UMA METODOLOGIA PARA IDENTIFICAÇÃO DE MÓDULOS DE CIRCUITOS INTEGRADOS PROPENSOS A ERROS

JOSÉ AUGUSTO MIRANDA NACIF

UMA METODOLOGIA PARA IDENTIFICAÇÃO DE MÓDULOS DE CIRCUITOS INTEGRADOS PROPENSOS A ERROS

Tese 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 Doutor em Ciência da Computação.

Orientador: Antônio Otávio Fernandes

Belo Horizonte

01 de março de 2011

© 2011, José Augusto Miranda Nacif.
Todos os direitos reservados.

Nacif, José Augusto Miranda
N124m Uma Metodologia para Identificação de Módulos de Circuitos Integrados Propensos a Erros / José Augusto Miranda Nacif. — Belo Horizonte, 2011
xxiii, 120 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais
Orientador: Antônio Otávio Fernandes

1. Computação - Teses. 2. Circuitos digitais - Teses. I. 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

Uma metodologia para identificação de módulos de circuitos integrados propensos a erros

JOSÉ AUGUSTO MIRANDA NACIF

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

PROF. RICARDO DOS SANTOS FERREIRA
Departamento de Informática - UFV

PROF. WANG JIANG CHAU
Departamento de Engenharia de Sistemas Eletrônicos - USP

PROF. CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR
Departamento de Ciência da Computação - UFMG

PROF. DIÓGENES CECÍLIO DA SILVA JÚNIOR
Departamento de Engenharia Elétrica - UFMG

Belo Horizonte, 01 de março de 2011.

Aos meus pais,

Por me ensinarem a valorizar as coisas que realmente importam.

Agradecimentos

A Deus, por me permitir vencer mais essa etapa.

À minha família, principalmente pais e irmãos, pelo apoio incondicional nos momentos mais difíceis dessa caminhada.

À minha namorada Lorena, por me apoiar, compreendendo as ausências e sacrifícios que se fizeram necessários para a conclusão deste trabalho.

Ao meu orientador, prof. Antônio Otávio Fernandes, por compartilhar sua sabedoria e amizade sempre me ouvindo com paciência e proporcionando todos os recursos para desenvolvimento do trabalho.

Às pessoas que, ao longo desses anos, contribuíram tecnicamente para que esta tese de doutorado fosse concluída. Gostaria de agradecer, em especial a: Andréa Iabrudi, pelas diversas discussões técnicas e pelo incentivo final para conclusão deste trabalho; Thiago Souza, Thiago Cardoso, André Souza e Abner pela ferramenta EyesOn; Thiago Souza e Celina, enquanto monitores da disciplina Organização de Computadores II, pelo apoio na condução dos experimentos; Alex Borges pela colaboração na redação de diversos trabalhos; Prof. Marcelo Azevedo, do Departamento de Estatística da UFMG, pela ajuda na interpretação dos resultados.

Aos funcionários do Departamento de Ciência da Computação da UFMG Maria Aparecida Lages, Renata Viana Rocha, Rosencler Oliveira, Sheila Carneiro e Sônia Borges Melo, pela ajuda na resolução dos problemas do dia-a-dia.

Aos professores do Departamento de Ciência da Computação da UFMG Claudionor Nunes Coelho Jr., Gisele Pappa, Luiz Filipe Vieira, Marcos Augusto Vieira, Newton Vieira, Omar Paranaíba e Sérgio Campos, pelas sugestões e discussões técnicas que contribuíram para o aprimoramento deste trabalho.

Aos colegas da Universidade Federal de Viçosa - Campus de Florestal, do Laboratório de Engenharia de Computadores da UFMG, da Física e colegas de graduação, pelos momentos agradáveis de convívio e confraternização.

Aos alunos voluntários dos cursos de Arquitetura de Computadores e Organização de Computadores II por participarem dos experimentos que geraram os dados utilizados neste trabalho.

Aos membros das bancas examinadoras de qualificação e defesa de tese, professores Claudionor Nunes Coelho Jr., Diógenes da Silva Jr., Luigi Carro, Ricardo Santos Ferreira e Wang Jiang Chau, pelas críticas e sugestões.

À Universidade Federal de Viçosa - Campus de Florestal, pela liberação parcial das minhas atividades docentes durante um ano e dois meses.

Ao CNPq e à FAPEMIG pelo suporte financeiro, sob os processos #PNM-141201/2005-3, #CEX-1485-06 e #APQ-02217-10.

"You know you will never get to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the climb."
(Winston Churchill)

Resumo

O processo de verificação de circuitos integrados industriais se torna mais desafiador a cada dia. As metodologias de verificação atuais não são capazes de garantir que todos os erros de um circuito integrado sejam identificados e corrigidos antes da fabricação. Como não é possível checar todos os estados de circuitos integrados complexos, a equipe de verificação deve definir níveis de cobertura para cada módulo do circuito integrado. Se a cobertura de módulos mais propensos a erros é priorizada, consegue-se identificar um maior número de erros mais rapidamente.

A principal contribuição deste trabalho é propor uma metodologia para construir modelos que indiquem quais módulos de um circuito integrado são propensos a conter erros não identificados. O modelo é construído utilizando métricas de complexidade e de histórico de desenvolvimento, extraídas de sistemas de controle de versão e de sistemas de rastreamento de erros. A metodologia proposta permite definir, para um projeto específico, quais métricas têm correlação com o número de erros. Assim, a alocação de recursos de verificação é realizada mais eficientemente e pode-se definir módulos que devem ser submetidos a um processo de verificação mais rigoroso.

A metodologia proposta foi validada a partir de um ambiente experimental composto por métricas, dados e ferramentas. As métricas estudadas são extraídas por ferramentas comerciais e de código aberto. Os dados utilizados são repositórios de projetos de processadores descritos em linguagem de descrição de hardware. Foram desenvolvidas duas ferramentas para automatizar a extração de métricas de complexidade e de histórico de desenvolvimento. A primeira, o BugReporter, é uma ferramenta que auxilia os usuários a relatar erros identificados no projeto,

utilizando palavras reservadas definidas pela Bug Language. A segunda, EyesOn, automatiza o processo de extração, armazenamento e visualização das métricas de complexidade e de histórico de desenvolvimento.

