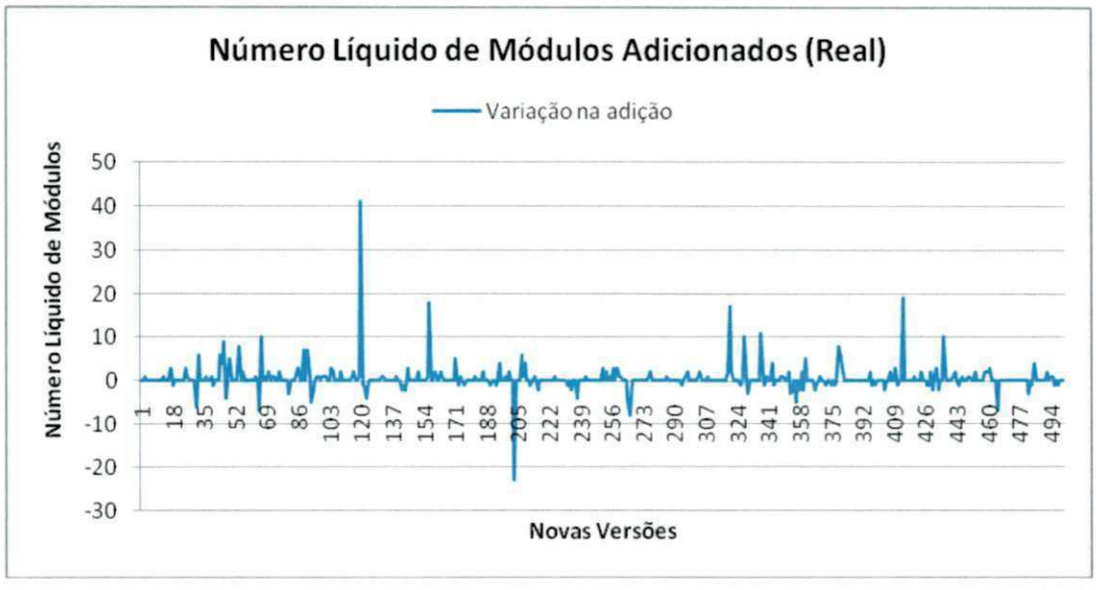


Figura 19 – Variação no número líquido de módulos adicionados versão após versão da evolução real **do Spring Framework.**

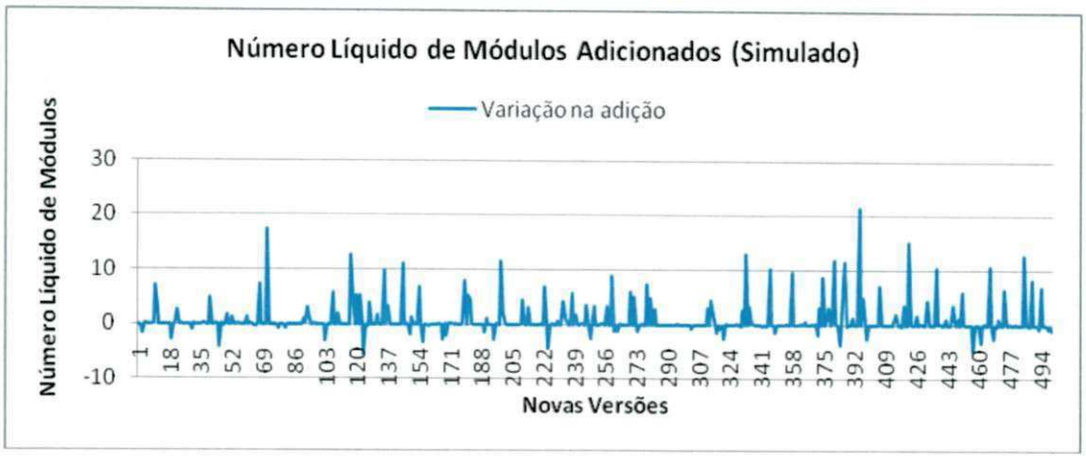


Figura 20 – Variação no número líquido de módulos adicionados versão após versão da evolução simulada **do Spring Framework.**

IV) Conservação da Estabilidade Organizacional – Espera-se que o número de mudanças entre versões de release permaneça constante. Neste estudo, esperamos que a soma de todas as mudanças sugeridas pelas tuplas sejam sempre similares, para cada um dos 50 pontos observados. Observemos a Figura 21, que contém os dados referentes ao esforço entre pseudo-*releases* (somas de 10 versões) do Spring Framework. Apesar de apresentarem comportamento oscilatório, parece-nos aceitável supor que há certa estabilidade entre os pontos. Uma das maiores discrepâncias se dá no final do histórico real do Spring Framework, onde estão presentes pontos que evidenciam um maior esforço da equipe de desenvolvimento nas suas últimas versões (próximo à versão 40). Porém, em seguida parece se estabilizar novamente. Já a versão da simulação se apresenta mais estável que a real, e mais próxima do conceito teórico. Uma possível razão para esta diferença entre real e simulado pode ser dada pelas limitações do modelo linear. De fato, os valores de R^2 dos modelos lineares ficam entre 0,7 e 0,99, indicando que este só modela entre 70 e 99% dos valores amostrais. Assim, é possível que o modelo não seja capaz de gerar tuplas repletas de valores altos, próximos dos *outliers*. São estas tuplas que produzem valores exagerados para o número total de mudanças, tais quais os contidos no histórico real.

