

**EYESON: UM ARCABOUÇO PARA EXTRAÇÃO,
ARMAZENAMENTO E ACOMPANHAMENTO DE
MÉTRICAS DE PROJETO DE CIRCUITOS
INTEGRADOS**

THIAGO SOUSA FIGUEIREDO SILVA

**EYESON: UM ARCABOUÇO PARA EXTRAÇÃO,
ARMAZENAMENTO E ACOMPANHAMENTO DE
MÉTRICAS DE PROJETO DE CIRCUITOS
INTEGRADOS**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: ANTÔNIO OTÁVIO FERNANDES
CO-ORIENTADOR: JOSÉ AUGUSTO M. NACIF

Belo Horizonte

Março de 2011

© 2011, Thiago Sousa Figueiredo Silva.
Todos os direitos reservados.

Silva, Thiago Sousa Figueiredo
S586e EyesOn: um arcabouço para extração,
armazenamento e acompanhamento de métricas de
projeto de circuitos integrados / Thiago Sousa
Figueiredo Silva. — Belo Horizonte, 2011
xvi, 59 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de
Minas Gerais
Orientador: Antônio Otávio Fernandes
Co-Orientador: José Augusto M. Nacif

1. Computação - Teses. 2. Circuitos digitais - Teses.
I. Orientador II. Co-Orientador III. Título

CDU 519.6*17(043)




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

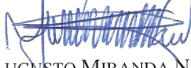
FOLHA DE APROVAÇÃO

EyesOn: um arcabouço para extração armazenamento e acompanhamento de métricas de projeto de circuitos integrados


THIAGO SOUSA FIGUEIREDO SILVA


Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. ANTÔNIO OTÁVIO FERNANDES - Orientador
Departamento de Ciência da Computação - UFMG


PROF. JOSÉ AUGUSTO MIRANDA NACIF - Co-orientador
Departamento de Informática - UFV


PROF. ROMANELLI LODRON ZUIM
Departamento de Ciência da Computação-PUC


PROF. LUIZ FILIPE MENEZES VIEIRA
Departamento de Ciência da Computação - UFMG


PROF. OMAR PARANAÍBA VILELA NETO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 25 de março de 2011.

Resumo

Este trabalho propõe o uso de dados armazenados em ferramentas de gerência de configuração para rastrear a evolução dos circuitos integrados. Esse rastreamento é feito pela ferramenta EyesOn. Esta ferramenta é um arcabouço de código aberto e projetada para ser extensível, permitindo a adição de novas métricas e integração com outras ferramentas de gerência de configuração. A arquitetura da ferramenta é composta por um núcleo básico de classes que representam as informações manipuladas e que é capaz de armazenar diferentes tipos de métricas e sistemas de controle de versão. Também são apresentados um conjunto de métricas de produto e de processo das etapas de implementação, testes e síntese.

Uma contextualização do problema é apresentada por meio de um estudo de caso. Para que fosse possível reconstruir um ambiente de desenvolvimento foi proposta uma infraestrutura na qual os desenvolvedores fornecem dados do desenvolvimento de um processador elaborado para o propósito acadêmico. Após a coleta dos dados são apresentados gráficos e um mecanismo para indicar propensão a erros, explorando o conceito de localidade temporal.

As informações obtidas no histórico de desenvolvimento já provaram ser úteis para a melhoria dos processos e dos produtos de software. Da mesma forma, essas informações podem também ser aplicadas em projetos de hardware. Por meio das informações que são extraídas a cada etapa de um processo de desenvolvimento de circuito integrado, e a cada revisão, esses dados podem contribuir para redução de erros e também para melhoria da qualidade desses processos.

Palavras-chave: Projeto de Circuitos Integrados Digitais, Evolução de Hardware, Métricas, e Propensão a Erros.

Abstract

We propose the usage of information obtained from configuration management tools stored data to track integrated circuit design evolution. The tracking is performed by the tool EyesOn. It is an open source framework designed to be extensible and to have easy integration with configuration management tools. The framework kernel architecture is composed by classes that represent handled entities and also store metrics and history information. We also present a set of product and process metrics gathered from design implementation, test and synthesis.

In order to contextualize the problem a case study is presented. We prepared a development environment where university students developed a processor for academic purposes and sent development data to configuration management tools. After data extraction, some charts and an error proneness indication mechanism, based on temporal locality, are presented.

Development history information has been already used to improve software products and their development processes. We propose that in same direction this kind of information can be also applied to hardware. The information gathered from each design process step can be used to reduce bugs before fabrication and also to improve design process quality.

Keywords: Digital Integrated Circuit Design, Hardware Evolution, Metrics, Error Proneness.

Lista de Figuras

2.1	Fluxo em alto nível de um processo de desenvolvimento.	5
2.2	Etapas do desenvolvimento de um circuito integrado que antecedem a fabricação [Mullane & MacNamee, 2008].	6
2.3	Etapas do desenho e implementação de um circuito integrado [Vaumorin & Romanteau, 2011].	7
2.4	Fluxo de funcionamento do Bugzilla [Bugzilla, 2011].	9
2.5	Evolução da estrutura de arquivos e diretórios [Collins-Sussman et al., 2007].	11
2.6	Acessos ao repositório para leitura e escrita.	11
3.1	Contexto de utilização de métricas apresentado na proposta do capítulo “Software Measurement” [Abran et al., 2008] do SWEBOK.	24
3.2	Recorte do gráfico de radar apresentado em Pinzger et al. [2005].	25
3.3	Mapa de posse apresentado em Girba et al. [2005].	25
3.4	Gráfico apresentado em Fischer et al. [2003].	25
3.5	Locais de erros no código-fonte do Firefox [Voinea & Telea, 2007].	26
4.1	Fluxos de dados na ferramenta.	29
4.2	Contextos de utilização	30
4.3	Arquitetura do núcleo da ferramenta.	31
4.4	Ponto de extensão para novas métricas.	33
4.5	Exemplo de especialização da classe que representa uma métrica.	34
5.1	Exemplo de módulo especificado.	38
5.2	Estrutura de diretórios do repositório para cada grupo.	40
5.3	Módulo de teste e módulo sob teste.	41
5.4	Algoritmo de alto nível para escolha dos módulos mais frequentemente modificados.	43
5.5	Revisões de correção. Estratégias MFC, FIFO e probabilidade, cache de tamanho 3.	44

5.6	Revisões de correção. Estratégias MFC, FIFO e probabilidade, cache de tamanho 5.	45
5.7	Revisões de modificação. Estratégias MFM, FIFO e probabilidade, cache de tamanho 3.	45
5.8	Revisões de modificação. Estratégias MFM, FIFO e probabilidade, cache de tamanho 5.	46
5.9	Mapa de calor.	47
5.10	Erros encontrados por quinzena.	48
5.11	Métricas de hardware contrapostas com o histórico de implementação. . . .	48
5.12	Visualização de múltiplas métricas com gráfico de radar.	50

Lista de Tabelas

2.1	Leis que regem a evolução de um software	13
4.1	Métricas utilizadas neste arcabouço.	35
4.2	Classificação das métricas utilizadas neste arcabouço.	36
5.1	Descrição dos sinais da ALU.	39
5.2	Condições de overflow para adição e subtração.	39
5.3	Lista de módulos aproveitados por grupo.	41
5.4	Tabela de execução do algoritmo MFM.	43

Sumário

Resumo	vii
Abstract	ix
Lista de Figuras	xi
Lista de Tabelas	xiii
1 Introdução	1
1.1 Objetivo	2
1.2 Motivação	2
1.3 Organização	3
2 Conceitos	5
2.1 Ciclo de vida e processos de desenvolvimento de HW/SW	5
2.2 Ferramentas de gerência de configuração	8
2.2.1 Sistemas de rastreamento de erros	8
2.2.2 Sistemas de controle de versão	10
2.3 Evolução de software	11
3 Contextualização	15
3.1 Métricas	16
3.1.1 Métricas de software	16
3.1.2 Métricas de software aplicáveis a circuitos integrados	17
3.1.3 Métricas de circuitos integrados	18
3.2 Ferramentas	21
3.3 Visualização	22
4 EyesOn: Ferramenta de extração	27
4.1 Como funciona a extração dos dados	28

4.2	Cenários de utilização	30
4.3	Arquitetura	31
4.3.1	Novas métricas	33
4.3.2	Modelo de banco de dados	33
4.3.3	Compatibilidade com repositórios de SCVs	34
4.4	Métricas implementadas	35
5	Ambiente de experimentação e resultados	37
5.1	Implementação de um circuito para experimentação	37
5.1.1	Infraestrutura adotada e instruções de uso do software de controle de versão	38
5.1.2	Testes e documentação dos erros e modificações	40
5.2	Dados obtidos do ambiente de experimentação	41
5.3	Utilização de métricas do processo como base para um modelo de propensão	42
5.4	Visualização da evolução do circuito	47
5.4.1	Erros no tempo	47
5.4.2	Métricas de hardware contrapostas com o histórico de implementação	47
6	Conclusões e trabalhos futuros	51
6.1	Conclusões	51
6.2	Trabalhos futuros	52
	Referências Bibliográficas	53
	Apêndice A Publicações durante mestrado	59

Capítulo 1

Introdução

Pode-se dizer que um dos maiores problemas enfrentados por fabricantes de bens de consumo é o tempo de mercado. Quanto maior é o tempo de desenvolvimento de um produto, maior é o tempo dado a concorrência, menor é o tempo de comercialização, assim como menor é o retorno financeiro. Quanto mais rápido um produto fica pronto maior é o lucro proporcionado. Na área de hardware este conceito também se aplica. Contudo, a complexidade e o tamanho dos circuitos integrados (CIs) continuam crescendo conforme a lei de Moore [Moore, 1965], demandando um esforço maior de desenvolvimento. Para que o tempo de mercado seja mantido, faz-se necessário que os processos de desenvolvimento sejam cada vez mais eficientes. Infelizmente a quantidade de tempo e recursos são escassos. Para se manter ativa no mercado e obter bons lucros, a indústria necessita constantemente de novos métodos e ferramentas que possam contribuir para o aumento da eficiência de seus processos de desenvolvimento.

Quando o assunto é esforço de desenvolvimento e tempo de mercado, o fator que mais impacta são os erros. Estima-se que o esforço necessário em testes e verificação é igual ou maior do que o de implementação ([Bentley et al., 2004] e [Foster et al., 2004]). A motivação pela busca dos erros é o fato de que quanto maior é o tempo gasto para se detectar um erro, maior é o custo que ele representa. Um erro identificado por inspeção durante a implementação pode ser rapidamente sanado. No entanto, um erro de especificação descoberto após o desenho e a implementação pode exigir que estas duas etapas sejam refeitas. Um lote de *chips* com erros representam um altíssimo custo pois a fabricação é feita em grande quantidade.

Os erros estão inseridos em todas as etapas de um processo de desenvolvimento. Para que a melhoria destes processos seja efetiva, de maneira que os erros sejam identificados mais rápido, é necessário conhecer melhor as especificidades de cada uma destas etapas. Para isto faz-se necessário a utilização de mecanismos que possibilitem

uma avaliação destas, quantitativamente. Na indústria de software, a área de pesquisa que tem as etapas de um processo de desenvolvimento como seu objeto de estudo é conhecida como evolução de software. Este termo ainda é desconhecido para circuitos integrados digitais, no entanto, existem linhas de pesquisas já maduras na área de software que investigam a evolução de componentes e utilizam estes dados para finalidades como predição de erros, medição de complexidade de componentes, medição do esforço de desenvolvimento e outras finalidades gerenciais.

Apesar das similaridades com a área de software, este trabalho é direcionado ao desenvolvimento de circuitos integrados digitais. As pesquisas de evolução de software servem de inspiração e demonstram um potencial para investigação na área de hardware. Este trabalho propõe um arcabouço para extração, armazenamento e rastreamento de métricas da implementação de circuitos integrados digitais feitas em linguagens de descrição de hardware ou HDL (*Hardware Description Language*). Estas métricas são de fundamental importância para que se possa investigar a evolução de componentes de circuitos integrados no tempo.

1.1 Objetivo

Sabe-se que a implementação é a base para o ciclo: verificação, testes e correções, que antecipam a fabricação de um *chip*. A redução dos erros nesta primeira etapa implica em redução de tempo, esforço e custo das seguintes. Indicadores de maior ou menor propensão ao erro podem ser utilizadas para guiar os esforços de verificação e testes, aumentando eficiência destas etapas. O objetivo principal deste arcabouço é contribuir para a exploração de métricas de evolução da implementação de circuitos integrados e sua correlação com propensão a erros. Estas métricas são obtidas da descrição do circuito, do histórico de revisões e dos relatórios de erros que são armazenados durante a implementação. Após um pré-processamento, estas informações obtidas do histórico de desenvolvimento são reconstituídas numa linha do tempo. A análise estatística destes dados pode ser usada para diversos outros fins além de indicar propensão a erros, como por exemplo, mostrar o esforço de desenvolvimento e complexidade.

1.2 Motivação

A motivação deste trabalho é a necessidade de uma ferramenta que forneça mecanismos de extração automatizado e armazenamento de dados estatísticos da etapa de implementação de circuitos integrados. Estas informações poderão contribuir com a melhoria

dos processos de desenvolvimento, permitindo o acompanhamento mais detalhado da evolução dos circuitos. Esta ferramenta viabilizou as primeiras pesquisas de evolução em circuitos integrados, e poderá contribuir para futuras.

1.3 Organização

No Capítulo 2 são apresentados alguns conceitos básicos sobre processos de desenvolvimento, ciclo de vida, ferramentas de gerência de configuração e evolução de software, que facilitarão o entendimento do texto. A revisão bibliográfica está no Capítulo 3. No Capítulo 4 é apresentado o EyesOn, e também como são feitas as extrações de dados, quais são seus cenários de utilização, e como sua arquitetura foi desenhada. O Capítulo 5 apresenta o ambiente de experimentação utilizado para avaliar a ferramenta, e os resultados obtidos. Finalmente no Capítulo 6 são apresentadas as conclusões e propostos trabalhos futuros. No Apêndice A são apresentadas as publicações realizadas durante o mestrado, e que são parte dos resultados obtidos por meio do EyesOn.

Capítulo 2

Conceitos

2.1 Ciclo de vida e processos de desenvolvimento de HW/SW

Entende-se por processo uma sequência de ações sistemáticas que levam a algum resultado. Os **processos de desenvolvimento** de software e hardware são organizados e constituídos por atividades, métodos, práticas e transformações para que no final se tenha um produto que atenda as especificações desejadas. Existe também o termo **ciclo de vida** de um produto. Diferentemente de estudar o processo de desenvolvimento é estudar o ciclo de vida, no qual um ou mais processos podem estar inseridos. O ciclo de vida como o próprio termo indica, contempla não apenas um objetivo como é o caso do processo, mas todas as etapas do desenvolvimento de um produto, considerando no entanto suas possíveis reutilizações.

Em geral os processos de desenvolvimento seguem um fluxo como o apresentado na Figura 2.1. Nesta são ilustradas em alto nível as etapas percorridas pelos desenvolvedores pelo menos uma vez durante o ciclo de vida de um produto.

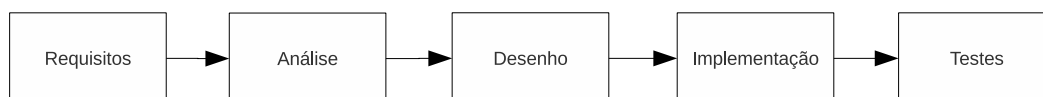


Figura 2.1. Fluxo em alto nível de um processo de desenvolvimento.