São apresentados resultados de utilização da metodologia proposta em dois processadores: MIPS e OpenSPARC. Para cada um dos processadores são construídos modelos lineares, de Poisson, binomiais negativos e de regressão logística. Para o processador MIPS, o modelo que apresenta o melhor desempenho é o linear. Já no OpenSPARC, o melhor desempenho é alcançado pelo modelo binomial negativo. Em ambos os casos os modelos que apresentam o melhor desempenho utilizam cinco métricas. Finalmente, os resultados de utilização de métricas de desenvolvimento no processador MIPS são discutidos. É proposto um algoritmo que utiliza o número de alterações para construir, dinamicamente, uma lista de módulos propensos a erros. Ao longo do desenvolvimento dos módulos, a lista contém, em 80% das vezes, os módulos relatados com erros. Assim, os resultados alcançados pela metodologia proposta abrem caminho para que novos estudos sejam realizados com outros conjuntos de dados e métricas.

Abstract

Verifying large industrial designs is getting harder each day. The current verification methodologies can not guarantee bug free designs. Considering that is not possible to check all states of complex designs, the verification team should define coverage levels for each integrated circuit module. If the coverage of error-prone modules is prioritized, it is possible to identify more bugs in less time.

The novelty of this work is to propose a methodology that is able to build a model which points the hardware modules that are most likely to have undetected design bugs. The model is built using revision history information and complexity metrics extracted from concurrent versioning systems and bug tracking systems. The proposed methodology allows the identification, for a specific project, of which metrics are correlated with bugs. Thus, we are able to allocate verification resources more efficiently and define high risk modules to be submitted to a more rigorous verification process.

The proposed methodology is validated in an experimental environment composed by metrics, data, and tools. The studied metrics are extracted by comercial and open source tools. The data used in this work are repositories of processors described in hardware description languages. In order to automate the complexity and design history metrics extraction, two tools were developed. The first, BugReporter, is a tool used to help users to report project bugs, using keywords defined by BugLanguage. The second, EyesOn, automates the extraction, storage and visualization of complexity and history metrics.

We present results from the use of the proposed methodology in two processors: MIPS and OpenSPARC. For each of these processors, linear, Poisson, negative binomial, and logistic models are built. In the MIPS processor, linear

model presents the best results. In the OpenSPARC project, the best performance is reached by negative binomial model. In both cases, the best models use five metrics. We also discuss results from the use of history metrics in the MIPS processor. We propose an algorithm that uses the number of modifications to dynamically build a list of error-prone modules. During the modules development the list stores, in 80% of the time, the modules that had reported errors. Thus, the results achieved by the proposed methodology could be extended with new studies using other data sets and metrics.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Verificação de circuitos integrados é o processo utilizado para garantir que o comportamento especificado foi preservado na implementação (Bergeron [2003]). Este processo é computacionalmente complexo, de ordem $2^{(n+m)}$, onde n é o número de entradas e m, o número de estados do circuito integrado. A etapa de verificação é o gargalo no ciclo de desenvolvimento de um circuito integrado, sendo responsável por aproximadamente 60% a 80% do tempo total de desenvolvimento (Foster et al. [2004]; Bentley et al. [2004]; Manners [2008]; ITRS [2009]). O problema de verificação se torna, a cada dia, mais crítico devido à combinação dos seguintes fatores (Carter & Hemmady [2007]):

- Aumento no tamanho dos projetos;
- Diminuição das janelas de oportunidade para os novos circuitos integrados serem lançados no mercado;
- Aumento dos prejuízos causados por erros.

Devido à utilização maciça de multiprocessamento, os circuitos integrados modernos têm alcançado novos níveis de integração. Atualmente, a maioria dos processadores de uso geral utilizam múltiplos núcleos (*cores*), vários níveis memória (cuja coerência deve ser mantida), redes de interconexão *onchip* e controladores de memória e E/S. Novas tecnologias como virtualização, gerenciamento dinâmico de

energia e extensões 64 bits também contribuem para o aumento de complexidade dos circuitos integrados atuais.

A combinação dos fatores de aumento de complexidade e diminuição de tempo de desenvolvimento leva a situações indesejáveis, tais como a comercialização de circuitos integrados contendo erros que se manifestam apenas depois de vendidos. O exemplo clássico é o erro de divisão de ponto flutuante do processador Pentium da Intel (Beizer [1995]). Apesar de não afetar diretamente a maioria dos usuários, a Intel trocou todos os processadores defeituosos gratuitamente, tendo um prejuízo de 475 milhões de dólares. Recentemente, outro erro foi descoberto pela Intel em seus *chipsets* da série 6 (Intel [2011]). O problema consiste na degradação gradual de dispositivos ligados à interface SATA e pode resultar no não reconhecimento desses dispositivos. Para corrigir o erro foi necessário fabricar novos circuitos integrados. Estima-se que a Intel terá um prejuízo de 700 milhões de dólares devido a esse erro. A Figura 1.1 apresenta exemplos de erros de projeto de processadores modernos da Intel divulgados na forma de erratas. Felizmente, a maior parte desses erros pode ser resolvida por software.