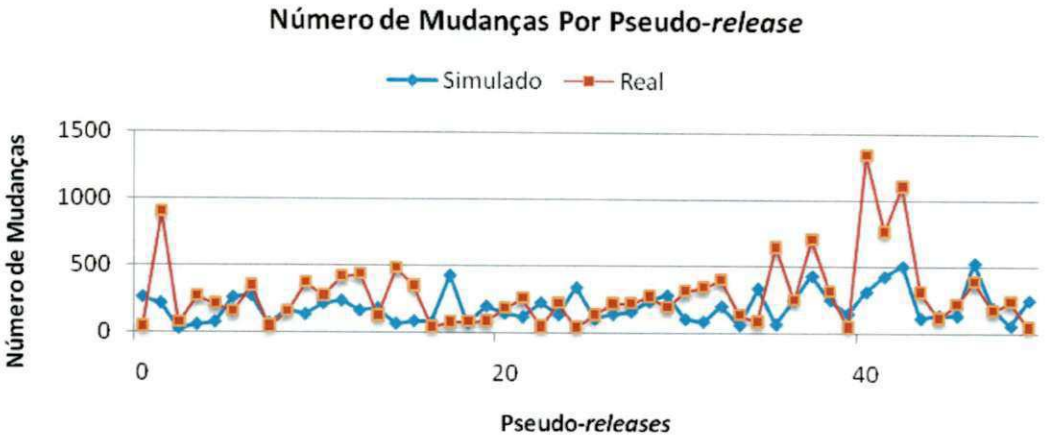


Figura 21 – O número de mudanças realizadas por pseudo-release do Spring Framework. Apesar de apresentar variações, a tendência dos pontos sugere uma reta em ambas as evoluções.

VI) Conservação da Familiaridade – O crescimento do número de módulos dos software apresenta um crescimento extremamente próximo da sua linha de tendência linear, conforme observa-se na Figura 22. Esta linha sugere que o crescimento do software acontece em uma constante, confirmando a lei. Observa-se que os pontos apresentam-se acima e abaixo de suas respectivas curvas de tendência linear. No entanto, a proximidade à reta é bem preservada.

No que tange o corolário de que mudanças grandes são seguidas por mudanças pequenas, as Figuras 19 e 20 são evidências suficientes de que esta propriedade se preserva, especialmente nos dados reais.

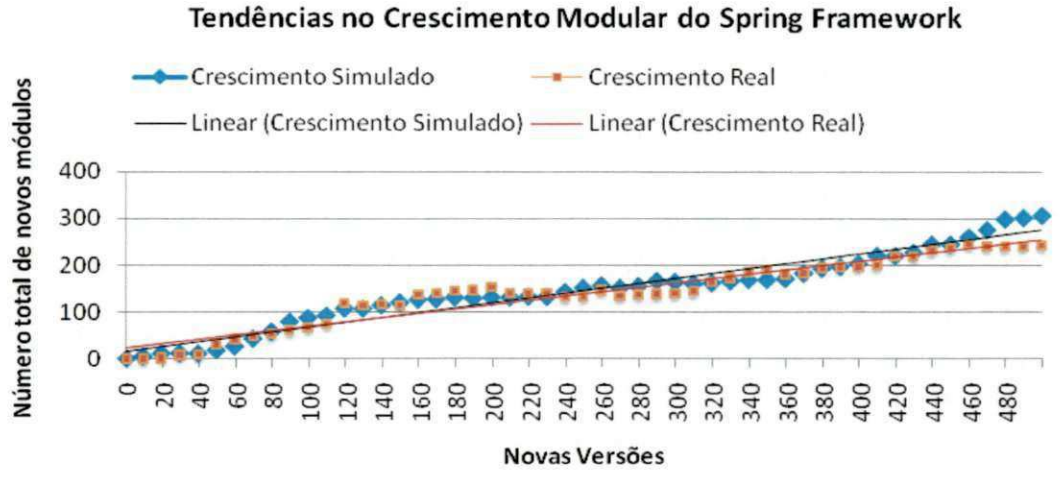


Figura 22 – Crescimento modular do Spring Framework, com linhas de tendência para ambas as evoluções.

VII) Qualidade Decrescente – Espera-se que o número de funções modificadas cresça entre *releases*. Conforme dito anteriormente, no nosso caso trabalhamos com o conceito de pseudo-*release*. Ainda, as informações estruturais que possuímos para definir quantas funções (métodos) foram modificadas é apenas o número de adições e remoções, e esta é a nossa referência para a métrica de qualidade decrescente. A Figura 23 a seguir apresenta os resultados desta métrica. Percebe-se que não há evidências de crescimento, especialmente pela similaridade com os pontos das mudanças totais (Figura 21).