Na primeira etapa são levantadas as funcionalidades do sistema deverá conter, formando uma lista de requisitos. Em seguida estes requisitos são analisados e uma especificação mais detalhada é produzida. Esta especificação contém as informações necessárias para que os desenvolvedores das etapas seguintes possam desenhar, imple-

mentar e testar o produto. Na etapa de desenho são elaboradas as estruturas principais do sistema, e na etapa seguinte estas são implementadas. A etapa de testes permite identificar erros na implementação.

Os processos de desenvolvimento de hardware e software são bem distintos quando comparados por completo, mas pode-se afirmar que ambos compartilham as etapas básicas ilustradas na Figura 2.1. As etapas de um processo real dependem das características do produto que será desenvolvido e também da metodologia utilizada pela organização desenvolvedora, que nem sempre atua em todo ciclo de vida do produto. O processo de desenvolvimento de um circuito integrado possui mais etapas em seu fluxo, como ilustrado pela Figura 2.2, podendo incluir também as etapas de fabricação.

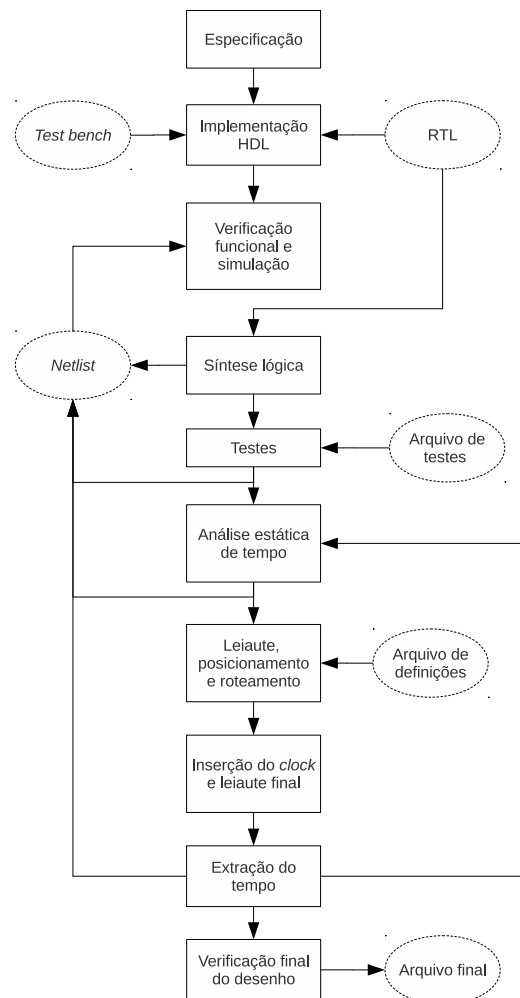


Figura 2.2. Etapas do desenvolvimento de um circuito integrado que antecede a fabricação [Mullane & MacNamee, 2008].

Este fluxo apresenta as etapas que antecede a fabricação, começando na elaboração da especificação e terminando na verificação final do desenho. Observa-se que

o formato e os nomes das etapas são diferentes quando comparados com o fluxo da Figura 2.1. Por meio de uma análise mais cuidadosa é possível entender que especificação equivale a requisitos e análise, e implementação HDL equivale a desenho e implementação. Verificação funcional e simulação são etapas características do processo de desenvolvimento de um ASIC¹.

A implementação do código HDL e as etapas seguintes são feitas com base na especificação. Esta etapa ainda pode ser subdividida, por exemplo, em desenho e implementação do modelo comportamental, dos módulos de teste (*test benches*), e do código RTL (*Register Transfer Level*). É uma prática comum adotada na indústria fazer uma primeira implementação que apenas simule o funcionamento do circuito. Isto permite que suas estruturas principais sejam definidas em um nível maior de abstração e requeiram um esforço menor de desenho. Na Figura 2.3 esta primeira implementação seria o modelo comportamental, que normalmente é feita com o uso de uma linguagem de alto nível. O próximo passo é a tradução deste modelo comportamental para um código em linguagem de descrição de hardware em nível sintetizável, ou seja, um código RTL que será transcrito em portas lógicas. Esta tradução é feita por um ciclo de refinamentos sucessivos. A verificação dos modelos garante que menos erros sejam inseridos na transição de um modelo para outro. A prototipação é o passo que garante o fim da etapa de implementação.

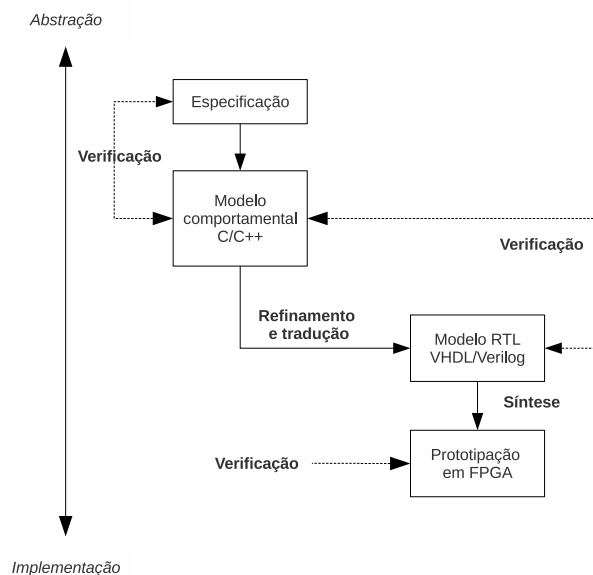


Figura 2.3. Etapas do design e implementação de um circuito integrado [Vau-morin & Romanteau, 2011].

O foco deste trabalho é estudar os dados do desenvolvimento de um circuito inte-

¹Sigla para circuito integrado de aplicação específica (*Application-Specific Integrated Circuit*).

grado, gerados a partir das iterações que ocorrem no fluxo de desenho, implementação, verificação e testes, até que código HDL esteja pronto. As etapas: análise estática de tempo; leiaute, posicionamento e roteamento; inserção do *clock* e leiaute final; extração do tempo; e verificação final do desenho também fornecem informações que pode ser estudadas. No entanto, estas etapas são da montagem final do *chip*, e normalmente partem de um desenho já pronto. O desenho de um circuito integrado completo ou de um módulo pronto, após a etapa de implementação é um produto de propriedade intelectual, também referenciado como núcleo IP (*Intellectual Property*) ou bloco IP. No fluxo ilustrado pela Figura 2.2, as etapas que contém as informações objeto de estudo deste trabalho estão concentradas do início da implementação até testes.

2.2 Ferramentas de gerência de configuração

Entende-se por configuração o conjunto de artefatos físicos e/ou funcionais como hardware, software e documentos produzidos e/ou utilizados no processo de desenvolvimento. Gerência de configuração é uma subárea da engenharia de software que tem como objetivo oferecer suporte a estes processos. O suporte começa pela definição de métodos e configurações que irão compor o ambiente de trabalho dos desenvolvedores, e envolve também atividades de gerência, planejamento, auditoria e controle. Um referencial teórico mais abrangente sobre gerência de configuração está no livro SWEBOK [Abran et al., 2004].

Como suporte à gerência de configuração existem as ferramentas de gerência de configuração (FGC). Os dois tipos de FGCs abordados neste texto são: sistemas de rastreamento de erros² (SRE) e sistemas de controle de versão³ (SVC).

2.2.1 Sistemas de rastreamento de erros

Durante o desenvolvimento de circuitos integrados pode-se dizer que os erros são inevitavelmente inseridos em qualquer uma das etapas. Os sistemas de rastreamento de erros são compostos por uma base de dados e uma interface de usuário, na qual o desenvolvedor relata todas as ocorrências de erros no decorrer do desenvolvimento. Por meio destas ferramentas, os erros são registrados na medida em que vão sendo identificados, e consultados na medida em que puderem ser resolvidos ou concluídos. O uso destes sistemas facilita as atividades de correção, principalmente quando as equipes são

²Tradução literal de *Bug Tracking Systems*.

³Tradução literal de *Versioning Control Systems*.

compostas por vários desenvolvedores. Na Figura 2.4 é ilustrado o fluxo de registro de erros na ferramenta Bugzilla⁴ que é popularmente conhecida e utilizada na indústria.

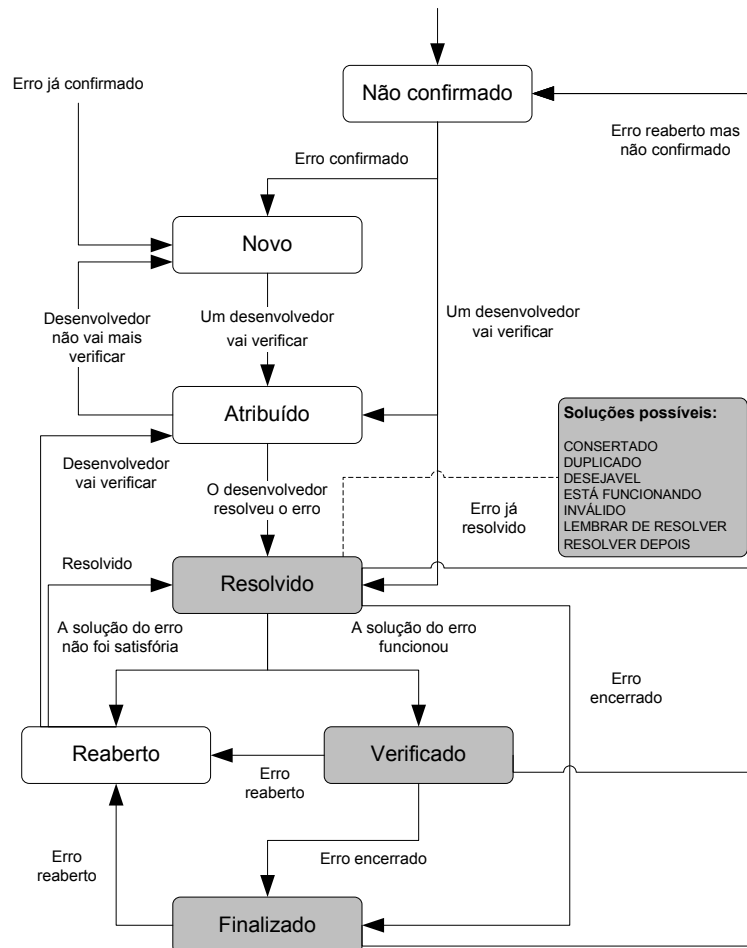


Figura 2.4. Fluxo de funcionamento do Bugzilla [Bugzilla, 2011].

Nestas ferramentas os erros podem receber classificações diversas, e por isso pode-se dizer também que são as modificações que são documentadas. Dentre as classificações existentes, um erro pode ser classificado como por exemplo uma modificação de melhoria, ou um problema minoritário, maioritário ou severo. O registro destas ocorrências é uma boa prática, pois além de ajudar nas tarefas de desenvolvimento, possibilita no final a obtenção de informações sobre o que aconteceu durante o processo. Os dados poderão ser coletados e utilizados pelos gerentes como fonte de informações estatísticas.

Alguns softwares populares para rastreamento de erros são: Bugzilla, Mantis⁵ e

⁴Bugzilla é uma das ferramentas de rastreamento de erros mais utilizada pelos desenvolvedores de software de código aberto. Mais informações em: <http://www.bugzilla.org/>.

⁵Mantis é um software de rastreamento de erros, gratuito e de código aberto, disponível no sítio: <http://www.mantisbt.org/>.

JIRA⁶. Os dois primeiros são gratuitos e estão disponíveis na Internet para livre utilização. Contudo é comum que estes softwares citados nem sempre atendam a algumas organizações. O Bugzilla e o Mantis são soluções fechadas que oferecem um certo nível de personalização, mas nem sempre o esforço de modificar uma ferramenta pronta é menor do que construir ou adquirir outra. Para preenchimento dos relatórios de erros no ambiente de experimentação apresentado no Capítulo 5, foi utilizada uma linguagem de descrição de erros de circuitos integrados, BugLanguage, apresentada em Cardoso et al. [2009] como alternativa mais simples e específica para esta finalidade.

2.2.2 Sistemas de controle de versão

Os sistemas de controle de versão são repositórios de arquivos, utilizados pelos desenvolvedores para facilitar sincronização entre sua equipe e documentar modificações que são feitas em arquivos do projeto como documentação, código-fonte etc. Utilizar um sistema como este é mais eficiente e organizado do que manter diversas cópias de arquivos ou tentar sincronizá-los com uma equipe por meio de troca de mensagens. Muitas organizações desenvolvedoras de software e também de circuitos integrados utilizam este tipo de ferramenta.

Quando se utiliza um SCV todas as modificações feitas no repositório são registradas. Para cada modificação enviada é gerado um número de revisão ou versão. Este número representa o estado da árvore de diretórios naquele momento. Posteriormente os desenvolvedores podem recuperar os arquivos e diretórios de cada uma das revisões. Estas revisões representam um histórico no qual é possível saber quem realizou modificações, quando foram enviadas, e quais foram. A Figura 2.5 ilustra uma estrutura de arquivos e diretórios que são modificados e registrados por uma ferramenta de controle de versão. Esta apresenta mudanças feitas em três revisões (1 a 3).

O acesso a estas ferramentas são realizados como no modelo cliente-servidor, ilustrado na Figura 2.6, onde o programa de controle de versão é o servidor acessado remotamente pelos desenvolvedores para acessos de leitura ou escrita. Cada desenvolvedor instala em seu computador um programa que sincroniza sua cópia local dos arquivos com uma determinada versão no repositório. A sincronização entre a cópia local e o repositório é feita por meio do envio das modificações (escrita) ou leitura.

⁶JIRA é uma ferramenta comercial para rastreamento de detalhes do desenvolvimento de projetos de software, e dentre estes, erros também. Mais informações podem ser encontradas no sítio: <http://www.atlassian.com/software/jira/>.

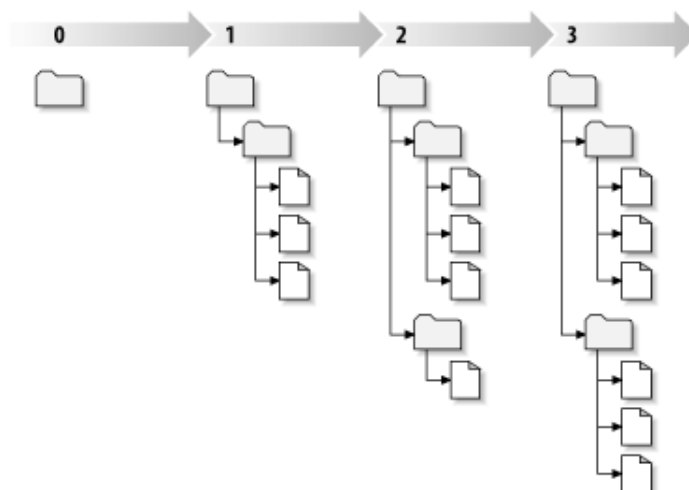


Figura 2.5. Evolução da estrutura de arquivos e diretórios [Collins-Sussman et al., 2007].

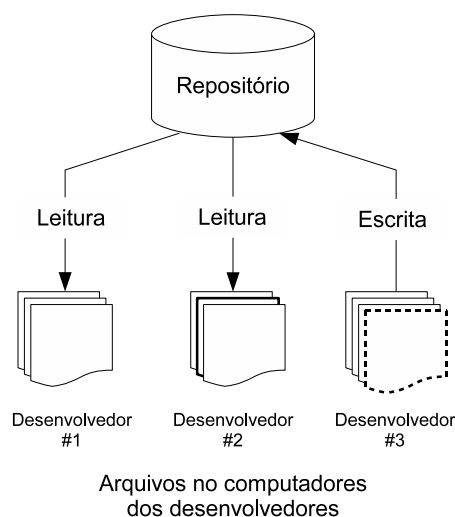


Figura 2.6. Acessos ao repositório para leitura e escrita.