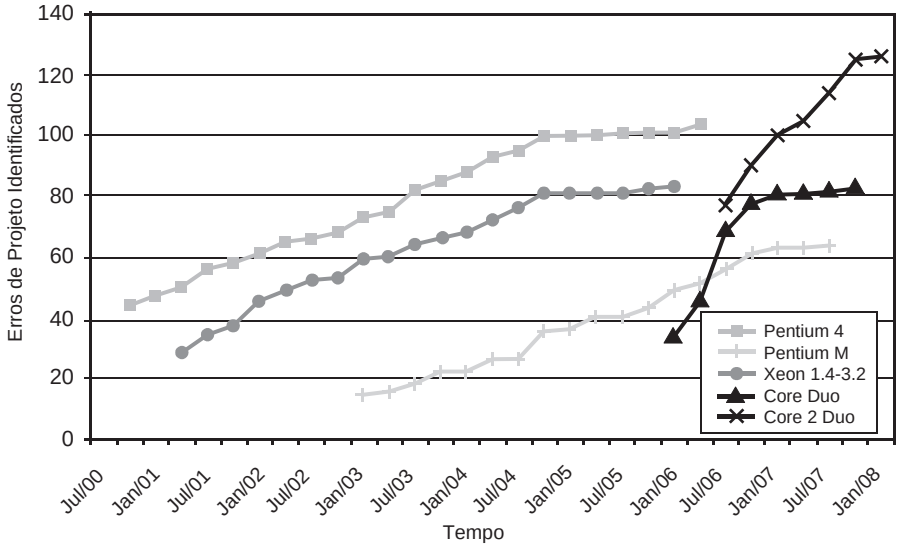

Figura 1.1. Erros de processadores modernos da Intel divulgados na forma de erratas. Adaptada de Constantinides et al. [2008].

Quando um erro é identificado e corrigido antes da fabricação, o custo de correção é baixo. Os prejuízos financeiros decorrentes da identificação de erros após a fabricação do circuito integrado são maiores (Wiemann [2007]), conforme ilustra a Figura 1.2. Quando um erro é identificado na fase de testes em silício o custo pode chegar a centenas de milhares de dólares, pois o circuito integrado precisa ser fabricado novamente, além do atraso no cronograma de entrada no mercado. Finalmente, o pior cenário é quando um consumidor identifica o erro. Muitas vezes, nestes casos, o fabricante tem que substituir o circuito integrado defeituoso e a imagem da empresa pode ser atingida permanentemente.

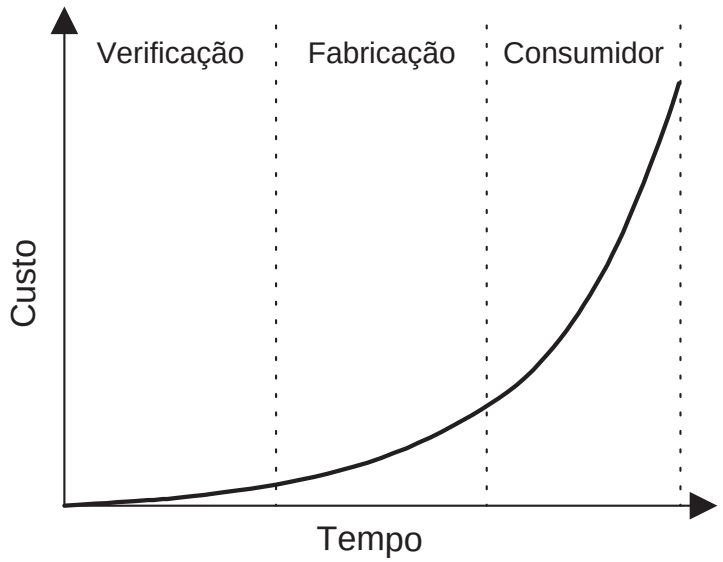

Figura 1.2. Custo decorrente da identificação de erros de projeto em circuitos integrados. Adaptada de Wile et al. [2005].

Durante o desenvolvimento de um circuito integrado, quanto mais eficiente é a etapa de verificação, menores são os custos e o tempo de desenvolvimento. Este comportamento pode ser observado na Figura 1.3, na qual três curvas de identificação de erros são apresentadas. Na curva mais longa, o processo de verificação se mostra ineficiente e erros de projeto continuam a ser identificados após a fabricação do circuito integrado. As duas curvas restantes representam processos de verificação mais eficientes, nos quais os erros de projeto são identificados mais

rapidamente, reduzindo o tempo e os custos de desenvolvimento.

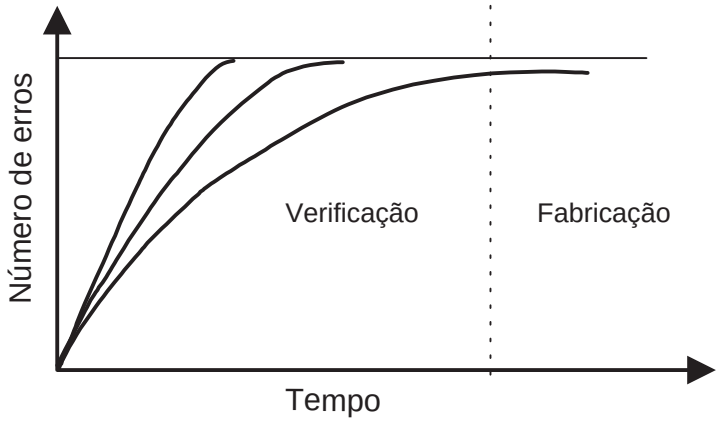

Figura 1.3. Curvas de identificação de erros de projeto ao longo do processo de desenvolvimento. Adaptada de Wile et al. [2005].

1.1 Objetivo

O objetivo deste trabalho é apresentar uma metodologia para definir quais módulos de um circuito integrado, implementado em linguagem de descrição de hardware, são propensos a conterem erros. Para tanto, são extraídas métricas de cada módulo durante o processo de desenvolvimento do circuito integrado. As métricas extraídas dos módulos são selecionadas de forma a identificar quais se correlacionam com erros de projeto. O subconjunto de métricas resultante da seleção é utilizado para construir os modelos que são específicos para cada projeto, pois, cada projeto tem características distintas em relação a: processo de desenvolvimento, equipe, ferramentas utilizadas e cronograma. As duas principais classes de métricas estudadas são as métricas de produto e as métricas de processo.