Os resultados acima apresentados evidenciam uma limitação do modelo da seguinte forma: apesar de não existir uma definição final sobre o que são as mudanças em funções, podemos assumir que uma abordagem apropriada levaria em consideração mudanças no nível de *statements*, ou de *method calls*, que são as mudanças mais comuns. No entanto, nosso modelo baseado em adições e remoções, e limitado quanto às *method calls* impossibilita tal abordagem mais adequada. A métrica aqui utilizada é limitada, mas possivelmente a única forma de realizar a tarefa, dadas as variáveis que compõem o nosso modelo.

Figura 23 – O número de funções modificadas por pseudo-release do Spring Framework. De forma similar às curvas de mudanças totais, não há evidências de crescimento, mas de constância.

VIII) Sistema de Feedback – Espera-se que o número de módulos, versão após versão, seja similar ao que será possível conceber através do modelo exponencial conhecido:

$$S = a \times \sqrt[s]{\#R} + b$$

Conforme dito, faz-se necessário estimar os parâmetros a e b do modelo. A base do cálculo é o valor de $\#R^{1/3}$ proposto, possibilitando uma regressão linear para estimar os parâmetros. Como trabalhamos com pseudo-*releases*, ao invés de utilizarmos o valor 270 (ponto inicial da geração de 500 versões do experimento), utilizamos 27. E a última pseudo-*release* será a de número 77, portanto. Assim, os dados para a regressão linear estão organizados de tal forma que inicialmente #R é igual a 27 e S = 89 (o número de tipos existentes nesta versão do Spring Framework), e assim por diante. Os parâmetros estimados foram $a = 29{,}1$ e $b = 1{,}66$ e o valor de R^2 ficou igual a 0,95. O que indica que o modelo estimado é capaz incorpora 95% das amostras. Valores desta escala são considerados extremamente satisfatórios. Assim, a evolução real do Spring Framework revela-se um sistema de *feedback* igual ao esperado, no que tange o número de módulos dado o número do *release*. De forma similar, foi possível estimar parâmetros para a evolução simulada, com valores de $a = 33{,}2$ e $b = 5{,}726$ e o valor de R^2 ficou igual a 0,87. Apesar de menor que o original, este valor também é visto como satisfatório do ponto de vista estatístico, confirmando que a simulação também correspondeu ao sistema de *feedback* conforme esperado.

4.6.3. Simulação Aplicada × Redes Livres de Escala

Souza [Sou2010], em seu trabalho de dissertação de mestrado, definiu um critério de classificação para rede de estruturas de software frente à teoria das redes complexas livres de escala, de tal forma que seja possível distingui-la de diversos outros tipos de redes complexas.

Além do suporte teórico para tal, o autor disponibiliza ferramental adequado para a checagem de estruturas de software quaisquer, desde que a estrutura seja convertida para uma notação específica, onde os nomes das entidades são organizados em pares, indicando haver uma relação entre ambas, no sentido da esquerda para a direita. Por simplicidade, o autor não considera tipos de relacionamentos, e trabalha com classes.

No Capítulo 3 descrevemos o processo da Simulação Aplicada, e como este modifica a estrutura original a fim de comportar a sugestão de mudança proveniente da Simulação Numérica. Esta mudança procura preservar a propriedade das redes livres de escala, mas de forma limitada. Realizamos, assim, um experimento que consistiu em

manipular a saída de cada iteração da Simulação Aplicada, convertendo a estrutura gerada para o formato acima explicitado. Há perda de informação, dado que métodos e atributos são ignorados. No entanto, todos os relacionamentos são preservados, visto que relacionamentos entre métodos podem ser traduzidos como relacionamentos entre tipos. Por exemplo, se uma classe C1 possui um método m1(), e este método chama o método m2() da classe C2, é possível abstrair este relacionamento entre métodos como um relacionamento dirigido entre as classes C1 e C2. Esta abordagem se aplica a quaisquer relacionamentos que envolvam subentidades, tais quais os métodos.

Este experimento constitui-se, portanto, de uma série de 500 assertivas sobre o fato de uma dada versão simulada preservar a propriedade de uma rede livre de escala de software ou não. Idealmente, o resultado desta tarefa seriam 500 resultados com valor *verdade*. No entanto, observou-se que, apesar da versão inicial do Spring Framework preservar a propriedade, bastaram algumas iterações da Simulação Aplicada para que a estrutura fosse corrompida. Apesar de o processo ter sido repetido, todas as rodadas produziram resultados similares aos apresentados na Figura 24, onde, para as 500 versões, destacamos aquelas que preservam a estrutura.

Figura 24 – Ocorrência de versões com propriedade preservada na simulação. Valor igual a 1 indica a ocorrência; 0, a não ocorrência.

Pode-se perceber claramente que a propriedade estrutural se preserva nos momentos iniciais da simulação, sendo corrompida quando da manipulação inadequada da estrutura pelo algoritmo de acomodação de mudanças. Em particular, no caso acima apresentado, há 32 versões inicialmente estáveis, o que sugere certa flexibilidade no modelo de rede de livre de escala de software. Percebe-se que após a estrutura ser corrompida (versão 33), permanece assim até que uma das modificações do algoritmo produz uma estrutura novamente correta (versão 47). Esse comportamento se repetiu várias vezes em diversas rodadas do experimento. Assim, quer seja por competência do

modelo ou mérito da aleatoriedade, a rede consegue se reestabelecer. Novamente, a estrutura é corrompida e segue assim indefinidamente. Curiosamente, nesta rodada, a versão 338 evidencia outro reestabelecimento. Este resultado é raro, e aconteceu apenas em 2 de mais de 20 execuções. O sentimento é que o algoritmo de acomodação de mudanças corrompe continuamente a estrutura do software tornando-a cada vez menos reconhecível como tal. A princípio, é possível acreditar que – uma vez que a estrutura se reestabeleceu em uma versão intermediária –, ele não a degrade de forma tão drástica, ou seja, que ela permanece próxima ao que seria uma organização correta. Porém, o fato é que até o momento não há como confirmar ou negar esta hipótese, visto que a pesquisa original de Souza [Sou2010] não incorpora a análise de imprecisões desta natureza.