2.3 Evolução de software

A evolução de software é uma área de pesquisa já consolidada da engenharia de software. Nesta são estudados toda a dinâmica do ciclo de vida de um produto. O ciclo completo inclui a implementação inicial e as modificações realizadas na manutenção de um produto até que o mesmo não seja mais utilizado. Os trabalhos encontrados nesta área têm como foco uma parte do ciclo de vida ou o ciclo com um todo. O estudo da evolução de um produto pode estar limitado a apenas conhecer como ele de fato acontece em determinadas circunstâncias, ou seu objetivo pode ser simplesmente a obtenção de dados para alguma finalidade gerencial ou estatística.

Na área de software o estudo do ciclo de vida de um produto continua após a primeira implementação, na medida em que vão sendo realizadas manutenções. Na área de hardware acontece a mesma coisa. Um *chip* após fabricado não tem mais como ser modificado e já teve seu ciclo de vida encerrado. No entanto, a parte de propriedade intelectual que é o desenho e a implementação do circuito pode ser dividida em um ou mais núcleos que continuam sendo comercializados e aprimorados. O ciclo de vida do produto de propriedade intelectual seja ele de software ou hardware acaba apenas quando é mais interessante jogá-lo fora e projetar outro. Existem várias semelhanças entre o ciclo de software e o ciclo da parte de propriedade intelectual dos CIs. A Tabela 2.1 apresenta as oito leis que regem este domínio na área de software e que também pode ser investigada para hardware.

Tabela 2.1. Leis que regem a evolução de um software

I	Lei da modificação contínua: no período em que um software é utilizado, modificações são necessárias para que o mesmo seja adaptado às mudanças e aos novos requisitos. Portanto sua manutenção torna-se contínua até que num determinado momento seja mais barato substituí-lo [Belady & Lehman, 1976].
II	Lei do aumento da complexidade: com a constante mudança, na medida em que um sistema é modificado, sua arquitetura tende a se degenerar. A complexidade deste sistema aumenta e o mesmo torna-se mais difícil de ser mantido. Isto ocorre a menos que seja feito um esforço extra para mantê-lo e reduzir sua complexidade [Belady & Lehman, 1976].
III	Lei fundamental da evolução de um software: as métricas obtidas de atributos de um software quando coletadas localmente ou num espaço curto de tempo são aparentemente estocásticas. No entanto, quando observadas num período longo e determinado, estas métricas representam estatisticamente tendências e invariâncias, podendo se dizer que elas se correlacionam [Belady & Lehman, 1976].
IV	Lei da estabilidade organizacional: durante o ciclo de vida de um software, a quantidade de trabalho de desenvolvimento é estatisticamente invariante [Lehman, 1980].
V	Lei da conservação da familiaridade: durante o ciclo de vida ativo de um software, a quantidade de conteúdo modificado (adicionado, removido, alterado) nas versões que se sucedem é estatisticamente invariante [Lehman, 1980].
VI	Lei do crescimento contínuo: constantemente novas funcionalidades são incorporadas ao sistema para satisfazer os novos requisitos do usuário, aumentando o tamanho do software. [Lehman, 1980].
VII	Lei do declínio da qualidade: a qualidade do software se mostrará em queda a menos que seja investido em sua manutenção e adaptação às mudanças que ocorrem no seu ambiente operacional [Lehman et al., 1997].
VIII	Lei da contra resposta: os softwares evoluem devido sua interação com os usuários em vários níveis e etapas. Pode-se entender que este processo de evolução é um refinamento da implementação inicial [Lehman et al., 1997].

Capítulo 3

Contextualização

A utilização de métricas tem fundamental importância para o desenvolvimento de software e hardware, sendo um instrumento básico de gestão para atividades nas quais há planejamento e acompanhamento. A partir da década de 1970, no auge da então identificada crise do software, o uso de métricas para quantificar, qualificar e permitir a avaliação de processos de desenvolvimento de software já era objeto de estudo. Para avaliar projetos com dezenas de milhares de linhas de código, fez-se necessário quantificar aspectos qualitativos. Os primeiros modelos sobre quais seriam e como se classificariam os aspectos qualitativos foram publicados por Boehm et al. [1976] e Cavano & McCall [1978].

Desde a crise do software sempre houve uma busca por novos modelos qualitativos, padrões de classificação e também a investigação de métricas que levassem em consideração características específicas das tecnologias que foram surgindo ao longo do tempo, como as das linguagens orientadas a objetos por exemplo. Com o passar do tempo e o surgimento de novos conceitos e tecnologias como os sistemas de controle de versão e documentação de erros, a Internet e a grande quantidade de código-fonte que foram disponibilizados livremente em repositórios na web, nunca houve tanta informação disponível para ser processada e utilizada como fonte para pesquisas. Em decorrência deste potencial a ser explorado, vários pesquisadores da área de software propuseram mecanismos de extração de dados e produziram diversos trabalhos, consolidando então a evolução de softwares como um nova área de pesquisa.

Este capítulo apresenta as principais pesquisas e ferramentas de evolução de software (*Software Evolution*). É objetivo também fornecer um conjunto de idéias do que pode ser feito na área de circuitos integrados digitais. Cabe aqui ressaltar que embora exista a distinção fundamental do que é hardware e o é software, existem grandes similaridades entre os dois, principalmente na maneira como são projetados, desenhados e

implementados. Pode-se dizer que as linguagens HDL tiveram seu êxito incorporando as facilidades introduzidas pelas linguagens de software, que foi a área aonde originaram os conceitos utilizados hoje com relação as abstrações, a sintaxe, e a estrutura de código [Ashenden & Lewis, 2008]. Da mesma maneira como a área de software já contribuiu para a área de circuitos integrados, os conceitos que foram e que são empregados em pesquisas na área de evolução de software podem ser empregados no desenvolvimento de circuitos integrados digitais.

3.1 Métricas

São medidas quantitativas de atributos de software ou hardware. As métricas são utilizadas extensivamente nas atividades de desenvolvimento e gestão para definir aspectos como por exemplo tamanho, quantidade, complexidade, desempenho e custo.

3.1.1 Métricas de software

Para a implementação de circuitos integrados digitais não existe um referencial teórico que trata sobre suas métricas assim como é feito na engenharia de software. Contudo, existem semelhanças entre estas duas áreas e muito pode ser aproveitado. Na área de software as métricas são classificadas como sendo de **produto** ou **processo**. Estas são definições de Fenton & Pfleeger [1997] e Kan [2002], apresentadas também por Gousios [2009] e Nacif [2011], detalhadas a seguir:

- **Produto:** são métricas dos produtos de qualquer espécie obtidos em um processo de desenvolvimento. No caso de software se enquadram nesta categoria as métricas de seus atributos específicos, que ainda podem ser subdivididos em:
 - Internos: são medidas feitas no software propriamente, como por exemplo nas estruturas de seu código-fonte.
 - Externos: são medidas do software com relação aos aspectos que podem ser percebidos externamente. Exemplo: quando o software é executado.
- **Processo:** medem atributos relativos às atividades do desenvolvimento de software. Se enquadram nestes atributos medidas de esforço, tempo, revisões e erros.

O SWEBOK¹ [Abran et al., 2004] é o livro de referência utilizado na engenharia de software, e que estrutura os principais tópicos da área. Até a versão de 2004, a

¹Sigla para o nome do livro escrito, *Software Engineering Body of Knowledge*. O sítio do livro na Internet é: <http://www.computer.org/portal/web/swebok/>.

utilização de métricas é discutida apenas dentro de determinados temas. Em 2008 foi proposto o capítulo Software Measures [Abran et al., 2008] dedicado exclusivamente às métricas, e que apresenta o contexto geral no qual elas se inserem. A Figura 3.1 ilustra estes contextos.

3.1.2 Métricas de software aplicáveis a circuitos integrados

Em Schafers [1995] é proposta a utilização de métricas de complexidade para análise das estruturas de código HDL. Estas métricas foram criadas para medir a “complexidade psicológica” de softwares, e podem ser adaptadas para linguagens de hardware. Este tipo de complexidade é fundamentada na dificuldade que os desenvolvedores têm de manter, modificar e entender o código, e são influenciadas por propriedades como:

- tamanho;
- aninhamento;
- controle de fluxo;
- fluxo de dados;
- hierarquia;
- localidade;
- regularidade;
- modularidade;
- acoplamento entre módulos (instâncias);
- concorrência;
- temporização.

As métricas reunidas por Schafers [1995] são:

- Medida de dificuldade e de volume de um software: é baseada no número de operadores e operandos [Halstead, 1977].
- Complexidade ciclomática: tenta determinar o número de caminhos de execução de uma função, baseando-se nas estruturas de controle de fluxo e iteração do código-fonte [McCabe, 1976].

- NPATH: conta o número de caminhos de execução acíclicos de uma função, se baseia em estruturas de controle de fluxo, iteração e expressões matemáticas [Nejmeh, 1988].
- Entropia de Harrison: é uma medida de complexidade que se baseia quantidade de referências a um mesmo operador no código-fonte, onde um operador pode ser uma função, uma palavra reservada ou um símbolo especial [Zuse, 1990].
- Medida de legibilidade: cálculo de complexidade que se baseia na contagem do número de espaços, caracteres, caracteres de variáveis, variáveis, operadores aritméticos, linhas de comentários, e linhas sem código-fonte [Jørgensen, 1980].

Apesar do trabalho de Schafers [1995] não ser muito recente, não foram encontradas citações ou outro tipo de referência além da documentação da ferramenta HCT [Maurer & Sviridenko, 2009]. Em Meeuws et al. [2007] 24 métricas de software são extraídas utilizadas para montar um modelo de predição de área do circuito e auxiliar na partição hardware/software. Este trabalho avalia descrição de circuitos implementados em linguagem C, que posteriormente é compilado para VHDL, não sendo feitas adaptações das métricas de software diretamente para HDL.

3.1.3 Métricas de circuitos integrados

Os processos de desenvolvimento de circuitos integrados são multidisciplinares. Existem trabalhos de pesquisa que focam em partes específicas como na implementação, na verificação, no teste ou na fabricação. Não existe nesta área uma referência que aborde a totalidade de seus tópicos como o SWEBOK faz na área de software. Pode-se afirmar que o ponto forte da literatura na área de CIs é o processo de fabricação, o qual sempre foi o foco das pesquisas [Rabaey et al., 2003; Hodges et al., 2003]. Outro foco de pesquisas são em verificação e testes, impulsionadas principalmente pelo aumento da complexidade e o tempo de mercado [Burch et al., 1994].

Portanto, não existe ainda uma literatura vasta que trata exclusivamente da parte do desenho e implementação do código HDL e suas métricas. Estas etapas se assemelham muito com a área de software, e devido a este fato, suas principais referências são inspiradas na engenharia de software. Em Piziali [2004], livro da área de verificação de circuitos, as métricas são classificadas de acordo com sua origem e tipo: de especificação ou implementação, e intrínsecas (interna) ou extrínsecas (externas). Considerando a origem destes diferentes contextos de pesquisa apresentados, as métricas de circuitos integrados aqui são classificadas como sendo de **implementação**, **verificação**, **teste** ou **síntese**.

3.1.3.1 HDL Complexity Tool

O *HDL Complexity Tool* (HCT) é uma ferramenta que extrai dados de complexidade de código HDL [Maurer & Sviridenko, 2009]. São suportadas as linguagens Verilog, VHDL e Cyclivity CDL. Sua principal métrica é a complexidade ciclomática baseada na métrica de software proposta por McCabe [1976]. As métricas disponíveis são:

- número de sinais de entrada/saída de um módulo;
- número de fios;
- complexidade ciclomática;
- linhas de código;
- número de linhas comentadas.

3.1.3.2 JasperGold

O JasperGold[®] [Jasper Design Automation, 2010] é uma ferramenta comercial utilizada para verificação de circuitos digitais. Após a síntese é possível extrair métricas estruturais e funcionais dos módulos utilizando o comando `get_design_info`. As métricas utilizadas neste trabalho foram:

- número de registradores tipo *flip-flops*;
- número de registradores tipo *latch*;
- número de portas lógicas;
- número de fios;
- número de sinais de entrada e saída;
- número de linhas de código RTL;
- número de instâncias declaradas ou submódulos.

3.1.3.3 Verilog-Perl

Este é um arcabouço de código aberto que reconhece as linguagens Verilog e System-Verilog. Este arcabouço é dividido em duas partes. O analisador léxico e sintático de Verilog, que está escrito em linguagem C++, e alguns utilitários estão escritos em Perl [Snyder, 2009]. Realizando algumas modificações no código-fonte foi possível implementar métricas de estruturas do código. Estas métricas são:

- número de palavras-chave;
- número de operadores;
- número de símbolos;
- número de comentários;
- número de atributos;
- número de constantes numéricas;
- número de diretivas pré-processadas;
- número de cadeia de caracteres.

3.1.3.4 BugLanguage

A BugLanguage [Cardoso et al., 2009] é uma linguagem de descrição de modificações desenvolvida para facilitar o preenchimento dos relatórios de revisão em ferramentas de controle de versão, e depois simplificar a extração de dados. A linguagem é composta por um conjunto de rótulos que objetivam pré-formatar o texto submetido pelo desenvolvedor segundo uma estrutura de dados definida. Os dados obtidos representam a motivação da modificação de cada revisão. A contagem de revisões cada tipo de modificação são novas métricas:

- revisões de continuação;
- revisões de modificação;
- revisões de simplificação;
- revisões de otimização;
- revisões de documentação;
- revisões de correção de erros, encontrados por:
 - inspeção;
 - simulação;
 - síntese.

3.1.3.5 Métricas de verificação

As métricas de cobertura pertencem ao domínio da verificação, e são utilizadas para quantificar o número de funcionalidades do circuito que poderão ser exercitadas. Estas métricas são obtidas das estruturas de código da implementação em RTL, e também podem ser utilizadas para avaliar a evolução do circuito. Em Piziali [2004] são apresentadas as métricas de cobertura:

- Linha de código: reporta linhas que podem ser executadas ou não.
- Instruções: instruções que podem ou não ser executadas. É diferente de cobertura de linha, onde podem existir tipos diferentes de instrução, e mais de uma por linha.
- Desvio: conta o número de transferências de controle de fluxo feitas por meio de estruturas de controle de fluxo e iteração.
- Condições: conta cada permutação de termos booleanos que satisfazem uma condição verdadeira ou falsa.
- Eventos: conta o número de vezes que um evento é disparado.
- Sinais: conta quantas vezes um bit de um registrador foi trocado.
- Máquina de estados: conta as transições entre os estados e quantas vezes um estado foi alcançado.

3.2 Ferramentas

Uma ferramenta apresentada recentemente para pesquisas na área de software é o Kenyon (Bevan et al. [2005]), desenvolvido para o acompanhamento da evolução de softwares. Esta foi arquitetada para ser compatível com várias ferramentas de gerenciamento de configuração de software. Por meio de uma interface, módulos para cada FGC que se deseja utilizar podem ser implementados e utilizados. As informações coletadas alimentam um banco de dados relacional abstraído por uma camada de mapeamento objeto-relacional. Por meio de uma interface abstrata, outras métricas também podem ser implementadas e adicionadas à esta ferramenta de extração.

Com o objetivo de obter uma riqueza de detalhes superior à que os softwares de controle de versão oferecem, em Robbes [2007] foi desenvolvido um repositório baseado em modificações na estruturas do código-fonte. Estas estruturas são representadas

por uma árvore de sintaxe abstrata ou AST (*Abstract Syntax Tree*), e as modificações são armazenadas neste nível. A argumentação para este trabalho é que sistemas de controle de versão armazenam apenas modificações em determinados instantes e que grande parte da evolução não é armazenada.