As métricas de produto medem a complexidade dos módulos sob diversos aspectos: tamanho, documentação, estruturas de controle, hierarquia, interface e sincronização. Estas métricas são largamente utilizadas na identificação de erros de projeto em engenharia de software (Zimmermann et al. [2008]). Dentro do contexto de módulos implementados em linguagem de descrição de hardware,

exemplos destas métricas são os números de: sinais de E/S, flip-flops, portas lógicas, linhas de comentários, sinais de sincronização. As métricas de processo são extraídas de ferramentas de controle de versão (*revision control*) e de rastreamento de erros (*bug tracking systems*). Hoje em dia, estas ferramentas são utilizadas em grande parte das empresas que desenvolvem circuitos integrados. Exemplos destas métricas são os números de: linhas modificadas, erros de projeto identificados e de desenvolvedores.

Atualmente, o nível de reuso de blocos de propriedade intelectual (IP - *Intellectual Property*) de circuitos integrados tem crescido. Muitos blocos IP são desenvolvidos durante vários anos e suas vidas úteis chegam a uma década (Stapleton & Tobin [2009]). Esses blocos IP podem passar por diversas modificações ao longo desse processo. Essas modificações muitas vezes são realizadas para adaptar o bloco IP a novas tecnologias de fabricação (transição de 65 nm para 45 nm, por exemplo) ou atualizar o bloco IP a especificações mais recentes (USB 2.0 para USB 3.0, por exemplo). Assim, os sistemas de controle de versão e gerenciamento de erros possuem várias métricas de processo que descrevem o histórico de desenvolvimento desses blocos IP e podem ser utilizados para identificar módulos propensos a erros.

1.2 Aplicações

O conhecimento de que um conjunto de módulos é propenso a conter erros de projeto pode ser usado de diversas formas ao longo do processo de desenvolvimento do circuito integrado. A seguir, são apresentados exemplos de cenários de utilização da metodologia proposta:

1. Alocação de recursos de verificação: Os principais recursos a serem alocados durante o processo de verificação são desenvolvedores e tempo de processamento. Alocar mais recursos em módulos mais propensos a erros traz como benefício o aumento da probabilidade de identificação e correção precoce de erros, tornando o processo de verificação mais eficiente.

2. Definição de módulos que devem passar por processos de verificação mais rigorosos, por exemplo, geração de mutantes (Hampton & Petithomme

[2007]), verificação formal ou emulação em FPGA (Bayazit & Malik [2005]; Reorda et al. [2006]).

3. Inclusão de arquiteturas tolerantes a falhas: Recentemente, vários trabalhos propõem novas arquiteturas para a detecção e recuperação de erros de projeto (hui Chang et al. [2009]; Wagner et al. [2008]; Meixner et al. [2007]; Cardoso et al. [2008]). Muitas destas arquiteturas são baseadas na inclusão e monitoramento *online* de propriedades do circuito integrado. Nesse caso, deve-se selecionar os módulos propensos a conterem erros, pois toda arquitetura tolerante a falhas possui um custo associado em termos de área e/ou velocidade.

4. Definição de ordem de verificação de módulos. A maioria dos fluxos de verificação industriais utilizam asserções para descrever propriedades do circuito integrado (Foster et al. [2004]). No início do processo de verificação, diversas asserções são violadas. Nesse caso, seria priorizada a resolução violações de asserções que estivessem localizadas em módulos propensos a erros.

1.3 Contribuições

As principais contribuições deste trabalho são apresentadas abaixo:

1. Definição de uma metodologia para construção de modelos de propensão a erros de módulos de circuitos integrados.

2. Demonstração da correlação entre métricas de complexidade e propensão a erros de módulos de circuitos integrados.

3. Utilização de métricas de processo para identificar módulos propensos a erros ao longo do desenvolvimento do projeto (Nacif et al. [2008, 2011a]).

4. Análise e acompanhamento de informações disponíveis em sistemas de controle de versão de projetos de circuitos integrados. O termo evolução de hardware (*hardware evolution*) é proposto para denominar esse processo (Nacif et al. [2011b]).

5. Definição de uma linguagem para descrição de erros de projeto, incluída nos comentários das modificações realizadas. Essa linguagem é denominada Bug Language (Cardoso et al. [2009]).

Foram desenvolvidas duas ferramentas para automatizar a extração de métricas de produto e de processo. A primeira, o BugReporter, é uma ferramenta que auxilia os usuários a relatar erros identificados no projeto, utilizando palavras reservadas definidas pela Bug Language. As informações dos erros são incluídas nos comentários dos `commits` sendo, dessa forma, facilmente processadas. A segunda ferramenta, EyesOn, automatiza o processo de extração, armazenamento e visualização das métricas de produto e processo. A ferramenta EyesOn oferece suporte para a inclusão de novas métricas e tipos de visualização.

1.4 Organização

O Capítulo 2 apresenta os trabalhos relacionados. Inicialmente é discutido o estado da arte da pesquisa em extração de métricas de sistemas de controle de versão na área de engenharia de software, enfatizando as técnicas empregadas na construção de modelos de predição de erros. Em seguida, são apresentados os trabalhos que utilizam métricas de complexidade extraídas de projetos de circuitos integrados. O Capítulo 3 apresenta a metodologia de construção do modelo de predição de erros de projeto. No Capítulo 4 é apresentado o ambiente experimental construído para demonstrar a utilização da metodologia proposta. O ambiente experimental é composto por métricas, dados e ferramentas. No Capítulo 5 são discutidos os resultados alcançados em dois estudos de caso com os processadores MIPS e OpenSPARC. No Capítulo 6 são apresentadas as conclusões e propostas de trabalhos futuros. O Apêndice A apresenta os artigos publicados desde o início do curso de doutorado. Finalmente, o Apêndice B exemplifica algumas etapas do desenvolvimento do processador MIPS utilizado no Capítulo 5.