Um resultado adicional se refere à mesma experimentação com o Spring Framework em suas últimas 500 versões reais, apresentado na Figura 25 – Ocorrência de versões com propriedade preservada no Spring Framework. Valor igual a 1 indica a ocorrência; 0, a não ocorrência.. Nestas, também se percebe a perda da propriedade, mas em versões esporádicas: 54 a 60, 69 a 71, 90, 116, 168 e 169, 404 a 420 e 485. Uma breve análise dos *commits* envolvidos nas gerações destas versões não demonstrou atividades atípicas. Portanto, não foi possível precisar a razão de suas ocorrências. Conclui-se, portanto, que o modelo apresenta limitações neste sentido e não se adéqua apropriadamente às propriedades de RLE.

Figura 25 – Ocorrência de versões com propriedade preservada no Spring Framework. Valor igual a 1 indica a ocorrência; 0, a não ocorrência.

Capítulo 5

Trabalhos Relacionados

Este trabalho caracteriza-se pela concepção de um modelo puramente quantitativo e sua aplicação em simulação. A seguir, explicitamos as principais contribuições dos trabalhos que diretamente se relacionam a este, desde a introdução das pesquisas em evolução de software ao atual viés de aplicação de modelos em simulação[14].

5.1. Estudos em Evolução de Software

Atribui-se a Lehman e Belady [Leh1969, LB1976, Leh1996] o conjunto de observações iniciais que explicitam os primeiros padrões de mudanças em software, em especial, as que tratam da mudança contínua e do aumento da complexidade se destacam. Trabalhos posteriores de Lehman expandiram este conjunto estas observações, consolidando um conjunto de leis que teoricamente regem a evolução de software.

Tamai e Torimitsu [TT1992] observaram que o processo de degradação até o fim do software não é apenas decorrente de sua má manutenção, mas também de outras características organizacionais, tal qual a substituição do software. Em uma pesquisa de questionário realizada com 42 empresas japonesas e as conclusões mais relevantes dão conta que softwares de pequena escala sobrevivem menos tempo que softwares grandes; softwares de administração duram mais tempo que softwares voltados ao negócio da

[14] Apesar de termos conhecimento do vasto referencial teórico das décadas de 1980 e 1990, contemporâneos aos trabalhos da concepção das Leis de Lehman, o esforço em obtê-los mostrou-se infrutífero, e poucos trabalhos foram obtidos. A revisão bibliográfica privilegia, assim, os anos 2000.

empresa; softwares crescem nas trocas, ou seja, a cada troca, um software maior e com mais funcionalidades emerge.

Em sua tese de doutorado, Opdyke [Opd1992] cunha o termo refatoramento, e racionaliza os primeiros padrões de mudanças estruturais em software, em particular, aquelas que não alteram a semântica do código. Em seguida, Martin Fowler [Fow1999] apresenta um catálogo de refatoramentos, sugerindo a obrigatoriedade na aplicação de mudanças estruturais que visam aprimorar qualidades como legibilidade do código e flexibilidade do design.

Kemerer e Slaugher [KS1997] realizaram um estudo observacional sobre 621 módulos de 5 softwares distintos em evolução. Seus resultados evidenciaram que apenas pequena parte destes módulos era continuamente alterada. Em particular, observou-se que módulos estratégicos são continuamente **aumentados**; módulos complexos, relativamente antigos e grandes são **reparados**; e módulos grandes e antigos são **reestruturados** e **aprimorados** com maior frequência.

Fatores humanos que influenciam no desenvolvimento de softwares também foram diretamente observados. Cusumano e Yoffie [CY1999] evidenciaram características comerciais como fatores de influência na concepção de novas versões do software, tal qual a satisfação de clientes com versões preliminares de softwares. Já Perry *et al.* [PSV2001] apontam que confusões geradas no desenvolvimento devido a dinâmicas entre os programadores (tal qual programação em paralelo) podem afetar a evolução do software.

Graves *et al.* [GKM+2000] e Arisholm and Briand [AB2006], indicam que modificações prévias em um arquivo são bons indicadores de que este arquivo será modificado no futuro novamente.

Barry *et al.* [BKS2003], apresenta alguns padrões de evolução baseado em uma métrica própria, denominada volatilidade de software, através da qual detectou-se quatro padrões comuns para o conjunto de softwares observados. A métrica é composta, entre outros, pelos tamanhos das mudanças aplicadas ao software entre suas versões oficiais.

Le Goaer e Ebraert [LE2007] observaram adequadamente que padrões de evolução (ou de mudanças) estão além das mudanças que preservam comportamento.