Em Girba et al. [2005] são utilizados logs de repositórios CVS [Fogel & Bar, 2003] para estudar a participação de cada desenvolvedor no processo de implementação de softwares de código aberto. Neste estudo foram analisados como cada participante interagiu em cada parte do software desenvolvido. Para tal experimento foi implementada a ferramenta Chronia.

No trabalho apresentado por Greevy et al. [2006] são apresentadas análises de evolução de software em uma abordagem mista, com análise estática de códigos-fonte em repositórios de arquivos com controle de versão. Neste são comparados as diversas versões armazenadas e analisadas as modificações no código fonte quanto a sua motivação, complexidade e acoplamento de estruturas. Estudar estes fatores por meio de métricas ajuda também a entender como os erros são propagados durante o desenvolvimento de software.

Para esta finalidade por exemplo, pode-se utilizar métricas para um fator de susceptibilidade a erro. Em Zimmermann & Zeller [2005] são estudadas mudanças no código com a finalidade de descobrir erros. No trabalho apresentado por Nacif et al. [2008] é demonstrada a relevância do uso de informações provenientes do histórico de revisões para alocação de recursos de verificação. Neste trabalho houve a necessidade de mecanismos de extração automatizados, mas que foram sugeridos como trabalhos futuros.

3.3 Visualização

Uma das maneiras de analisar os dados obtidos é por meio da representação gráfica, que torna o entendimento mais intuitivo. Fato que na maioria dos trabalhos sobre evolução de software são encontrados gráficos e alguns deles são apresentados como novas propostas de visualização. Em Wu [2003] é apresentada uma dissertação de mestrado que trata apenas de formas de visualização do conteúdo de SCVs. A seguir são apresentados alguns mecanismos de visualização interessantes de serem posteriormente observados no contexto do desenvolvimento de circuitos integrados.

RelVis [Pinzger et al., 2005] é uma ferramenta desenvolvida para a visualização da evolução do código-fonte e de suas métricas obtidas de várias revisões. Um gráfico de radar com vários eixos é utilizado para representar múltiplas métricas de um módulo

de software. Métricas de uma mesma revisão são interligadas por arestas que cortam os eixos. A variação entre os valores de métricas para as diferentes revisões observadas são ilustradas por cores que preenchem os espaços entre as arestas de uma revisão e outra. As relações entre um módulo do software e outro são ilustradas por meio de arestas que interligam os gráficos da maneira como vértices são interligados em um grafo. Um recorte deste gráfico é ilustrado na Figura 3.2.

Na ferramenta Chronia [Girba et al., 2005] é apresentado um gráfico denominado “Mapa de Posse”². Neste mapa são representados o tempo, os arquivos, e as contribuições feitas pelos autores do código, onde cada um deles é identificado por uma cor. Por meio deste gráfico foi possível identificar diferentes padrões de interação dos desenvolvedores com o código modificado. A Figura 3.3 ilustra este gráfico.

O Release History Database (RHDB) [Fischer et al., 2003] foi um dos primeiros softwares criados para extrair informações de revisões e armazená-las em um banco de dados para finalidade de pesquisa de evolução de software. Para ilustrar as informações obtidas no RHDB, um gráfico de dois eixos (tamanho e tempo), ilustrado na Figura 3.4, é utilizado para ilustrar a evolução do código-fonte do software Mozilla³ durante 56 revisões. Uma escala de cores é utilizada para mostrar que 50% do código foi modificado, e 25% foi adicionado na quarta parte final do tempo de desenvolvimento.

Em Karanam & Akepogu [2009] são apresentados modelos de visualização que servem para representar cada das 8 leis da evolução de software, e que poderão inspirar modelos de visualização para evolução de circuitos integrados.

Voinea & Telea [2007] apresenta um conjunto de ferramentas e técnicas para análise de dados dos repositórios CVS e Subversion (SVN) [Collins-Sussman et al., 2007]. Trechos de código-fonte são analisados com relação ao número de modificações feitas, e depois remontados em gráficos que ilustram a intensidade e a motivação destas modificações. A Figura 3.5 ilustra um destes gráficos.

Em Langelier et al. [2005] são apresentados gráficos tridimensionais para apresentação de grandes quantidades de métricas.

²Tradução literal de *The Ownership Map*.

³Navegador web de código aberto disponível em: <http://www.mozilla.org/>.

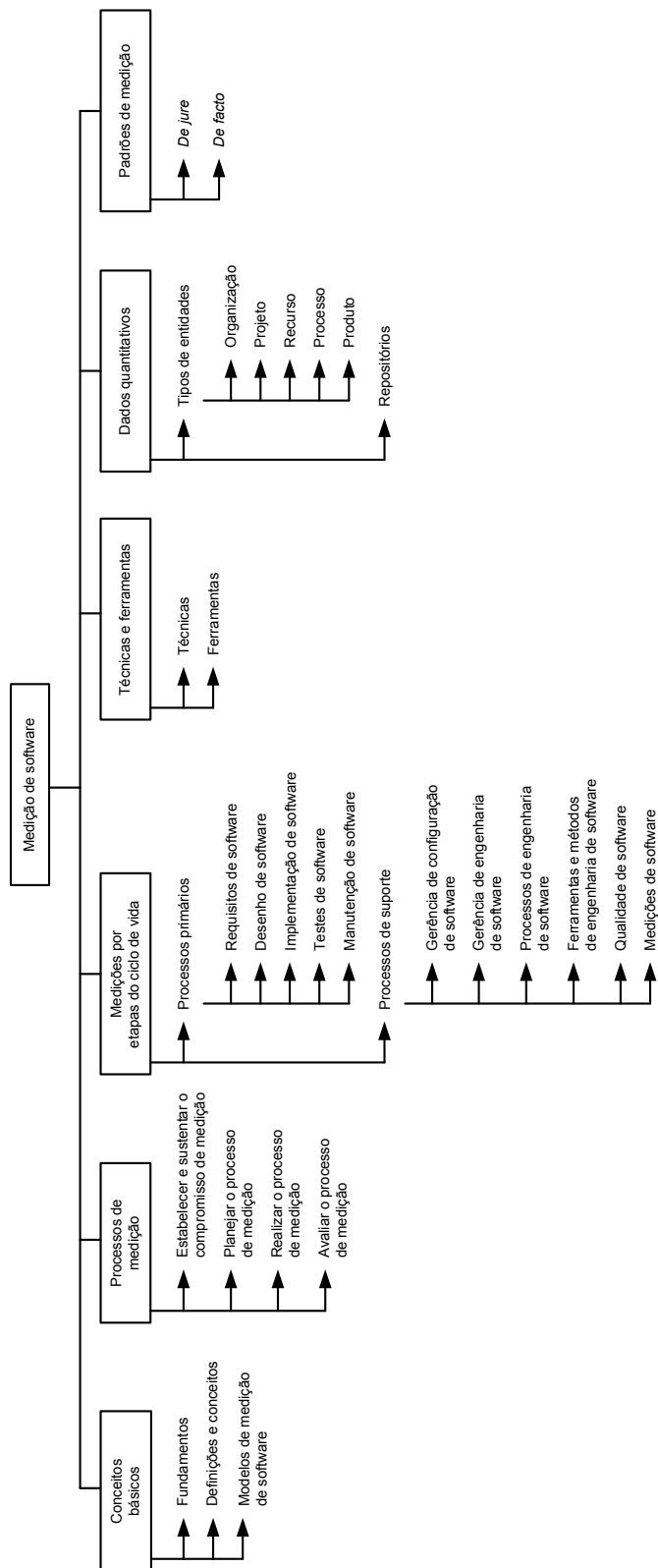


Figura 3.1. Contexto de utilização de métricas apresentado na proposta do capítulo “Software Measurement” [Abran et al., 2008] do SWEBOK.

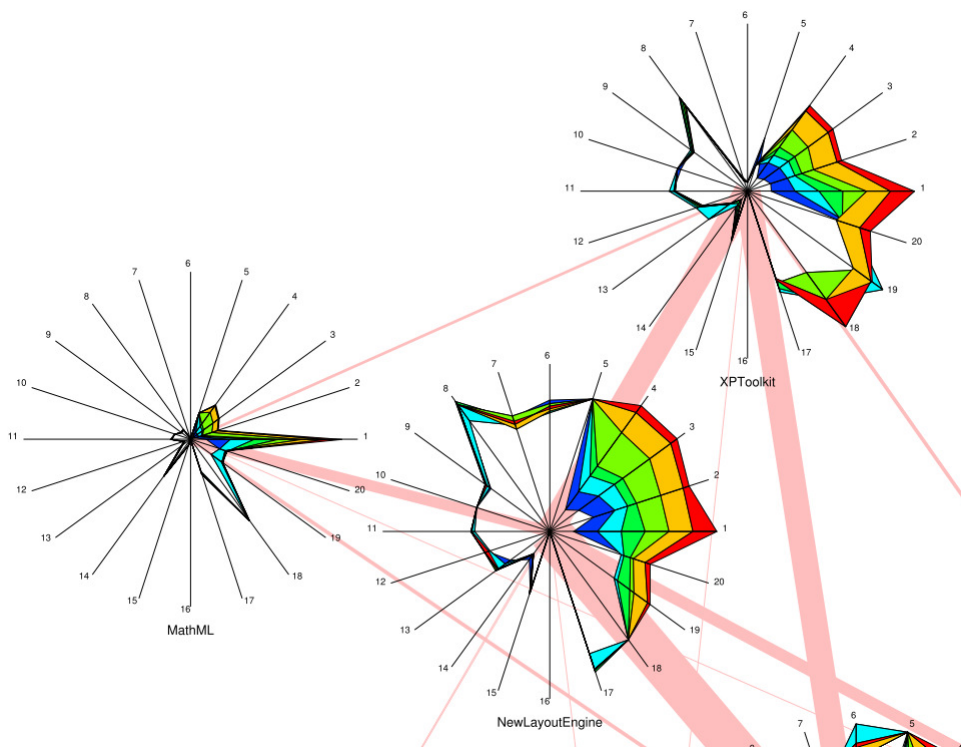


Figura 3.2. Recorte do gráfico de radar apresentado em Pinzger et al. [2005].

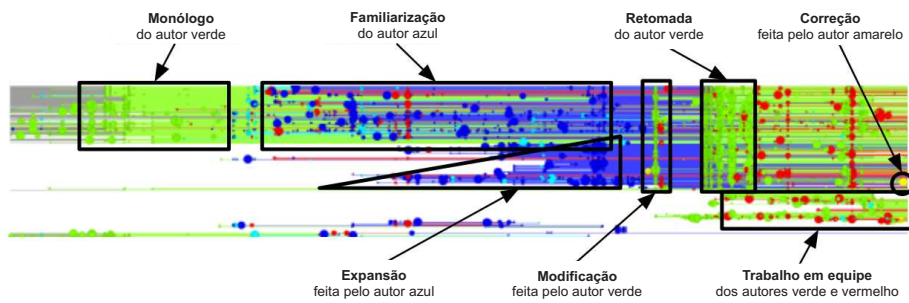


Figura 3.3. Mapa de posse apresentado em Girba et al. [2005].

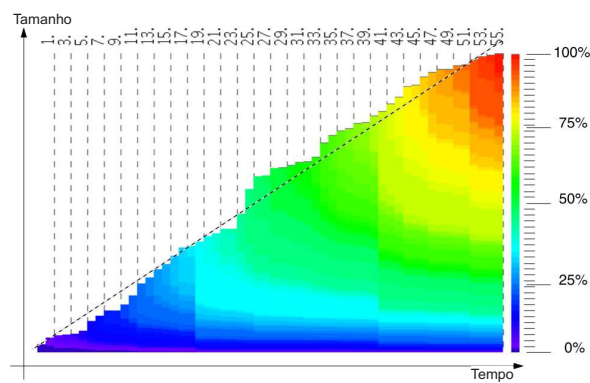


Figura 3.4. Gráfico apresentado em Fischer et al. [2003].

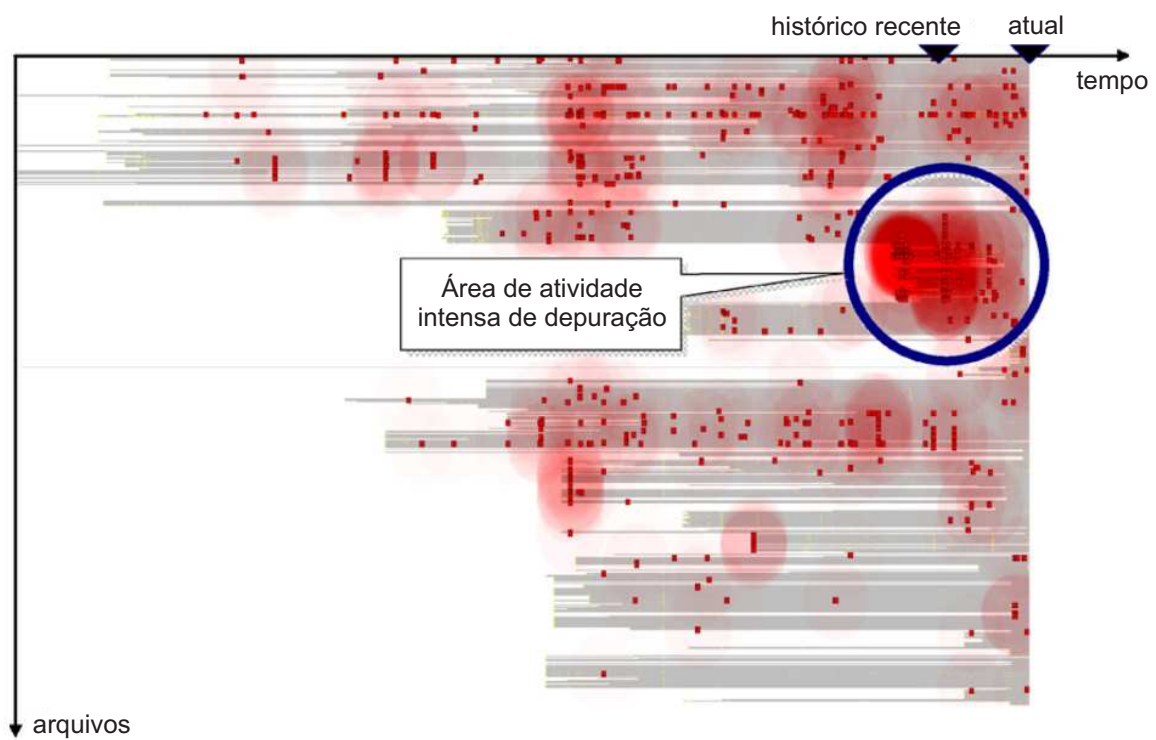


Figura 3.5. Locais de erros no código-fonte do Firefox [Voinea & Telea, 2007].

Capítulo 4

EyesOn: Ferramenta de extração

Os dados armazenados pelas ferramentas de gerência de configuração têm muito a oferecer para a melhoria dos processos de desenvolvimento. O grande número de trabalhos encontrados na área de engenharia de software comparado aos poucos na área de hardware mostra que ainda existe muito para ser explorado. Algumas etapas do processo de desenvolvimento de software e hardware possuem semelhanças que possibilitam que trabalhos publicados na área de software possam ser também analisados no contexto do desenvolvimento de circuitos integrados.