Capítulo 2

Trabalhos Relacionados

Neste capítulo são descritos os principais tópicos relacionados à identificação e predição de erros de projeto. Apesar de este trabalho tratar de projetos de circuitos digitais, grande parte das técnicas e conceitos apresentados são legados da área de engenharia de software, pois pouca literatura existe acerca de sua aplicação no desenvolvimento de hardware. O problema de identificação e predição de erros de projeto envolve, invariavelmente, observação empírica de sistemas reais ou realização de experimentos. Os métodos empíricos utilizados em engenharia de software envolvem observação de fenômenos, formulação de hipóteses, e coleta de dados ou realização de experimentos para validar ou invalidar as hipóteses. Uma prova de hipótese baseada em métodos empíricos nunca é conclusiva, mas apenas um experimento que contradiz uma hipótese é suficiente para invalidá-la. Segundo Glass et al. [2002]; Sjoberg et al. [2007]; Easterbrook et al. [2008]; Wohlin & Wesslen [2000], os principais métodos de pesquisa utilizados em engenharia de software empírica são:

- Experimentos: estudos empíricos que investiga as relações entre variáveis. Especificamente, os experimentos são caracterizados pela medição dos efeitos da manipulação de uma variável em outra variável e pelo fato que as tarefas são aleatoriamente assinaladas para os participantes. Um experimento implica em controle sobre as condições nas quais o estudo é conduzido e também controle sobre as variáveis independentes que estão sendo testadas.

- Questionários ou entrevistas: conduzidos em uma população específica. Úteis para estudar um grande número de variáveis utilizando uma amostra significativa e análise estatística rigorosa.

- Estudos de casos: investigam um conjunto de fenômenos dentro do contexto que estes ocorreram. No contexto de engenharia de software, isso geralmente significa o estudo do comportamento de um conjunto de variáveis independentes com o objetivo de se construir modelos que descrevem as descobertas em grandes conjuntos de dados.

Os principais componentes dos estudos empíricos de engenharia de software são: **Métricas**, **Dados** e **Ferramentas**. Normalmente os pesquisadores têm acesso limitado a fontes de dados, ferramentas e paradigmas de extração. A contribuição de um estudo empírico não é uma nova forma de extração de dados ou métrica, mas a validação de um modelo em conjuntos de dados existentes. Inicialmente, são apresentadas importantes definições. As próximas Seções descrevem dados, métricas e ferramentas. Em seguida, serão discutidos os principais trabalhos na área de identificação e predição de erros de projeto em software. Finalmente, serão apresentados trabalhos que utilizam métricas de projetos de circuitos integrados.

2.1 Definições

Os termos falha, erro e falta são utilizados de acordo com as definições propostas por Avizienis et al. [2004]:

- Falha (*failure*): Uma falha acontece quando um sistema não se comporta conforme sua especificação;

- Erro (*error*): É a parte do sistema que exibe o comportamento incorreto. Alguns trabalhos na área de verificação definem *bug*, defeito (*defect*) e *errata* como sinônimos de erro (Constantinides et al. [2008]);

- Falta (*fault*): É a causa do erro.

O foco deste trabalho é identificar quais módulos de um projeto de um circuito integrado são mais propensos a conter erros de projeto (*design errors*). Erros de projeto podem ser definidos como erros inseridos pelos desenvolvedores, não intencionalmente, durante a implementação, em linguagem de descrição de hardware, do circuito integrado.

2.2 Métricas

Métricas são medições de propriedades ou especificações do software. Várias características de software e de projetos de software podem ser medidas como, por exemplo, tamanho, complexidade e qualidade. Segundo Fenton & Pfleeger [1998] e Kan [2003], as métricas de software podem ser divididas em:

- Métricas de produto: usadas para quantificar várias características do software. São divididas em duas categorias (Fenton & Pfleeger [1998]):
  - Métricas de atributos internos: relativas ao tamanho e estrutura do produto;
  - Métricas de atributos externos: medem o impacto dos atributos internos do software em sua qualidade.
- Métricas de processo: se referem às atividades de desenvolvimento e processos de software. Medições de defeitos por hora de teste, tempo, tamanho da equipe recaem nessa categoria.
- Métricas de projeto: se referem a qualquer entrada do processo de desenvolvimento como, por exemplo, pessoas e métodos.

Nas próximas Seções são apresentadas as métricas de produto e processo mais utilizadas em estudos de engenharia de software. As métricas de projeto não serão discutidas pois não se incluem no escopo deste trabalho.

2.2.1 Métricas de Produto

As métricas de produto medem atributos internos do software que está sendo estudado. Caso consideradas isoladamente, as métricas de produto não são capazes de revelar informações relevantes. Por esse motivo, a maioria dos trabalhos utiliza as métricas de produto para medir a eficácia do processo de desenvolvimento. As principais métricas de produto são: tamanho, complexidade e qualidade.

2.2.1.1 Tamanho

Tamanho é um atributo fundamental em software. Medições de tamanho são importantes para comparar os projetos, avaliar o esforço de desenvolvimento ou a produtividade da equipe. O tamanho do projeto pode ser correlacionado com a contagem de erros, falhas de segurança ou evolução de métricas. Normalmente, as métricas de tamanho medem palavras, linhas ou funcionalidades do software.

A métrica mais simples de tamanho do software é o número de linhas de código (LC, *Lines of Code*) dos arquivos fonte dos módulos do software. Existem dois tipos de métricas de linhas de código: físicas e lógicas. As LC físicas são o número total de linhas de um módulo de software enquanto, as LC lógicas, são compostas pelo número de linhas de comandos. A métrica LC é simplista e não leva em conta comentários, notas de licença e formatação de código e, mais importante, as diferenças entre as diversas linguagens de programação. Apesar destes pontos falhos, a LC é largamente utilizada em trabalhos, especialmente na pesquisa de evolução de software.

Halstead [1977] estendeu a noção de tamanho de software para incluir outros elementos do código como número de operandos e operadores. Para Halstead, o $tamanho(N)$ de um programa é igual ao número total de $operadores(N_1)$ e $operandos(N_2)$, enquanto o volume é igual a $N \cdot log_2(\mu_1 + \mu_2)$, no qual μ_1 e μ_2 é o número de operadores distintos e operandos, respectivamente. O volume do programa é essencialmente uma medição do tamanho de uma implementação sem ser baseado em formatação de código. Apesar de diversas críticas na literatura (Coleman et al. [1994a]; Fenton & Pfleeger [1998]; Menzies et al. [2004]; Weyuker [1988]), as métricas de tamanho de Halstead são populares na avaliação de manutenibilidade de software.