Em seu trabalho os autores definem o termo estilos de evolução (*evolution styles*), que são modificações estruturais que podem ser definidas uma vez e aplicadas várias vezes a um software. Um exemplo de estilo de evolução apresentado seria a adição de uma funcionalidade de criptografar um texto. Os autores acreditam que através de estilos de evolução é possível racionalizar toda a evolução de um software. Em seu trabalho apresentam dois exemplos de uma evolução sistematizada através destes estilos. O fato é que o estudo é simples e carece de boa experimentação. Por fim, ao que tudo indica os autores conceberam um conjunto de estilos evolucionários em estudos posteriores e, em especial, focaram em evolução arquitetural; porém, não conseguimos acesso a tais trabalhos.

A fim de verificar as leis de Lehman, Xie *et al.* [XCN2009] investigou um total de 69 anos de evolução para sete softwares *open source* em suas versões oficiais. Os autores concluíram que as leis referentes às propriedades estruturais (*e.g.* crescimento) se verificavam, enquanto outras dependiam de uma interpretação mais leniente para serem validadas. A segunda contribuição explicita que um total que varia entre 20 e 40% das funções recebe mais de dois terços das operações de mudanças. Este dado mostra haver uma concentração exacerbada e cíclica de erros em partes específicas do código. Também é demonstrado que interfaces mudam muito menos frequentemente que implementações.

Em seu trabalho, Ajila e Alam [AA2009] propuseram um framework teórico para a concepção de modelos de evolução de software através da utilização de uma linguagem formal, a saber: uma extensão da linguagem de descrição de restrições OCL denominada CAL.

Em sua tese de doutorado, Araújo [Ara2010] propõe um modelo que relaciona diretamente métricas das leis de Lehman e gera relações de implicância entre estas, a fim de observar e gerar um modelo de evolução para um dado software sob observação. Por exemplo, são criadas associações lógicas que podem afirmar que caso o **tamanho** permaneça constante, mas a complexidade e o **esforço** em manter o software cresçam, a lei da autorregulação está sendo verificada no software em questão. A verificação de leis complexas como a que se refere ao declínio da qualidade do software também estão presentes. Para esta definiu-se, dentre outras, que **complexidade** estável, **esforço** estável, **confiabilidade** crescente e **manutenabilidade** decrescente implicam em

declínio da qualidade do software. Desta forma, concebe-se todo um modelo de dinâmica de software baseado em seu histórico recente de mudanças.

Notadamente, todos os modelos acima apresentados se referem a padrões que se aplicam ao menos um conjunto de softwares, quando não todos. Em contraste a estes resultados, neste trabalho a maior evidência é para os resultados e modelos individuais que cada software gera.

No que se refere à aplicação de modelos para fins de simulação de processos, é interessante citar o trabalho de Kellner *et al.* [KMR1999], talvez o primeiro a justificar a necessidade de simular processos de software. O autor afirma que simulação pode ser utilizado para modelar softwares que apresentam três propriedades: incerteza e estocacidade, comportamento dinâmico e mecanismo de *feedback*. Stopford e Counsell [SC2008] afirmam que o processo de evolução de software possui tais propriedades, podendo, portanto, ser simulado. Em seu trabalho, os autores propõem um *framework* para a síntese de evolução de software baseado em regras. Tais regras derivam especialmente de resultados anteriores que descrevam padrões de mudanças estruturais; ou, caso seja necessário, o usuário pode especificar outros critérios que parametrizam a evolução de um dado software. Em particular, é possível que o modelo estatístico proposto nesta pesquisa possua compatibilidade com o trabalho do Stopford e Counsell [SC2008], apesar de não haver planejamento inicial para tal atividade.

Em relação direta com o objetivo deste trabalho, Pereira *et al.* [PAT2009] justificam apropriadamente a necessidade de se evoluir aspectos de experimentação em engenharia de software através da utilização de modelos, que em outras áreas de conhecimento são denominados *workflows* científicos. Seu trabalho visa auxiliar na concepção de *workflows* voltados para pesquisas em engenharia de software *in vitro* ou *in silico*, nas quais predomina o ambiente simulado. O processo sugerido se assemelha ao seguido na concepção deste trabalho, as diferenças se referem às particularidades que esta pesquisa possui e que não estão cobertos. De fato, a metodologia sugerida apresenta-se em perfeita consonância com o trabalho de Araújo [Ara2009], pois aparentemente derivou deste.

Tal qual se pode observar na revisão apresentada, existem diversas visões sobre a dinâmica da evolução, cada qual contendo uma série de observações interessantes e

complexas por natureza, especialmente aquelas diretamente relacionadas a fatores humanos.

Estes resultados evidenciam a impossibilidade de se conceber um modelo que incorporem todos os resultados experimentais já conhecidos. De fato, modelos já foram gerados e visam ser aplicados em contextos e para fins específicos, tal qual o apresentado neste trabalho.

Capítulo 6

Conclusão

Este trabalho possuiu como motivação prover suporte teórico e ferramental para que seja possível realizar mudanças de granularidade pequena de forma simulada, tal qual percebemos ser a demanda de alguns trabalhos científicos da área de engenharia de software.