A evolução de hardware é um novo tema de pesquisa. As ferramentas desenvolvidas para software não oferecem recursos para análise de código-fonte HDL e precisariam ser adaptadas. Infelizmente, devido a falta de documentação, a adaptação dessas ferramentas não é uma tarefa fácil. Dessa forma, optou-se por desenvolver uma ferramenta expansível para projetos desenvolvidos em HDL.

O propósito da ferramenta EyesOn é viabilizar a realização das primeiras pesquisas na área de evolução de hardware num ambiente acadêmico, permitindo que este conceito seja validado e demonstrando um novo potencial a ser explorado. Apesar de várias tentativas, para este trabalho não foi possível conseguir dados industriais. Para que estes pudessem ser utilizados, alguma empresa desenvolvedora teria de disponibilizar seu código-fonte expondo informações como revisões e também os erros encontrados. Contudo, esta ferramenta é de código aberto e poderá também ser utilizada por pesquisadores da indústria. No Capítulo 5 é apresentado o ambiente de experimentação utilizado e o processo de obtenção dos dados.

4.1 Como funciona a extração dos dados

Existem dois componentes que armazenam dados: repositório e banco de dados. O repositório representa os dados das ferramentas de gerência de configuração no seu formato bruto, e geralmente em grande volume. O banco de dados representa a estrutura do arcabouço na qual os dados ficam organizados em tabelas. Esta estrutura relacional do banco de dados permite a padronização e facilita a realização de consultas, feitas por SQL (*Structured Query Language*).

Os softwares de controle de versão foram desenvolvidos para o armazenamento e não para consulta. Se comparados com os bancos de dados relacionais pode-se dizer que o acesso a estes repositórios é lento, e dependem de ferramentas de manipulação que não são específicas para consulta. Como o formato de consultas não é estruturado, muitas informações desnecessárias são retornadas. Por outro lado, não é necessário armazenar novamente, por exemplo, o conteúdo de todas as versões de todos os arquivos. Estes podem ocupar muito espaço em disco desnecessariamente uma vez que podem ser recuperados dos repositórios sob demanda usando a informação do número de revisão e caminho na árvore de diretórios. O ideal é coletar os dados básicos dos repositórios armazenando no banco de dados apenas a linha do tempo e dados que ocupam pouco espaço como: revisões, autores, relatórios de modificação e nomes dos arquivos e diretórios modificados. Em uma consulta posterior podem ser buscadas outras informações. A Figura 4.1 ilustra os fluxos de dados na ferramenta EyesOn.

Após o pré-processamento, o segundo passo é extrair as métricas. Uma vez que a linha do tempo é conhecida a extração pode ser programada para um conjunto conhecido de dados. As informações necessárias para extração são provenientes do repositório e do banco de dados. Alguns casos de extração seriam:

- Métricas de arquivos: para a extração das métricas de código-fonte HDL é necessário fazer um **checkout** do arquivo em um determinada revisão. Dependendo do que for analisado, pode ser necessário remontar toda a árvore de diretórios.
- Métricas de erros: existem diversos locais para se extrair dados de erros. Estes podem ser relatados nos relatórios de modificação, nos arquivos do código-fonte e também em banco de dados de sistemas de rastreamento de erros.
- Métricas do repositório: métricas de repositório são números extraídos dos dados guardados no repositório: número de revisões, número de modificações, número de pessoas que fizeram modificações, etc.

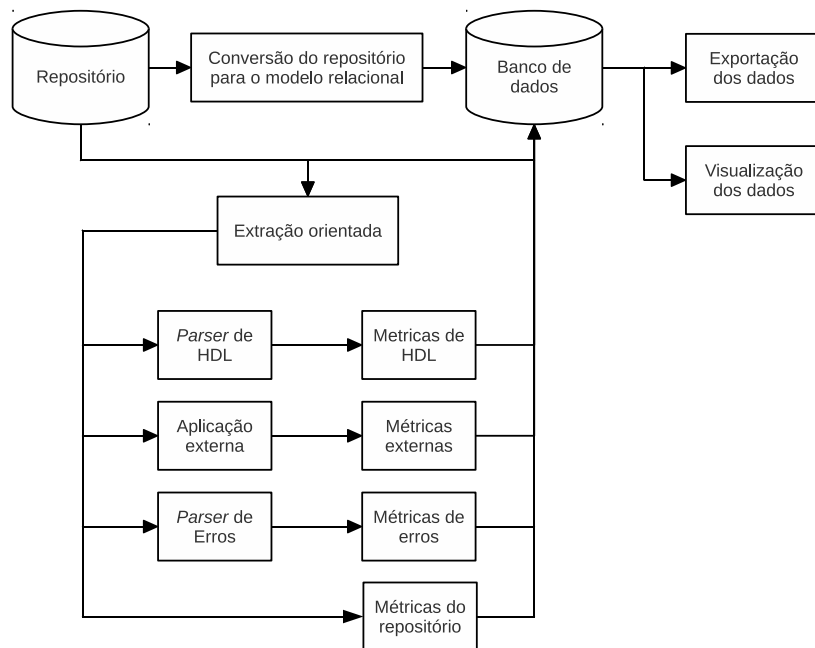


Figura 4.1. Fluxos de dados na ferramenta.

- **Métricas externas:** as informações do repositório podem ser repassadas para ferramentas externas que processam os dados e retornam valores de métricas para o EyesOn. Exemplos seriam análise estática de um arquivo de código-fonte, dados extraídos de ferramentas de síntese, ou uma análise da mensagem de texto do relatório de modificações feitas em uma revisão. O relatório de erros preenchido com uma linguagem de descrição de erros se encaixa nessa categoria.

Após estas duas etapas de extração, as métricas ficam armazenadas no banco de dados. As métricas são então relacionadas aos dados básicos do histórico de revisões, podendo ser visualizados ou exportados. A Seção 4.2 apresenta estes cenários de utilização.

4.2 Cenários de utilização

Esta ferramenta foi desenhada para ser utilizada em pesquisas acadêmicas, podendo também ser utilizada na indústria. A diferença entre suas duas aplicações é que na indústria é mais comum o acompanhamento dos processos. Devido a este fato é provável que no futuro exista uma demanda maior por metodologias diferentes de visualização dos dados, permitindo um acompanhamento mais eficiente da evolução do circuito. Por outro lado, os dados extraídos ainda podem ser utilizados em outras pesquisas. Portanto são listados dois contextos de utilização:

1. Apresentação dos dados na forma de gráficos: estes dados seriam sobre versões, erros e métricas. Neste cenário são apresentadas informações estatísticas do desenvolvimento, que podem se tornar interessantes de serem utilizadas como instrumento de gestão. Este é o cenário típico para o acompanhamento no qual são visualizadas as informações ao longo das atividades de desenvolvimento.
2. Exportação dos dados coletados para softwares externos: a estrutura relacional do banco de dados facilita a exportação dos dados para ferramentas externas. A partir deste ponto começa a análise de evolução na qual todas as informações necessárias já foram coletadas e estão organizadas no banco de dados.

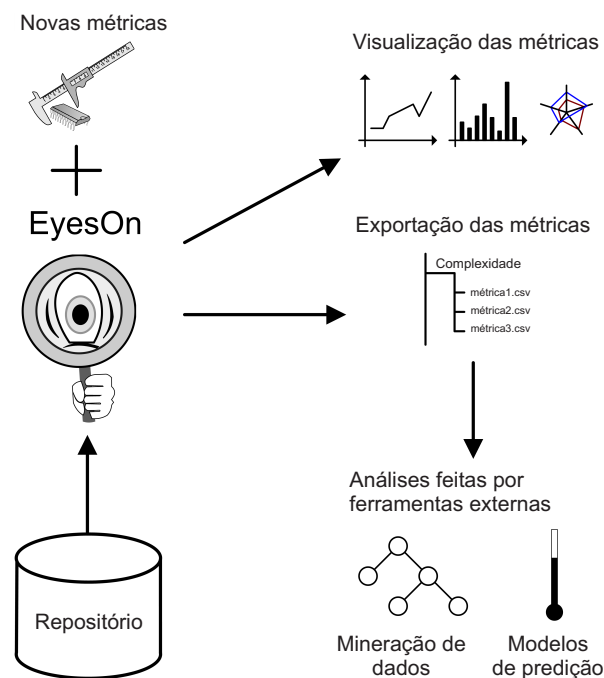


Figura 4.2. Contextos de utilização

4.3 Arquitetura

Este arcabouço foi projetado para permitir a integração com diferentes sistemas de controle de versão, e também com diferentes tipos de métricas. O núcleo da ferramenta é composto por classes que representam as entidades principais dos SVCs e das métricas. Cada uma destas classes que compõem o núcleo é uma entidade persistente¹ que armazena dados e é referenciada no código-fonte do arcabouço para passagem de valores.

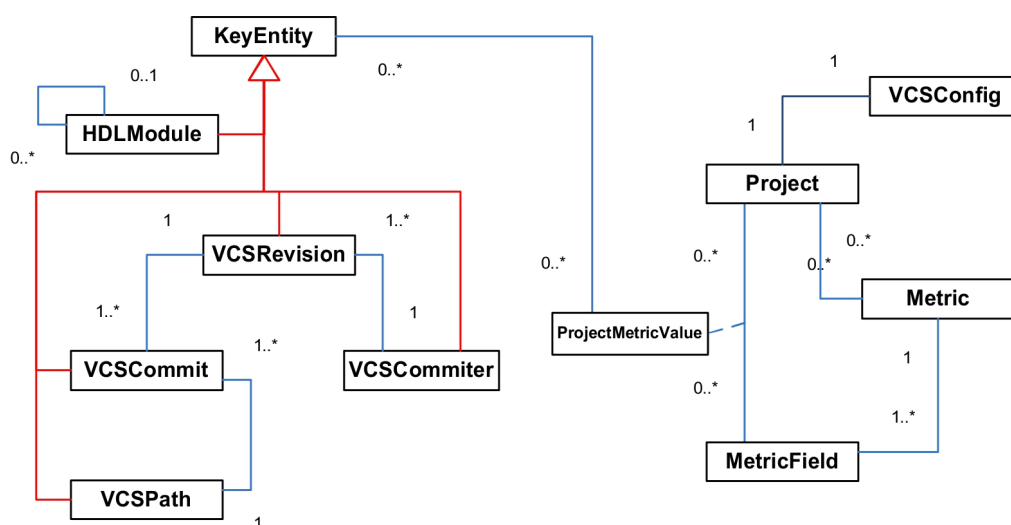


Figura 4.3. Arquitetura do núcleo da ferramenta.

Estas entidades se relacionam entre si, e podem ser estendidas. Na Figura 4.3 é ilustrado um diagrama de classes da UML² com as principais associações, destacadas em azul, e generalizações de uma classe deste núcleo, destacadas em vermelho. As associações são relacionamentos entre duas classes, ilustradas por uma aresta. O rótulo de multiplicidade delimita os relacionamentos. Lê-se: muitos (*), um (1) ou zero (0). Quando combinados lê-se: zero ou um (0..1), zero ou muitos (0..*), e pelo menos um ou muitos (1..*). Cada lado da associação possui um rótulo que indica a multiplicidade de um classe para com a outra. A generalização é ilustrada por um aresta com uma seta indicando a classe especializada. A seguir são detalhadas cada uma das classes do núcleo:

- **Project:** classe que representa um projeto. Todos os dados armazenados estão

¹Uma entidade persistente é uma classe utilizada para o armazenamento de dados, e que geralmente são persistidos em uma base de dados com a finalidade de posteriormente serem recuperados.

²Linguagem Unificada de Modelagem, tradução de *Unified Modeling Language*. Sítio de referência na Internet: <http://www.uml.org/>.

vinculados a um objeto desta classe. São membros de um projeto a configuração do repositório de código-fonte e um conjunto de métricas.

- **VCSConfig**: representa as informações de acesso ao sistema de controle de versão.
- **Metric**: esta classe representa um conjunto de dados de uma mesma origem. Todas as métricas do arcabouço são derivadas desta classe. São membros desta classe um conjunto de campos que instanciam a classe **MetricField**.
- **MetricField**: campo de métrica vinculado a uma métrica. Esta classe apenas denomina o nome de um campo, servindo para distinguir cada um dos tipos de dados de um mesmo conjunto de métricas. Qualquer classe que estender **Metric** possuirá campos de métrica do tipo **MetricField**.
- **ProjectMetricValue**: responsável por armazenar o valor de um campo de métrica.
- **KeyEntity**: entidade chave. Esta classe é utilizada como chave primária de todas suas classes derivadas. Os valores de campos de métricas fazem referência a esta classe permitindo que qualquer dado de métrica armazenado possa ser referente a uma classe derivada desta. Exemplos: a uma revisão, a um módulo HDL etc.
- **VCSRevision**: representa uma revisão. São dados de uma revisão a pessoa que enviou a modificação, o projeto, o número da revisão, o relatório de modificação e a data/hora do envio.
- **VCSCommit**: representa as modificações na estrutura de arquivos e diretórios. Exemplo: um arquivo ou diretório adicionado, removido, modificado etc.
- **VCSCommitter**: representa a pessoa que enviou a modificação.
- **VCSPath**: representa o arquivo ou diretório referenciado.
- **HDLModule**: representa um módulo HDL, que pode fazer parte de um módulo maior e também possuir submódulos.

Esta ferramenta foi implementada sob o paradigma de orientação a objetos (POO), utilizando a linguagem Java³. O motivo desta escolha foi a agilidade de desenvolvimento proporcionada, a gratuidade da licença, e a capacidade de integração com diversos arcabouços de código aberto disponíveis na Internet.

³<http://www.java.com/>

4.3.1 Novas métricas

Novas métricas podem ser incorporadas ao EyesOn por meio de extensão de seu código-fonte. Na Figura 4.4 é ilustrado um diagrama de classes demonstrando a classe `Metric`, e também as classes `HCT`, `VParser`, `JG`, `BugLangMetrics`, e `OpenSPARCBugCount` que representam as métricas implementadas e são derivações da primeira. A seta indica a classe estendida. O retângulo de linha pontilhada denominado `OutraNovaMetrica` representa a continuação do arcabouço por meio da implementação de mais uma métrica.

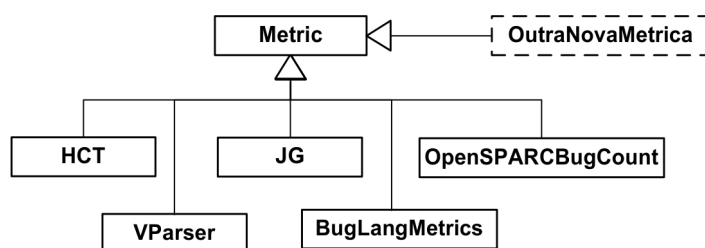


Figura 4.4. Ponto de extensão para novas métricas.

Todas as métricas devem especializar a classe `Metric`. Em cada uma delas deve ser implementado seu mecanismo de extração. Além disso devem ser também definidos algumas propriedades como nomes de seus campos, e também nome e descrição da métrica, permitindo sua identificação pelo usuário quando o mesmo for utilizar a ferramenta. A Figura 4.5 apresenta um exemplo de código de especialização da classe `Metric`.

4.3.2 Modelo de banco de dados

A estrutura do banco de dados segue o modelo de entidades persistentes apresentadas como núcleo da ferramenta. Os relacionamentos e os atributos destas classes foram mapeados do paradigma da orientação a objetos para banco de dados relacional. A utilização de um banco de dados relacional para o armazenamento das métricas foi uma maneira adequada para facilitar a realização de consultas e também agilizar o processo de extração de métricas.

Como a linguagem escolhida para implementação deste arcabouço foi Java, a solução utilizada mapeamento objeto-relacional foi o Hibernate⁴ que é um arcabouço popular e de código aberto. Por meio desta camada de persistência de dados o desenvolvimento desta ferramenta foi facilitado. As informações que são guardadas no banco de dados são definidas apenas em classes.