As métricas anteriores contam tamanho físico: linhas, operadores e operandos. Alguns pesquisadores (Albrecht & Gaffney [1983]; Chatman [1995]), defendem que este tipo de métrica é ineficaz, pois não existe uma relação clara entre essas métricas e as funcionalidades que o software executa. De forma a tentar medir o tamanho do programa em termos de funcionalidade, Albrecht [1979] desenvolveu uma metodologia para estimar o número de funcionalidades do software em termos de dados gerados e utilizados. Mais especificamente, são quantificados pontos de função como uma soma ponderada do número de entradas, saídas, arquivos, e consultas recebidas ou geradas pelo software. Análise de pontos de função tem sido utilizada para planejamento e verificação de produtividade, embora seja uma metodologia difícil de aplicar e automatizar.

2.2.1.2 Complexidade

Uma das primeiras e mais utilizadas métricas de complexidade é complexidade ciclomática proposta por McCabe [1976]. Essa métrica examina o grafo de fluxo de controle de uma função e calcula a medida de sua complexidade, enumerando o número de possíveis caminhos de execução.

As métricas de complexidade estruturais levam em consideração as interações entre os módulos em um sistema. A métrica de fluxo de informações, proposta por Henry & Kafura [1981], utiliza o número de módulos que chamam um determinado módulo (*fan-in*) e o número de módulos chamados por um determinado módulo (*fan-out*). A métrica de complexidade é obtida multiplicando-se estes dois fatores. Quanto menor o valor, mais bem estruturado está o sistema.

Baseado em várias abordagens para medir a complexidade do software, Card & Glass [1990] propuseram um modelo de complexidade baseado na soma da complexidade estrutural e de dados do software. A complexidade estrutural pode ser definida como o *fan-out* médio de todos os módulos do sistema e, a complexidade de dados, é proporcional ao número de variáveis de E/S e inversamente proporcional ao *fan-out* do módulo. Essa métrica foi utilizada para modelar a curva de erros do projeto.

As métricas apresentadas até agora são definidas para linguagens de programação estruturada. Um conjunto de métricas baseado em conceitos de orientação

a objetos foi proposto por Chidamber & Kemerer [1994]. Essas métricas são apresentadas na Tabela 2.1. Chidamber e Kemerer estudaram a propensão a erros em classes em oito projetos de estudantes. Os resultados demonstraram que as métricas Métodos Ponderados por Classe, Acoplamento Entre Objetos, Profundidade da Árvore de Herança, Número de Filhos e Resposta por Classe são correlacionadas aos erros, ao contrário da métrica Falta de Coesão de Métodos.

Tabela 2.1. Métricas de complexidade orientadas a objetos (Chidamber e Keremer).

Métrica	**Descrição**
Métodos Ponderados por Classe (MPC)	*Weighted Methods per Class*, é o número de métodos em cada classe do software.
Acoplamento Entre Objetos (AEO)	*Coupling Between Objects*, é o número de classes no qual uma determinada classe está acoplada.
Profundidade da Árvore de Herança (PAH)	*Depth of Inheritance Tree* é o maior caminho de herança em uma determinada classe.
Número de Filhos (NF)	*Number of Children* é o número de descendentes diretos de cada classe.
Resposta por Classe (RC)	*Response For a Class* é o número de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto desta classe.
Falta de Coesão de Métodos (FCM)	*Lack of Cohesion of Methods* é o número de pares de métodos sem variáveis compartilhadas, menos o número de pares de métodos com variáveis compartilhadas, caso o valor da subtração seja negativo, esta métrica tem seu valor alterado para zero.

2.2.1.3 Qualidade

A qualidade é uma métrica utilizada, por exemplo, para descrever a satisfação do usuário com o produto ou comparar o desempenho do software com outros similares. A qualidade é uma métrica muito difícil de se medir, pois, os usuários tendem a entender qualidade como sendo o nível de conformidade do software com suas especificações. A qualidade do software é formalmente definida pelo padrão ISO/IEC 9126 (ISO/IEC [2004]) e é composta por 6 características de alto nível: funcionalidade, confiabilidade, usabilidade, desempenho, manutenibilidade e

portabilidade. O padrão define também uma série de atributos para cada característica. Fenton & Pfleeger [1998] e Kan [2003] apresentam a manutenibilidade e a confiabilidade como sendo as métricas mais relevantes:

Manutenibilidade: É o estágio do ciclo de vida do software que se inicia após o lançamento do software. Acredita-se que ocupa a maioria do ciclo de vida do software (Hatton [1998]) e, portanto, é responsável pela maioria dos custos de desenvolvimento. O processo de manutenibilidade de software é oficialmente descrito pelo padrão ISO/IEC [2006]. O glossário IEEE de engenharia de software (IEEE [1990]) define 3 tipos de operações de manutenção: corretiva, preventiva e adaptativa. Baseado nessas definições, Coleman et al. [1994b] definem manutenibilidade como a facilidade com a qual o software, por ser modificado com o objetivo de corrigir e remover erros, se adaptar ao um novo ambiente de operações, satisfazer requisitos ou melhorar a estrutura do software de forma a tornar a manutenção mais fácil. Coleman et al. [1994b] apresentam uma fórmula para calcular a métrica de manutenibilidade do software. Essa fórmula, chamada MI, utiliza uma composição das seguintes métricas:

- Média de volume Halstead por módulo;
- Média da complexidade ciclomática por módulo;
- Média das linhas de código por módulo;
- Percentual de linhas de comentário sobre o total de linhas do software.