Não existe simulação sem um modelo que a suporte, e gerar alguns destes modelos mostrou-se um bom desafio. Este trabalho é, ao que sabemos, o primeiro dedicado a conceber um modelo de evolução quantitativo, estrutural, e de mudanças de granularidade pequena. Aqui, foi aplicado um esforço considerável na tentativa de modelar 8 softwares *open source* através do seu repositório de versões. A complexidade da tarefa impôs vários embates, desde a concepção de ferramentas apropriadas para extração dos dados, a análise e sua simulação. Muitos destes obstáculos deverão ser minimizados a quem aplicar o processo aqui descrito, bem como fizer uso do ferramental aqui disponibilizados. Assim, evidencia-se que a pretensão inicial de prover um suporte teórico e ferramental acerca da temática abordada foi cumprida. Este trabalho avança o estado da arte e provê uma série de resultados que ainda poderão ser aprimorados, ou melhor explorados em outros contextos de trabalhos futuros.

Há, como em qualquer outro trabalho acadêmico, inúmeras limitações neste. Uma, em especial, remete ao conceito de *bootstraping*, e dá conta da escassez de amostras para modelar adequadamente algumas variáveis do domínio dos softwares que exploramos. Propusemos-nos a criar um simulador capaz de gerar evoluções de software que sirvam de amostra para outros fins. Em contrapartida, enfrentamos problemas severos de amostragem quando da concepção de um modelo mínimo para tal

fim. Outros questionamentos se referem a o que consideramos como versão: a atividade de *commit*. Versões maiores ou menores poderiam influenciar diretamente a quantidade de entidade e relacionamentos envolvidos na análise, aprimorando ou inviabilizando a análise. Outra consideração se refere aos softwares escolhidos. Não foi verificada alguma relação de semelhança entre os softwares. É possível que caso os softwares analisados sejam todos de uma mesma "família" de aplicação, os resultados sejam ainda mais restritos do que originalmente se pensou.

Esta breve análise do contexto da pesquisa abre espaço para uma discussão mais apropriada e técnica sobre os resultados obtidos. A primeira conclusão é que, também do ponto de vista estrutural, softwares distintos evoluem de forma distinta. Apesar de este resultado parecer trivial, a dinâmica relacionada às mudanças de granularidade pequena era desconhecida até então, e este trabalho fornece uma luz inicial sobre estes dados, possibilitando o surgimento de um leque de aplicações. Saindo do modelo estatístico, temos a sua aplicação para fins de simulação. O simulador aqui concebido é parametrizável com relação ao número de versões, o software de entrada e o modelo a ser utilizado. Podemos gerar inúmeras amostras distintas e de tamanhos diferentes, baseado em um dos modelos pré-concebidos durante este trabalho. Se aplicarmos o modelo ao software que o gerou, temos capacidade de predição. Se não se trata do software que o gerou, estaremos, ainda assim, simulando evoluções realistas em um nível superior ao que existe atualmente. O maior problema da técnica de simulação aplicada reside no fato de que são introduzidas novas mudanças para comportar as sugeridas pelo simulador. Claramente, estamos enviesando as distribuições de mudanças, e não pudemos estimar quão corrompida a simulação original fica após sua aplicação. Do ponto de vista do objetivo do trabalho, ou seja, a aplicação das sugestões de mudanças para modificação estrutural para fins de pesquisa, é possível considerar o trabalho parcialmente realizado, uma vez que as mudanças aplicadas não condizem com o modelo que a define. Esta, assim como outras limitações do algoritmo de simulação, poderá ser revisitada em momento futuro. No entanto, há uma real percepção de que este problema talvez não possa ser completamente sanado, mas apenas minimizado.

Assim, no que se refere aos trabalhos futuros, um passo apropriado para dar continuidade a esta pesquisa, nos moldes em que foi proposta, é realizar um estudo de caso com algum dos trabalhos de engenharia de software cuja experimentação apontamos como inadequada, e é possível que encontremos resultados equivocados.

Outro viés de aplicação deste trabalho é no contexto organizacional de empresas que visam uma previsão mínima sobre o andamento de um dado software. De fato, este é o mote principal para qualquer pesquisa em evolução de software desde os seus primórdios, e neste caso não poderia ser diferente.

Referências Bibliográficas

[AA2009] S. A. Ajila e S. Alam. Em ICSC 2009: Proceedings of the 3rd IEEE International Conference on Semantic Computing, pp. 390-395, Berkeley, Califórnia, E.U.A. 14-16 de Setembro, 2009.

[Ara2009] M. A. P. Araújo. *Um Modelo para Observação de Evolução de Software.* Tese de Doutorado, COPPE, UFRJ. Rio de Janeiro, Rio de Janeiro, Brasil.

[ADPM2004] G. Antonio, M. Di Penta , E. Merlo. *An Automatic Approach to Identify Class Evolution Discontinuities*. Em IWPSE 2004: Proceedings of the 7th International Workshop on Principles of Software Evolution, pp. 31-40, 06-07 de Setembro, 2004.

[AKM2008] A. Alali, H. Kagdi, J. I. Maletic. *What's a Typical Commit? A Characterization of Open Source Software Repositories*. Em ICPC 2008: Proceedings of The 16th IEEE International Conference on Program Comprehension. pp. 182-191, 2008.