⁴Disponível em: <http://www.hibernate.org/>.

```
1 public class OutraNovaMetrica extends Metric {
2
3     ...
4
5     public static final String prvMetricName = "OutraNovaMetrica";
6     public static final String prvMetricDescription = "Metricas de...";
7     public static final String[] prvFieldNames = {
8         "IO",
9         "NET",
10        "FLOPS",
11        ...
12    };
13    public static final Class prvAttachmentType = HDLModule.class;
14
15    ...
16
17    public ProjectMetricValue[] extract(Object[] args) throws AppException {
18
19        ProjectMetricValue[] pmv;
20
21        ...
22
23        return pmv;
24    }
25 }
```

Figura 4.5. Exemplo de especialização da classe que representa uma métrica.

4.3.3 Compatibilidade com repositórios de SCVs

No EyesOn foi implementado uma interface de conexão para extração de dados do Subversion, devido a sua maior popularidade e por ter sido este o mesmo utilizado no ambiente de experimentação descrito no Capítulo 5. Embora cada tipo de SCV tenha seu próprio formato de dados, os tipos de dados mais importantes foram padronizados, de maneira que a integração com outras ferramentas possa ser feita sem o comprometimento das estruturas já definidas. A extração das métricas são atribuídas a cada revisão. Devido a esta organização é possível saber as métricas do código em uma determinada versão e também realizar o acompanhamento destas na medida em que o código evolui.

4.4 Métricas implementadas

As métricas implementadas são descritas na Tabela 4.1 e classificadas na Tabela 4.2. Os nomes das métricas são os mesmos dados às classes derivadas de `Metric`, e os campos de métrica são listados com os nomes dados às instâncias de `MetricField`, apresentados na Seção 4.3.

Tabela 4.1. Métricas utilizadas neste arcabouço.

Métricas	Campos de métricas
HCT	io: número de sinais de entrada/saída de um módulo; net: número de fios; mccabe: complexidade ciclomática; sloc: linhas de código; comment_lines: número de linhas comentadas.
JG	flops: número de registradores tipo <i>flip-flops</i> ; latches: número de registradores tipo <i>latch</i> ; gates: número de portas lógicas; nets: número de fios; ports: número de sinais de entrada e saída; lines: número de linhas de código; rtl_instances: número de instâncias declaradas ou submódulos.
VParser	keywords: número de palavras-chave; operators: número de operadores; symbols: número de símbolos; comments: número de comentários; attributes: número de atributos; numbers: número de constantes numéricas; preprocs: número de directivas pré-processadas; strings: número de cadeia de caracteres.
BugLangMetrics	resume: contagem de revisões de continuação; modification: contagem de revisões de modificação; simplification: contagem de revisões de simplificação; optimization: contagem de revisões de otimização; documentation: contagem de revisões de documentação; bugfix: contagem de revisões de correção de erros; bugfix_inspection: contagem de revisões de correção de erros encontrados por inspeção; bugfix_simulation: contagem de revisões de correção de erros encontrados por simulação; bugfix_synthesis: contagem de revisões de correção de erros encontrados por síntese.

Tabela 4.2. Classificação das métricas utilizadas neste arcabouço.

	Produto	Processo
Implementação	HCT: io, net, mccabe, sloc, comment_lines. JG: nets, ports, lines, rtl_instances. VParser: keywords, operators, symbols, comments, attributes, numbers, preprocs, strings.	BugLangMetrics: resume, modification, simplification, optimization, documentation, bug_inspection.
Verificação	-	-
Teste	-	BugLangMetrics: bug, bug_inspection, bug_simulation.
Síntese	JG: flops, latches, gates.	BugLangMetrics: bug_synthesis.

Capítulo 5

Ambiente de experimentação e resultados

Como recursos de suporte ao ambiente de experimentação da ferramenta EyesOn foram utilizados dados coletados de implementações do processador MIPS32, realizadas pelos alunos da disciplina Organização de Computadores II do segundo semestre de 2009. Estes dados também serviram como fonte para o trabalho apresentado por Nacif [2011]. A seguir é detalhado o experimento.

5.1 Implementação de um circuito para experimentação

Durante o segundo semestre de 2009 os alunos da disciplina Organização de Computadores II, ofertada pelo Departamento de Ciência da Computação da UFMG, implementaram uma versão simplificada do processador MIPS 32 bits. Esta implementação foi feita com base em uma especificação montada pelos monitores da disciplina, que aproveitaram uma implementação anterior do processador. Este processador foi desenhado com um conjunto simplificado das instruções, sem adicionar instruções de ponto flutuante, pré-busca, *traps*, multiplicação ou de co-processador. Para aumentar a complexidade e o aprendizado foram adicionadas algumas estruturas avançadas como *pipeline* e *caches*.

Esta implementação foi escolhida para servir de especificação para os alunos devido às suas interfaces e estruturas internas estarem bem desenhadas e organizadas. Durante o segundo semestre de 2009 cada um dos módulos foram especificados e divididos em doze partes para serem entregues durante o período letivo. Alguns módulos

foram agrupados para serem entregues juntos, e as datas de entrega foram organizadas de acordo com a complexidade dos módulos. O tempo médio de implementação de cada entrega foi de uma a duas semanas.

A versão implementada do processador MIPS32 é composta por 17 módulos que foram desenvolvidos durante o período de quatro meses. Para cada módulo os alunos receberam um documento com o detalhamento de suas interfaces. Dados os sinais de E/S, como ilustrado na Figura 5.1, os alunos foram orientados a implementar a parte comportamental do circuito. Na especificação do módulo foi incluído um texto explicando o como seria o funcionamento deste circuito e uma tabela descrevendo todos os sinais de entrada e saída, como exemplificado nas Tabelas 5.1 e 5.2. Também foi fornecido um arquivo inicial com os nomes padronizados e apenas o código das interfaces definidos.



Figura 5.1. Exemplo de módulo especificado.

5.1.1 Infraestrutura adotada e instruções de uso do software de controle de versão

Para cada grupo foi criado um repositório de arquivos utilizando o software SVN de controle de versão. Este software foi escolhido por ser difundido, de código aberto e de livre utilização, além de também possuir documentação e ferramentas acessíveis que podem ser encontradas na Internet. Nos repositórios foram criadas estruturas de diretórios padronizados como na árvore ilustrada pela Figura 5.2. Antes do trabalho prático ser passado aos alunos, eles receberam uma aula com as instruções básicas de utilização do Subversion. Eles também foram instruídos sobre como deveriam proceder para registrar as versões enviadas ao repositório, sempre preenchendo um breve relatório sobre as modificações feitas.

Após o trabalho ser explicado, as especificações dos módulos foram entregues em etapas para implementação. Para cada módulo especificado, um arquivo de código-

Tabela 5.1. Descrição dos sinais da ALU.

Sinal	E/S	Tamanho	Descrição
a	E	32 bits	Primeiro operando da ALU.
b	E	32 bits	Segundo operando da ALU.
aluout	S	32 bits	Resultado das operações lógicas aritméticas da ALU.
op	E	3 bits	Sinal de controle que indica qual operação deve ser executada: 000: a AND b; 001: a OR b; 010: a + b; (soma) 011: Sem operação associada; 100: a NOR b; 101: a XOR b; 110: a - b; (subtração) 111: Sem operação associada.
unsig	E	1 bit	Define se o sinal compout deve considerar as entradas a e b com ou sem sinal: 0: Com sinal; 1: Sem sinal.
compout	S	1 bit	Flag assume o valor 1 quando o operando a menor que o operando b. Em caso contrário esse flag deve assumir o valor 0.
overflow	S	1 bit	Flag que indica a ocorrência de um overflow (assume valor 1). As condições para ocorrência de overflow são apresenadas na Tabela 5.2 para operações com sinal.

Tabela 5.2. Condições de overflow para adição e subtração.

Operação	Operando a	Operando b	Resultado indicando overflow
a + b	≥ 0	≥ 0	< 0
a + b	< 0	< 0	≥ 0
a - b	≥ 0	< 0	< 0
a - b	< 0	≥ 0	≥ 0

fonte HDL contendo apenas as interfaces de entrada e saída era enviado para o repositório pelos monitores da disciplina, sendo colocados dentro do diretório /trunk/rtl. A declaração das interfaces era fixa e os alunos deveriam apenas implementar a parte funcional de cada módulo, conforme a especificação. Com a utilização do SCV cada

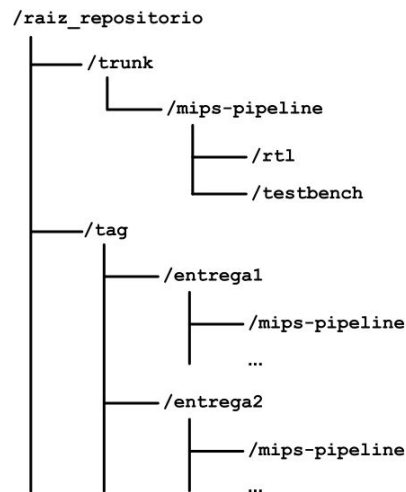


Figura 5.2. Estrutura de diretórios do repositório para cada grupo.

desenvolvedor modificava sua cópia dos arquivos e quando necessário enviava suas modificações para o repositório, ou atualiza sua cópia de trabalho sincronizando-a com a versão de sua equipe presente repositório. O diretório `/trunk/testbench` ficou reservado para que os alunos pudessem colocar os arquivos necessários para realização de testes de sua implementação.

Após a entrega dos módulos os alunos continuaram a implementação dos outros módulos podendo realizar pequenas alterações nos módulos já entregues, modificando sempre a versão atual no diretório `/trunk/rtl`. Portanto a orientação para entrega foi que os alunos criassem uma cópia da versão a ser entregue dentro do diretório `/tag/entregaN`, onde N é o número da entrega, copiando sempre o conteúdo de `/trunk` para dentro de `/tag/entregaN`.

5.1.2 Testes e documentação dos erros e modificações

Para cada um dos 17 módulos do processador foram implementados também módulos de teste ou *testbenches*. Os *testbenches* instanciam os módulos sob teste e provêm estímulos para suas entradas. Os módulos em teste processam estes sinais e retornam uma saída. Para saber se o módulo testado está correto, seus sinais de saída são comparados com um conjunto de sinais de referência (*Golden Model*). Como vetor de entrada do teste foram preparados um conjunto de sinais baseados nas instruções do processador, e que garantem o funcionamento correto de cada uma das funcionalidades do módulo. A Figura 5.3 ilustra este tipo de teste.

Durante o processo de implementação, os módulos foram constantemente modificados e suas revisões registradas no software controle de versão. Para documentação e

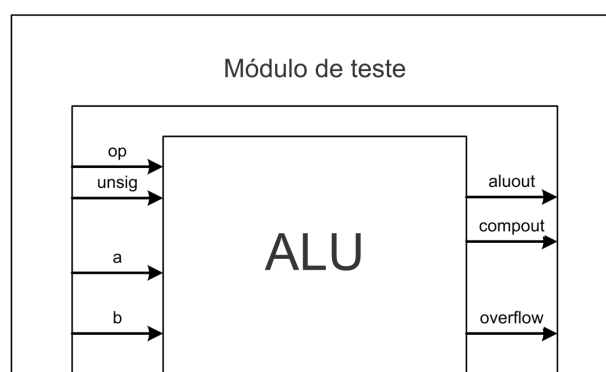


Figura 5.3. Módulo de teste e módulo sob teste.

depois a facilitação do rastreamento das modificações foi utilizada a BugLanguage. Os aplicativos BugReporter e BugRuler [Cardoso et al., 2009] também foram utilizados para auxiliar no preenchimento dos relatórios desta linguagem.

5.2 Dados obtidos do ambiente de experimentação

Ao final da implementação foram coletados os dados do experimento. Ao todo participaram 47 alunos de graduação e 4 alunos de pós-graduação divididos entre 12 grupos de trabalho. Destes grupos apenas 5 seguiram a metodologia de uso da ferramenta de controle de versão e documentou os erros de maneira satisfatória. A Tabela 5.3 enumera os módulos implementados e detalha o que foi aproveitado de cada grupo.

Tabela 5.3. Lista de módulos aproveitados por grupo.

Módulo	Grupo A	Grupo B	Grupo C	Grupo D	Grupo E
1 - ULA	✓(1)	✓(13)	✓(21)	✓(28)	✓(36)
2 - Comparador	✓(2)	✓(14)	✓(22)	✓(29)	✓(37)
3 - Controle	✓(3)	✓(15)	✓(23)	✓(30)	✓(38)
4 - Cache Dados					
5 - Decodificação	✓(4)		✓(24)	✓(31)	✓(39)
6 - Execução	✓(5)				
7 - Busca	✓(6)	✓(16)		✓(32)	✓(40)
8 - Repasse	✓(7)	✓(17)			✓(41)
9 - Geren. Cache					
10 - Geren. Memória					
11 - Cache Instruções					
12 - Memória	✓(8)	✓(18)		✓(33)	✓(42)
13 - MIPS	✓(9)				
14 - Memória RAM					
15 - Registradores	✓(10)	✓(19)	✓(25)	✓(34)	✓(43)
16 - Deslocador	✓(11)		✓(26)		
17 - Escrita	✓(12)	✓(20)	✓(27)	✓(35)	✓(44)

Dos módulos aproveitados foram extraídos dados das métricas do produto e do processo apresentadas na Tabela 4.2. Os dados do processo foram utilizados como base

para um modelo de propensão apresentado na Seção 5.3. Os dados do produto são relativos às medidas tomadas do circuito em cada etapa do desenvolvimento. Neste trabalho não foi investigado a utilização destes dados como parte do modelo de propensão. No entanto sua relevância é apresentada em Nacif [2011]. Na Seção 5.4 é apresentada a aplicação destas métricas para visualização da evolução do circuito.

5.3 Utilização de métricas do processo como base para um modelo de propensão

Uma das aplicações dos dados de evolução é utilizá-los como base para construção de modelos de propensão. Em Nacif [2011] é apresentada uma metodologia para identificação de módulos de circuitos integrados propensos a erro utilizando dados do histórico de desenvolvimento extraídos pelo EyesOn. Naquele trabalho são apresentados modelos estatísticos construídos com base nos dados do histórico do MIPS e do OpenSPARC.

Neste tópico é apresentado um caso de estudo que utiliza dados do histórico de revisões para apontar um subconjunto de módulos com maior susceptibilidade de conter erros que ainda não foram descobertos. O algoritmo apresentado pela Figura 5.3 usa o conceito de memória cache para armazenar apenas os **três** módulos Mais Frequentemente Modificados. Este algoritmo parte do princípio que da localidade temporal também pode ser utilizada para propensão a erros. Um trabalho semelhante é apresentado em Hassan & Holt [2005].

Este algoritmo utiliza dois critérios para determinação de quais módulos foram mais frequentemente modificados. O primeiro critério é a classificação da frequência, representado por (3) muito frequente, (2) frequente, e (1) carregado na cache. Na medida em que os módulos são corrigidos uma nova posição na cache é populada. Se um módulo é mais frequentemente modificado ele tem maior chance de permanecer na cache. O segundo critério serve para desempate. Quando dois módulos possuem a menor frequência, o mais antigo é removido da cache. A Tabela 5.4 apresenta o funcionamento deste algoritmo.