Confiabilidade: Pode ser definida como a probabilidade de funcionamento do software sem falhas por um período de tempo em um ambiente definido. A confiabilidade é uma métrica externa muito utilizada e está relacionada com a propriedade mais importante do software: corretude. Uma métrica simples de confiabilidade é a densidade de erros, expressa pelo número de erros identificados em determinado módulo ou conjunto de linhas de código do software. Muitos pesquisadores classificam os erros em dois tipos: erros conhecidos, que são os erros que foram descobertos durante os testes (antes do lançamento do produto) e erros latentes, que são os erros descobertos depois do lançamento do produto. Fenton & Pfleeger [1998] e Peled [2001] apresentam modelos de confiabilidade mais complexos.

2.2.2 Métricas de Processo

Eficácia de remoção de erros é uma métrica que mede a habilidade de correção de erros. Essa métrica é definida como: $\frac{N}{N+S}$, onde N é o número de erros encontrados durante o desenvolvimento do software e S é o número de erros identificados após o lançamento do software. A granularidade na qual N e S são calculados depende do processo de desenvolvimento utilizado. Por exemplo, em um processo em cascata, S é calculado após a entrega do software. Em processos mais iterativos, S é calculado após cada ciclo de iteração. Outros exemplos de métricas de processo são: a relação entre os erros relatados e corrigidos em um intervalo de tempo e o tempo médio para corrigir um erro.

2.3 Dados

Estudos empíricos utilizam várias fontes de dados, que se enquadram em uma das seguintes categorias:

- **Produto**: Refere-se principalmente ao código fonte, mas pode incluir outros artefatos como, por exemplo, documentação, imagens e arquivos de configuração de ferramentas.

- **Processo**: Exemplos de dados do processo são sistemas de controle de versão (*Version Control Systems*), comunicação eletrônica (emails, mensagens instantâneas), arquivos com dados da organização (sistemas Wiki, arquivos) e sistemas de rastreamento de erros (*Bug Tracking Systems*).

Nem todos os tipos de dados são igualmente utilizados em estudos empíricos. Os dados de produto estão disponíveis para estudo desde o início da engenharia de software, nos anos 60. Desde o início do movimento de desenvolvimento de software livre nos anos 90, muitos softwares vêm sendo codificados, utilizando sistemas de controle de versão e estão disponíveis na Internet, tornado-se uma fonte interessante para pesquisas. Uma nova subárea de pesquisa da engenharia de software, conhecida como mineração de repositórios de software, explora a utilização de dados de processo, conjuntamente com dados de produto, de forma a avaliar

o processo de desenvolvimento e qualidade dos projetos de software. A maioria destes estudos empíricos utiliza repositórios de software para extrair informações sobre o processo de desenvolvimento e para criar modelos de evolução do software.

2.3.1 Código Fonte

É o alvo mais comum para a aplicação de métricas de produto. O código fonte de um projeto representa uma imagem de toda a linha de vida do desenvolvimento do software. Ferramentas podem ser utilizadas para se obter informações sobre o tamanho, complexidade e estrutura do projeto. Uma análise mais aprofundada desses artefatos de software pode gerar informação de autoria a nível de arquivo (Frantzeskou et al. [2006]) ou linha (German et al. [2009]), a licença sob a qual é distribuído (Scacchi [2004]), o tamanho e a cobertura da documentação do código ou a existência de partes de código que precisam ser refatoradas (Mäntylä et al. [2003]). Outra utilização é para identificação de módulos com propensão a erros utilizando métricas de processo (Zimmermann et al. [2008]).

2.3.2 Sistemas de Controle de Versão

Sistemas de controle de versão (*Version Control Systems*) armazenam o código fonte juntamente com informação sobre o desenvolvimento do projeto. Normalmente o fluxo de utilização de um sistema de controle de versão é o seguinte:

- O comando `checkout` acessa a versão mais recente do repositório do projeto e cria uma cópia local;

- O comando `update` atualiza o estado da cópia local acessando todas as alterações desde o último `checkout` ou `update` e criando novos arquivos, sobrescrevendo arquivos antigos e removendo os arquivos apagados do repositório;

- Caso um arquivo tenha sido alterado, tanto no repositório quanto na cópia local, o sistema de controle de versão tenta integrar as mudanças automaticamente. Em caso de insucesso, é gerado um conflito que pode ser resolvido pelo usuário, através da edição do arquivo e utilização do comando `resolve`;

- O comando `commit` submete mudanças feitas pelo desenvolvedor ao repositório;
- Os comandos do tipo `changeset` podem modificar, adicionar ou remover arquivos;
- Uma marca (*tag*) se refere a um estado específico do projeto sob controle de versão como, por exemplo, uma versão de lançamento do projeto;
- Quando os desenvolvedores querem testar novas funcionalidades que podem deixar o código instável, usualmente é criada uma ramificação (*branch*) que é um caminho paralelo do projeto. Quando a nova funcionalidade é testada, a ramificação é inserida no caminho principal do projeto. Esse processo é denominado mesclagem (*merge*).

Normalmente é necessário realizar um pré-processamento dos repositórios e os dados são armazenados em uma base de dados (Zimmermann & Weißgerber [2004]; Mockus [2009]; Gousios & Spinellis [2009]). Usualmente, o pré-processamento inclui os seguintes passos:

- Extração do histórico de revisões do repositório. O histórico de revisões contém informações sobre quais arquivos foram alterados em cada `commit` e também sobre a mudança, como, por exemplo, autor e horário da mudança.
- Identificação das revisões do projeto: O histórico de revisões é usado para extrair informações sobre a versão que está sendo processada, como a presença de ramificações e marcas.
- Recriação do estado dos arquivos: Utilizando o histórico de revisões e os dados pré-processados das revisões anteriores, os algoritmos de processamento recriam a árvore de arquivos do projeto em todas as etapas do desenvolvimento.
- Identificação da ordem das revisões: Alguns sistemas de controle de versão mais antigos não possuem essa informação e é necessário realizar esse processamento na etapa final.