[AB2006] E. Arisholm e L. C. Briand. *Predicting Fault-Prone Components in a Java Legacy System*. Em ISESE 2006: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, pp. 8–17, Rio de Janeiro, Brasil. 2006.

[BL1976] L.A. BELADY, M.M. LEHMAN. A Model of Large Program Development. IBM Systems Journal, vol. 15, no. 1, 1976.

[BKS2003] E. J. Barry, C. F. Kemerer, S. A. Slaughter. *On the uniformity of software evolution patterns*. Em ICSE 2003: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, E.U.A. 03-10 de Maio, 2003.

[CSN2009] A. Clauset, C. R. Shalizi, M. E. J. Newman, *"Power-law distributions in empirical data*". SIAM Review. Volume 51. Issue 4, pp. 661-703, 2009.

[CSE2004] P. Clarkson, C. Simons, C. Eckert. *Predicting Change Propagation in Complex Design*. Journal of Mechanical Design, 126 (5). pp. 788-707, 2004.

[CY1999] M. Cusumano, D. Yoffie. *Software Development on Internet Time*. Em IEEE Computer, Vol. 32, Issue 10, pp. 60 – 69, 1999.

[Dam2011] J. F. Damásio. Dados de análises e resultados desta pesquisa – http://www.gmf.ufcg.edu.br/~jemerson/statsevo/results. Última visita: 21 de Julho, 2011.

[DW2010] Design Wizard – http://www.designwizard.org. Última visita: 05 de Maio, 2010.

[Fow99] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Martin Folwer, 1999.

[Fre2009] Free Mind – http://freemind.sourceforge.net. Última visita: 20 de Novembro, 2008.

[GKM+2000] T. L. Graves, A. F. Karr, J. S. Marron, e H. P. Siy. *Predicting Fault Incidence Using Software Change History*. IEEE Transactions on Software Engineering, Vol. 26, Issue 7: pp. 653– 661, 2000.

[GLD2005] T. Girba, M. Lanza, S. Ducasse. *Characterizing the Evolution of Class Hierarchies*. Em CSMR 2005: Proceedings of 9th European Conference on Software Maintenance and Reengineering), IEEE Computer Society, pp.2-11, 2005.

[GS2009] M. M. Geipel e F. Schweitzer. *Software change dynamics: evidence from 35 java projects*. Em ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. Amsterdam, Holanda. 24-28 Agosto, 2009.

[GT2002] M. Godfrey e Q. Tu, *Tracking Structural Evolution Using Origin Analysis*. Em IWPSE 2002: Proceedings of the International Workshop on Principles of Software Evolution, Orlando, Florida, E.U.A. 19-20 de Maio, 2002,

[GT2005] M. W. Godfrey e L. Zou. *Using Origin Analysis to Detect Merging and Splitting of Source Code Entities*. IEEE Transactions on Software Engineering, Vol. 31, Issue 2: pp. 166-181, Fevereiro, 2005.

[HL2008] L. Hattori e M. Lanza. *On the Nature of Commits*. Em ASE 2008: Proceeding of 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, p.63-71, 15-16 de Setembro, 2008.

[HWS2000] R. C. Holt, A. Winter, A. Schürr. *GXL: Towards a Standard Exchange Format.* Em WCRE 2000: Proceedings 7th Working Conference on Reverse Engineering, 2000.

[Jav2009] Javex Extractor – http://www.swag.uwaterloo.ca/javex/index.html. Última visita: 21 de Novembro, 2008.

[JUn2009] JUnit Framework – http://junit.sourceforge.net . Última visita: 21 de Novembro, 2008.

[KL2006] A. A. P. Kazem e S. Lofti. *A Modified Genetic Algorithm for Software Clustering Problem*. Em ICQIC 2006: Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications, pp. 306-311, Elounda, Greece, 18-20 de Agosto, 2006

[KS1997] C.F. Kemerer, S. Slaughter. *Determinants of Software Maintenance Profiles: An Empirical Investigation*. Em Journal of Software Maintanance, Vol. 9, pp 235-251, 1997.

[KMR1999] M. I. Kellner, R. J. Madachy, R. Raffo: *Software process simulation modeling: Why? What? How?* Journal of Systems and Software. Vol 46 Issue 2-3: pp. 91-105, 1999.

[LE2007] O. Le Goaer e P. Ebraert. *Evolution styles: change patterns for Software Evolution*. Em Proceedings Third International ERCIM Workshop on Software Evolution, 2007.

[Leh1969] M. M. Lehman. *The Programming Process*, IBM Res. Rep. RC 2722, IBM Res. Centre, Yorktown Heights, Nova Yorque, E.U.A.,Setembro, 1969.

[Leh1985] M. M. Lehman e L. A. Belady. *Program Evolution - Processes of Software Change*. Academic Press, Londres, pp. 538, 1985.

[Leh1991] M. M. Lehman, *Software Engineering, the Software Process and Their Support*, IEEE Software Engineering Journal Special Issue on Software Environments and Factories, Vol. 6, Issue 5, pp. 243-258. Setembro, 1991.