O nomes armazenados na cache a cada iteração do algoritmo (revisão) indicam os módulos como mais propensos a erro. Utilizando as informações obtidas do ambiente de experimentação este algoritmo foi realizado para quatro instâncias:

- Revisões de **correção** com caches de tamanhos 3 e 5.
- Revisões de **modificação** com caches de tamanhos 3 e 5.

```

1: inicializacao da Cache
2: for all modulos na ListaDeModulosModificados do
3:   if Cache esta vazia then
4:     armazena modulo com frequencia = 1
5:   else
6:     if modulo esta na Cache then
7:       frequencia ← frequencia + 1
8:       armazena ordem do modulo
9:     else
10:      if  $|Cache| = TAM\_MAX\_CACHE$  then
11:        escolha modulo com menor frequencia
12:        remova este modulo da Cache
13:      end if
14:    end if
15:    armazene o modulo com frequencia = 1
16:    armazene a ordem do modulo
17:  end if
18: end for

```

Figura 5.4. Algoritmo de alto nível para escolha dos módulos mais frequentemente modificados.

Tabela 5.4. Tabela de execução do algoritmo MFM.

Revisão	Posição 1	Posição 2	Posição 3	Frequência	Ordem	Próximo módulo
1	-	-	-	-	-	Alu
2	Alu	-	-	1	1	Alu
3	Alu	-	-	2	2	Alu
4	Alu	-	-	3	3	Alu
5	Alu	-	-	3	4	Ram
6	Alu	Ram	-	1	5	Ram
7	Alu	Ram	-	2	6	Registers
8	Alu	Ram	Registers	1	7	Alu
9	Alu	Ram	Registers	3	8	Shifter
10	Alu	Ram	Shifter	1	9	Shifter
11	Alu	Ram	Shifter	2	10	Comp
12	Alu	Comp	Shifter	1	11	Gdm
13	Alu	Gdm	Shifter	1	12	-

Foram escolhidos os tamanhos 3 e 5 porque o número de módulos é pequeno e estes já seriam valores razoáveis para uma instanciar uma cache. O tamanho 5 foi utilizado para comparação do ganho em acertos utilizando uma cache um pouco maior.

Estes valores representam respectivamente o espaço para identificar:

- 23,08% e 38,46% dos 13 módulos com erros documentados.
- 17,65% e 29,41% dos 17 módulos com modificações documentadas.

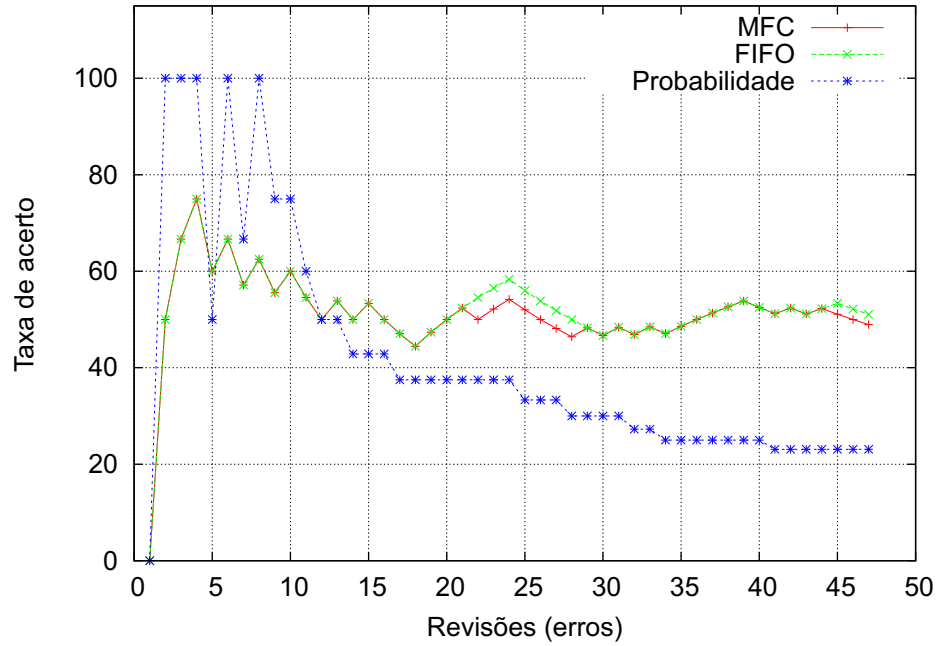


Figura 5.5. Revisões de correção. Estratégias MFC, FIFO e probabilidade, cache de tamanho 3.

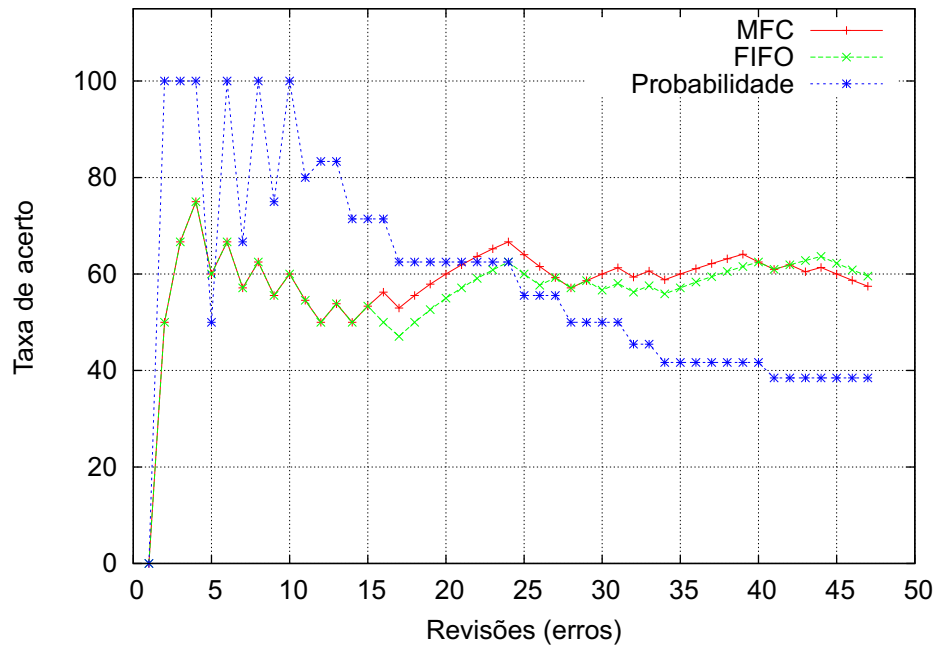


Figura 5.6. Revisões de correção. Estratégias MFC, FIFO e probabilidade, cache de tamanho 5.

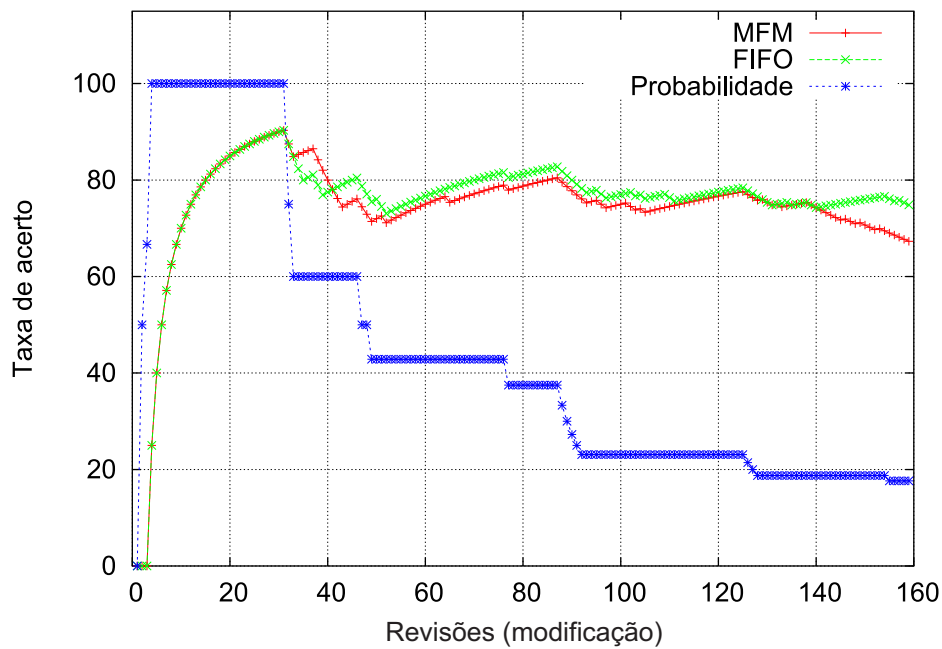


Figura 5.7. Revisões de modificação. Estratégias MFM, FIFO e probabilidade, cache de tamanho 3.

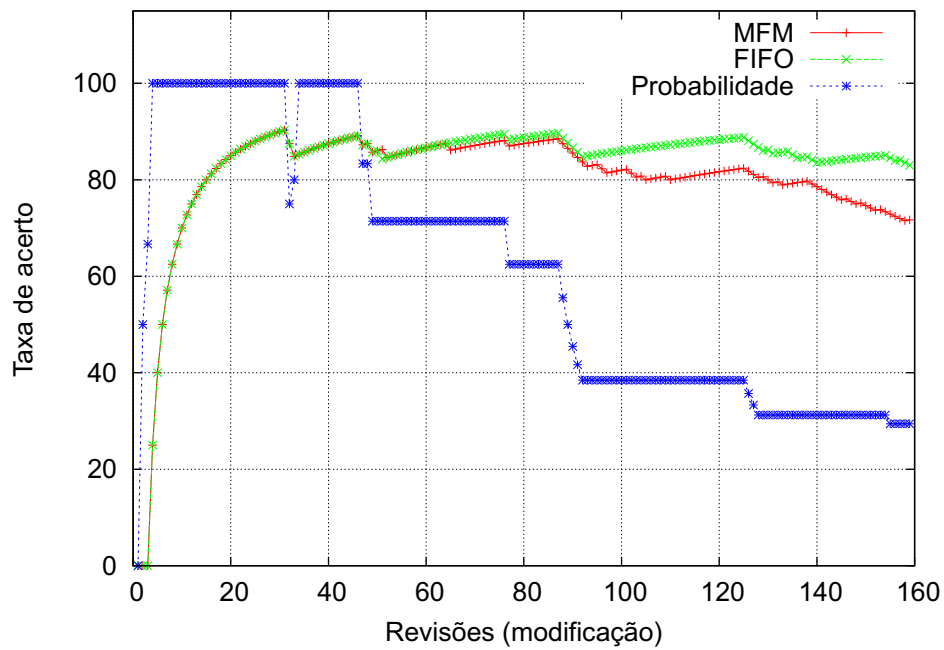


Figura 5.8. Revisões de modificação. Estratégias MFM, FIFO e probabilidade, cache de tamanho 5.

O mapa de calor ilustrado pela Figura 5.9 é um tipo de gráfico intuitivo e eficiente para apresentar os módulos indicados como mais propensos a erro. Nele os módulos do circuito são apresentados como retângulos coloridos. Neste gráfico a cor verde (cinza claro) significa baixa susceptibilidade a erro. A cor vermelho (cinza escuro) representa maior chance de erro. A área do retângulo é proporcional a quantidade de linhas de código.

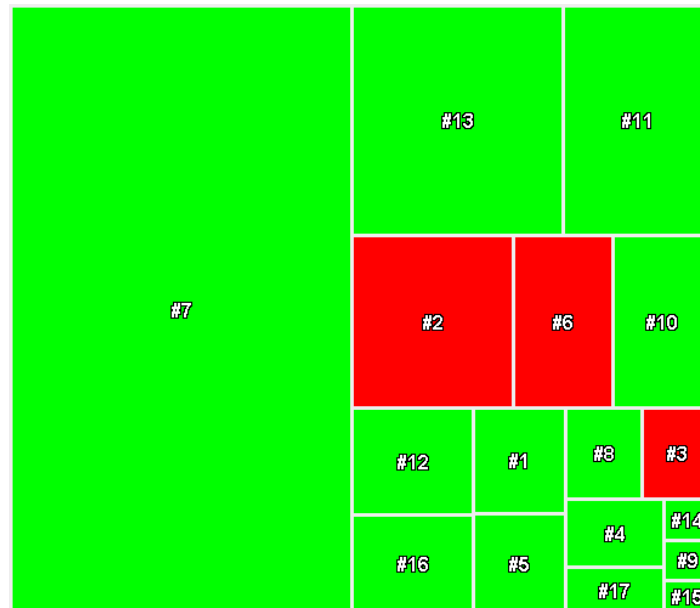


Figura 5.9. Mapa de calor.

5.4 Visualização da evolução do circuito

5.4.1 Erros no tempo

O primeiro gráfico gerado com os dados extraídos do MIPS foi o relatório de erros por quinzena, ilustrado pela Figura 5.10. As barras representam a quantidade de erros acumulados no período. A linha do tempo indica o número da quinzena. Neste gráfico foram contabilizadas apenas as modificações rotuladas como *bugfix*.

5.4.2 Métricas de hardware contrapostas com o histórico de implementação

No segundo gráfico, ilustrado pela Figura 5.11, as métricas de implementação do circuito são apresentadas na linha do tempo. Seus valores mudam na medida em que

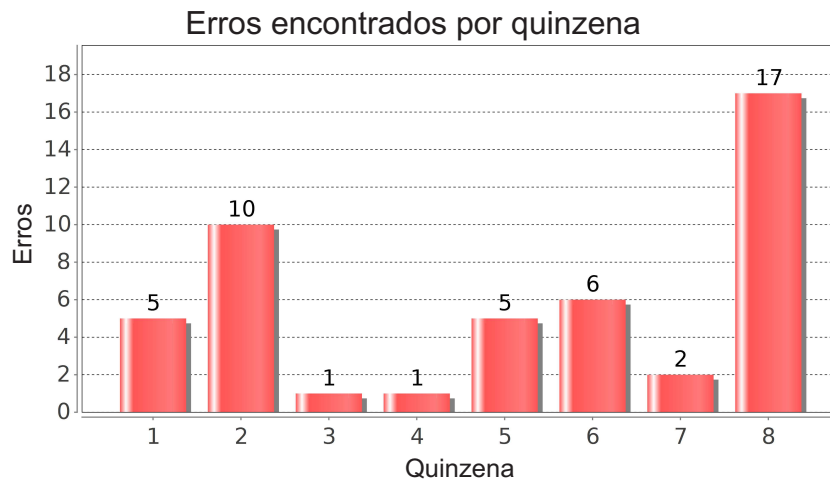


Figura 5.10. Erros encontrados por quinzena.

são feitas novas revisões. Em contraposição a estes valores, a motivação da revisão é apresentada por barras coloridas no fundo do gráfico. Cada cor representa uma motivação. Esta junção de tipos de dados enriquece o gráfico dando um contexto para cada intervalo de variação das métricas.