[Leh1996] M. M. Lehman, *Laws of Software Evolution Revisited*. Em EWSPT 1996: Proceedings of the 5th European Workshop on Software Process Technology, pp. 108-124. Nanci, França. Outubro, 1996.

[Leh1998] M.M. Lehman, J. F. Ramil, D. E. Perry. *On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution*. Em METRICS 1998: Proceedings of the 5th International Symposium on Software Metrics. 1998.

[Mat2010] Matlab – http://www.mathworks.com/products/matlab/. Última visita em 20 de Outubro, 2010.

[MHPB2009] E. Murphy-Hill, C. Parnin, A. P. Black. *How We Refactor, and How We Know It*. Em ICSE 2009: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, pp. 287-297, Outubro, 2009.

[Opd1992] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Tese de P.h.D. University of Illinois at Urbana Champaign, 1992.

[PAT2009] W. M. Pereira, M. A. P. Araújo, G. H. Travassos. *Apoio na Concepção de Workflow Científico Abstrato para Estudos in vitro e in silico em Engenharia de Software*. Em ESELAW 2009: Proceedings of the VI Experimental Software Engineering Latin American Workshop, pp. 22-31, 11-13 de Novembro, 2009.

[PR2009] M. Petrenko, Václav Rajlich. *Variable Granularity for Improving Precision of Impact Analysis*. Em ICPC 2009: Proceedings of the IEEE 17th International Conference on Program Comprehension, pp. 10-19, 17-19 de Maio, 2009.

[PSE2004] J.W. Paulson, G. Succi, A. Eberlein. *An empirical study of open-source and closed-source software products*. IEEE Transactions of Software Engineering, Vol. 30, Issue 4, pp. 246–256, 2004.

[PSV2001] D.E. Perry, H.P. Siy, L.G. Votta. *Parallel Changes in Large-Scale Software Development: An Observational Case Study*. Em ICSE 2001: Proceedings of the International Conference on Software Engineering, Volume 19, Issue 25, pp 251 – 260. 2001.

[R2010] *The R Project* – http://www.r-project.org. Última visita: 20 de Novembro, 2010.

[RL2006] R. Robbes e M. Lanza. *Change-based Software Evolution*. Em EVOL 2006: Proceedings of 1st International Workshop in Mining Software Repositories), IEEE CS Press, pp. 159-164, 2007.

[RT2001] B. G. Ryder e F. Tip. *Change impact analysis for object-oriented programs*. Em PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 46-53. Snowbird, Utah, E.U.A, 2001.

[RVP2006] A. Raza, G. Vogel, E. Plödereder. *Bauhaus - A Tool Suite for Program Analysis and Reverse Engineering*. In Reliable Software Technologies, Ada Europe, pp. 71-82, 2006

[SC2008] B. Stopford e S. Counsell. *A Framework for the Simulation of Structural Software Evolution*. Em ACM Transactions on Modeling and Computer Simulation. Vol. 18, Issue 4, pp. 1-36, 2008.

[Sou2010] R. Souza. *Dissertação de Mestrado: Modelos Computacionais Realistas para Dependências entre Entidades de Software*. DSC/UFCG, Brasil. 2010.

[TH2000] V. Tzerpos e R. C. Holt. *On the Stability of Software Clustering Algorithms*. Em IWPC 2000: Proceedings of the 8th International Workshop on Program Comprehension, pp. 211-218, 2000.

[TT1992] T. Tamai, Y. Torimitsu. *Software Lifetime and its Evolution Process over Generations*" Em Proceedings of Conference in Software Maintenance, IEEE, Volume 9, Issue 12, pp. 63– 69, 1992.

[TSM+2008] G. H. Travassos, P. S. M. dos Santos, P. G. Mian, A. C. D. Neto, J. Biolchini. *An Environment to Support Large Scale Experimentation in Software Engineering*. Em ICECCS 2008: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems, pp. 193-202, Washington, DC, E.U.A., 2008.

[WWY+2009] L. Wang, Z. Wang, C. Yang, L. Zhang. *Linux Kernels as Complex Networks: A Novel Method to Study Evolution.* Em ICSM 2009: Proceeding of International Conference in Software Maintenance, pp. 41-50, 2009.

[XCN2009] G. Xie, J. Chen, I. Neamtiu. *Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software*. Em ICSE 2009: Proceedings of International Conference on Software Engineering, pp.51-60, 2009.

[ZHK+2006] H. Zhang, M. Huo, B. Kitchenham, R. Jeffery. *Qualitative Simulation Model for Software Engineering Process*. Em ASWEC 2006: Proceedings of the 17th Australian Software Engineering Conference, pp. 391-400. Sydney, Austrália. 18-21 de Abril, 2006.

[ZKK+2008] H. Zhang , J. Keung , B. Kitchenham , R. Jeffery. *Semi-quantitative Modeling for Managing Software Development Processes*, Em ASWEC 2008: Proceedings of the 19th Australian Conference on Software Engineering, pp. 66-75, Perth, Western Austrália, Austrália, 25-28 de Março, 2008

[ZW2004] T. Zimmermann e P. Weißgerber. *Preprocessing CVS Data for Fine-Grained Analysis*. Em MSR 2004: Proceeding of the International Workshop on Mining Software Repositories, pp. 2-6, Edinburgh, Scotland, 2004.