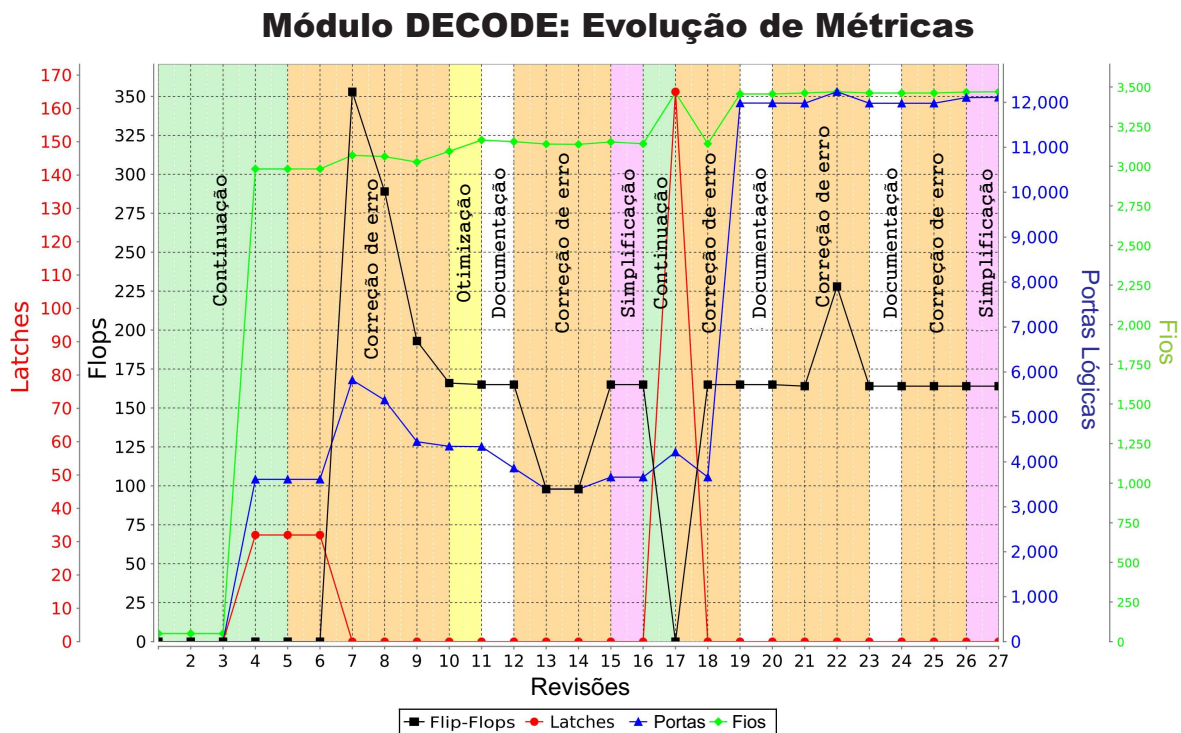


Figura 5.11. Métricas de hardware contrapostas com o histórico de implementação.

Os rótulos nas barras coloridas são a legenda do tipo da revisão. O rótulo **resume** indica que o projetista apenas enviou uma revisão de continuação do desenvolvimento sem informar nenhum erro. Os erros são rotulados como **bugfix**. Existem outros tipos de revisões como otimização (**optimization**), simplificação (**simplification**) e documentação (**documentation**).

Neste gráfico as primeiras cinco revisões são classificadas como de continuação. O aumento das métricas no intervalo 1-5 pode ser interpretado como o código RTL que os desenvolvedores foram adicionando na medida em que começaram a desenvolver o módulo. Após a revisão 5 a motivação principal das modificações foram a existência de erros que foram encontrados por inspeção ou simulação. O número de registradores (*latches* e *flops*) e portas lógicas tiveram seu valor reduzido. Na versão 10 a modificação é classificada como otimização, mas somente o número de fios (*nets*) mudou. Durante uma série de correções de erros entre 12-15, o número de *flip-flops* tiveram uma variação expressiva. Após a versão 16 todos os *flip-flops* foram sintetizados como *latches*, mas após o 18 o código foi corrigido e o número de *flip-flops* voltou ao seu valor original. Após a revisão 23 o código RTL fica mais estável, sendo necessárias menos modificações.

Uma visualização simplificada da evolução de métricas, mostrando apenas três revisões, é apresentada pela Figura 5.12. Este gráfico mostra métricas obtidas do módulo do estágio de acesso a memória no MIPS. As métricas nos cinco eixos são registradores, portas lógicas, fios, e linhas de código-fonte. As métricas são apresentadas para as revisões 6, 18 e 23, respectivamente coloridas de vermelho, azul e verde. Deste gráfico pode-se observar que a quantidade de erros acumulados aumenta na medida em que novas revisões são feitas. Os valores das métricas linhas de código e portas lógicas têm seu aumento na revisão 18, mas diminuem após a versão 23. O número de registradores não muda após a revisão 6.

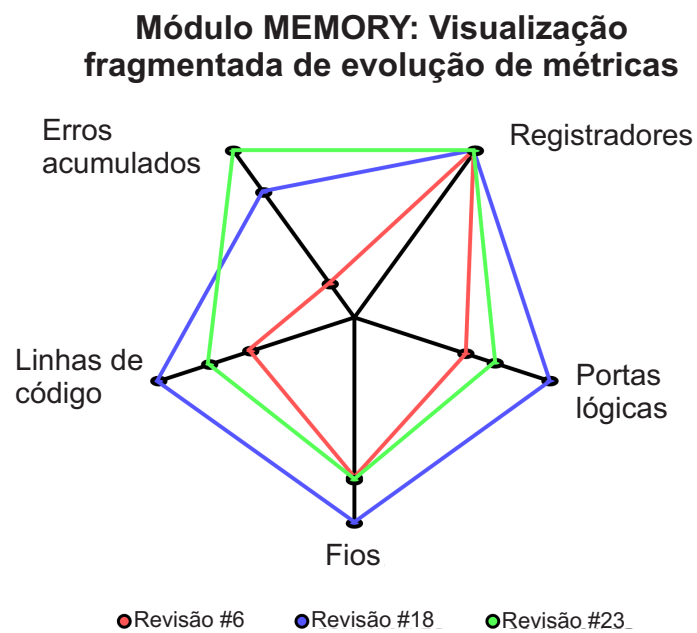


Figura 5.12. Visualização de múltiplas métricas com gráfico de radar.

Capítulo 6

Conclusões e trabalhos futuros

6.1 Conclusões

Este trabalho apresenta a ferramenta EyesOn que permite a extração de métricas do desenvolvimento de circuitos integrados. A ferramenta permitiu as primeiras pesquisas sobre a evolução de circuitos integrados e demonstrou um novo potencial de pesquisas a ser explorado. Essa ferramenta é composta por um conjunto de entidades que representam os dados coletados e seus mecanismos de extração. As entidades são representadas por classes extensíveis e que permitem a continuação do arcabouço, facilitando principalmente a adição de novas métricas e ferramentas de gerência de configuração.

As métricas são do produto e do processo de desenvolvimento de circuitos integrados, coletados no tempo, provenientes de repositórios de controle de versão e sistemas de rastreamento de erros. Neste trabalho é apresentado a coleta de dados de implementação, teste e síntese de um processador. Para que isso fosse possível, um ambiente de experimentação foi montado, provendo toda a infraestrutura necessária para desenvolver o processador e armazenar seu histórico de desenvolvimento.

Após o desenvolvimento do processador, os dados foram coletados e pré-processados. Os mesmos podem ser visualizados por meio de gráficos ou utilizados para finalidade estatística. É apresentado um mecanismo que explora a localidade temporal dos erros, e indica propensão a erros. Junto às informações coletadas são apresentados gráficos que ilustram fatos do processo de desenvolvimento e permitem a visualização da evolução das métricas.

6.2 Trabalhos futuros

Para que novos trabalhos sejam realizados seria interessante explorar outras fontes de dados e novos tipos de métricas:

- **Outras fontes de dados:** os dados de desenvolvimento representam peças importantes neste tipo de pesquisa. Devido a dificuldade de conseguir dados industriais, e de conseguir projetos de código aberto com um número significativo de revisões, neste trabalho utilizamos dados provenientes de um ambiente de desenvolvimento próprio e controlado. No entanto, no futuro espera-se que existam mais iniciativas como OpenCores [2009] e OpenSPARC [2009], e que as mesmas possam prover projetos com histórico mais detalhado de seu desenvolvimento. Seria muito interessante também observar dados obtidos de processos industriais.
- **Novos tipos de métricas:** junto ao EyesOn foram apresentadas métricas obtidas por ferramentas de código aberto e uma proprietária. Em uma abordagem mais aprofundada do que a apresentada em Schafers [1995], métricas de software como as lá propostas poderiam ser melhor analisadas. Outras métricas como as de cobertura de código-fonte, utilizadas na área de verificação e testes, apresentadas em Piziali [2004] também seriam interessantes de serem adicionadas na ferramenta. Para isso é necessário a implementação de um analisador de linguagem HDL mais completo do que o analisador léxico Verilog-Perl [Snyder, 2009], utilizado para Verilog neste trabalho.

Referências Bibliográficas

- Abran, A.; Moore, J. M.; Bourque, P. & Dupuis, R. (2008). Chapter 12 Software Measurement (Draft). <http://www2.computer.org/cms/Computer.org/SWEBOK/MeasurementKA-Draft-Feb2008.pdf>. Último acesso em 05/03/2011.
- Abran, A.; Moore, J. W.; Bourque, P. & Dupuis, R. (2004). Guide to the software engineering body of knowledge. Technical report, IEEE Computer Society.
- Ashenden, P. & Lewis, J. (2008). *The designers guide to VHDL*. Morgan Kaufmann.
- Belady, L. & Lehman, M. (1976). A model of large program development. *IBM Systems Journal*, 15(3):225--252.
- Bentley, B.; Baty, K.; Normoyle, K.; Ishii, M. & Yogev, E. (2004). Verification: What works and what doesn't. *Design Automation Conference*, 0:274–274.
- Bevan, J.; Whitehead, Jr., E. J.; Kim, S. & Godfrey, M. (2005). Facilitating software evolution research with kenyon. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 177--186, New York, NY, USA. ACM.
- Boehm, B. W.; Brown, J. R. & Lipow, M. (1976). Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pp. 592--605, Los Alamitos, CA, USA. IEEE Computer Society Press.
- Bugzilla (2011). The bugzilla guide.
- Burch, J.; Clarke, E.; Long, D.; McMillan, K. & Dill, D. (1994). Symbolic model checking for sequential circuit verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(4):401–424.

- Cardoso, T.; Nacif, J.; Fernandes, A. & Coelho, C. (2009). BugTracer: A system for integrated circuit development tracking and statistics retrieval. In *Test Workshop, 2009. LATW'09. 10th Latin American*, pp. 1--4. IEEE.
- Cavano, J. P. & McCall, J. A. (1978). A framework for the measurement of software quality. In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pp. 133--139.
- Collins-Sussman, B.; Fitzpatrick, B. & Pilato, C. (2007). *Version control with subversion*. (TBA).
- Fenton, N. & Pfleeger, S. (1997). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co. Boston, MA, USA.
- Fischer, M.; Pinzger, M. & Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, p. 23, Washington, DC, USA. IEEE Computer Society.
- Fogel, K. F. & Bar, M. (2003). *Open Source Development with CVS*. Paraglyph Press, Scottsdale, AZ, USA.
- Foster, H.; Krolnik, A. & Lacey, D. (2004). *Assertion-based design*. Springer.
- Girba, T.; Kuhn, A.; Seeberger, M. & Ducasse, S. (2005). How developers drive software evolution. *Principles of Software Evolution, Eighth International Workshop on*, pp. 113--122.
- Gousios, G. (2009). *Tools and Methods for Large Scale Software Engineering Research*. PhD thesis, Athens University of Economics and Business.
- Greevy, O.; Ducasse, S. & Girba, T. (2006). Analyzing software evolution through feature views: Research articles. *J. Softw. Maint. Evol.*, 18(6):425--456.
- Halstead, M. (1977). *Elements of software science*. Elsevier New York.
- Hassan, A. & Holt, R. (2005). The top ten list: Dynamic fault prediction. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 263--272. IEEE.
- Hodges, D.; Jackson, H. & Saleh, R. (2003). *Analysis and design of digital integrated circuits*. McGraw-Hill, Inc. New York, NY, USA.

- Jasper Design Automation, I. (2010). *JasperGold Verification System and JasperCore Command Reference Manual*.
- Jørgensen, A. H. (1980). A methodology for measuring the readability and modifiability of computer programs. *BIT Numerical Mathematics*, 20:393–405. 10.1007/BF01933633.
- Kan, S. (2002). *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.
- Karanam, M. & Akepogu, A. (2009). Model-Driven Software Evolution: The Multiple Views. *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 1.
- Langelier, G.; Sahraoui, H. & Poulin, P. (2005). Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 214--223. ACM.
- Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060--1076.
- Lehman, M.; Ramil, J.; Wernick, P.; Perry, D. & Turski, W. (1997). Metrics and laws of software evolution-the nineties view. In *metrics*, p. 20. Published by the IEEE Computer Society.
- Maurer, S. & Sviridenko, A. (2009). The hdl complexity tool. <http://hct.sourceforge.net/>. Último acesso em 06/03/2011.
- McCabe, T. (1976). A complexity measure. *IEEE Transactions on software Engineering*, pp. 308--320.
- Meeuws, R. J.; Yankova, Y. D.; Bertels, K. L. M.; Gaydadjiev, G. N. & Vassiliadis, S. (2007). A quantitative prediction model for hardware/software partitioning. In *Proceedings of 17th International Conference on Field Programmable Logic and Applications (FPL'07)*.
- Moore, G. (1965). Cramming more components onto integrated circuits. *Electronics*, pp. 114–117.
- Mullane, B. & MacNamee, C. (2008). From behavioral to rtl design flow in systemc.

- Nacif, J. (2011). *UMA METODOLOGIA PARA IDENTIFICAÇÃO DE MÓDULOS DE CIRCUITOS INTEGRADOS PROPENSOS A ERROS*. PhD thesis, Universidade Federal de Minas Gerais.
- Nacif, J. A.; Silva, T.; Tavares, A. I.; Fernandes, A. O. & Coelho, C. N. (2008). Efficient allocation of verification resources using revision history information. In *DDECS '08: Proceedings of the 2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pp. 1--5, Washington, DC, USA. IEEE Computer Society.
- Nejmeh, B. (1988). NPATH: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188--200.
- OpenCores (2009). Opencores. <http://www.opencores.org>.
- OpenSPARC (2009). Opensparc. <http://www.opensparc.net/>.
- Pinzger, M.; Gall, H.; Fischer, M. & Lanza, M. (2005). Visualizing multiple evolution metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, pp. 67--75. ACM.
- Piziali, A. (2004). *Functional verification coverage measurement and analysis*. Kluwer Academic Publishers.
- Rabaey, J.; Chandrakasan, A. & Nikolić, B. (2003). *Digital Integrated Circuits: A Design Perspective*. Prentice hall New Jersey.
- Robbes, R. (2007). Mining a change-based software repository. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, p. 15, Washington, DC, USA. IEEE Computer Society.
- Schafers, M. (1995). Measuring the complexity of hdl models. In *ASIC Conference and Exhibit, 1995., Proceedings of the Eighth Annual IEEE International*, pp. 113--116.
- Snyder, W. (2009). Verilog-perl. <http://www.veripool.org/wiki/verilog-perl>.
- Vaumorin, E. & Romanteau, T. (2011). Developing a reusable ip platform within a system-on-chip design framework targeted towards an academic r&d environment.
- Voinea, L. & Telea, A. (2007). Visual data mining and analysis of software repositories. *Computers & Graphics*, 31(3):410--428.
- Wu, X. (2003). Visualization of version control information.

- Zimmermann, T. & Zeller, A. (2005). When do changes induce fixes? on fridays. In *Proc. International Workshop on Mining Software Repositories (MSR), St.*
- Zuse, H. (1990). *Software complexity: measures and methods*. Walter de Gruyter & Co.

Apêndice A

Publicações durante mestrado

1. A Cache Based Algorithm to Predict HDL Modules Faults. In: Latin American Test Workshop, 2011, Porto de Galinhas. Proceedings of the 12th Latin American Test Workshop, 2011.
2. Tracking Hardware Evolution. International Symposium on Quality Electronic Design, 2011, Santa Clara. Proceedings of ISQED, 2011.
3. Visualizing HDL Code Evolution. In: Chip in Sampa Student Forum, 2010, São Paulo. Proceedings of the Chip in Sampa Student Forum, 2010.