Sistemas de controle de versão são fontes importantes de dados de processo e de produto. Do ponto de vista de pesquisa é como se fosse possível voltar no tempo em pontos específicos do desenvolvimento e correlacionar esses estados com métricas de processo. Sistemas de controle de versão têm sido utilizados para avaliar manutenibilidade (Mockus & Votta [2000]; Purushothaman [2005]), refatoração (Kim et al. [2005]), esforço de desenvolvimento (Amor et al. [2006]) e identificação de erros de projeto (Hassan & Holt [2005]; Kim et al. [2007]).

2.3.3 Comunicação Eletrônica

Desenvolvedores de software utilizam sistemas de comunicação eletrônica para tornar a troca de informações mais eficiente. Isso se aplica especialmente em desenvolvimento de softwares nos quais as equipes estão fisicamente distribuídas. As ferramentas de comunicação eletrônica utilizam emails e mensagens instantâneas para discussões entre os desenvolvedores. Mensagens instantâneas são consideradas informais e pessoais e, dessa forma, poucos históricos estão disponíveis para serem estudados (Shihab et al. [2009]). Ao contrário, emails são normalmente armazenados e, em caso de sistemas de código aberto, esses arquivos estão disponíveis para consulta (Michael Weiss [2007]).

O processamento dos emails não é trivial, pois a comunidade de software livre utiliza uma grande variedade de softwares de gerenciamento de listas de emails. Além disso, as informações normalmente são publicadas nas páginas dos projetos. Não existem ferramentas capazes de processar automaticamente listas de emails. Dessa forma, normalmente, o processamento de mensagens de listas de emails deve ser personalizado para cada projeto, o que muitas vezes inviabiliza a utilização de grandes volumes de dados nos trabalhos de pesquisa. Outro desafio é a extração das informações semânticas dos emails, sendo necessária a utilização de algoritmos de processamento de linguagem natural e recuperação de informação. Esses são os principais motivos pelos quais as comunicações eletrônicas são utilizadas para estudar os aspectos sociais do desenvolvimento de software e não as características técnicas do processo de desenvolvimento.

2.3.4 Sistemas de Rastreamento de Erros

Os sistemas de rastreamento de erros coletam relatórios de erros e solicitações de funcionalidades de usuários e desenvolvedores. A maioria dos sistemas de rastreamento de erros são baseados em relatórios de erros. Cada novo erro recebe um número de identificação único que será utilizado como referência pelos desenvolvedores. Os sistemas de rastreamento de erros podem utilizar estados como: novo, aberto, duplicado, inválido, assinalado para correção, parcialmente resolvido, testando, resolvido e fechado. No momento da criação, cada erro recebe uma severidade e uma prioridade de resolução. Usuários e desenvolvedores podem comentar sobre o progresso de resolução do erro. Os sistemas de rastreamento de erros normalmente, mantém uma base de dados com todas as atividades relativas aos erros.

Os sistemas de rastreamento de erros são uma fonte muito importante de informação de pesquisa, pois contém, dados de processo e de produto em um formato estruturado que pode ser facilmente processado. Os dados de produto são os defeitos do projeto e os dados de processo podem ser extraídos através da análise das características da operação de correção do erro como, por exemplo, o tempo necessário para corrigir defeitos críticos ou o número de erros abertos em um determinado momento. As entradas nos sistemas de rastreamento de erros podem ser correlacionadas com os sistemas de controle de versão através da análise de mensagens de `commit` (Śliwerski et al. [2005]) ou dos arquivos envolvidos nos relatórios de erros (Canfora & Cerulo [2005]).

2.3.5 Repositórios Disponíveis para Pesquisa

Um desenvolvimento recente no campo de fontes de dados é o fornecimento de conjuntos pré-processados de dados para pesquisa. Como a análise desses dados têm se tornado mais sofisticada e pesquisadores precisam de explorar as propriedades de grandes conjuntos de dados de projetos de código aberto, os experimentos ficam cada vez mais difíceis de serem conduzidos. A extração, pré-processamento, extração e agrupamento de métricas de grandes projetos são tarefas tediosas e propensas a erro. Em outras áreas da Ciência da Computação pesquisadores possuem conjuntos de dados pré-definidos ou plataformas de software para desenvolver e

executar experimentos. Dessa forma, os experimentos são facilmente replicáveis e os resultados podem ser comparados e estendidos por outros pesquisadores. Existem hoje dois projetos que têm por objetivo fornecer esses conjuntos de dados para pesquisa empírica:

- FlossMole (Howison et al. [2006]): Foi o primeiro a fornecer um conjunto de dados de pesquisa aberto. Coleta e processa dados de diversos repositórios de código aberto (SourceForge, ObjectWeb e Free Software Foundation). Útil para pesquisa que não envolve acesso a código fonte.

- FlossMetrics (FLOSS Metrics Consortium [2010]): A base de dados é composta pelas diferentes versões do código fonte, listas de comunicação e relatórios de erros. Fornece também algumas métricas simples do código fonte como, por exemplo, complexidade ciclomática e volume de Halstead.

2.4 Ferramentas

O cálculo das métricas é um pré-requisito para praticamente qualquer estudo envolvendo dados de projetos. Existem diversas ferramentas para auxiliar nessa tarefa e, a maioria delas, funciona com linguagens C++ e Java. Uma característica comum da maioria das ferramentas disponíveis é que as métricas são calculadas para cada versão do código-fonte.

A extração de métricas isoladamente não é suficiente para a condução de experimentos. Várias ferramentas foram desenvolvidas para automatizar a extração e processamento de dados de repositórios. A ferramenta CVSAnalY (Robles et al. [2004]) converte dados de repositórios CVS para um formato relacional. O CVSAnalY trabalha em três passos; primeiro realiza o processamento do histórico do CVS, em seguida limpa os dados e extrai as informações semânticas do repositório e, finalmente, gera as estatísticas de desenvolvimento do projeto.

A ferramenta Hackystat, proposta por Johnson et al. [2005], foi a primeira a considerar tanto métricas de processo quanto métricas de produto. A Hackystat é baseada em um modelo ativo de monitoramento no qual são instaladas ferramentas que armazenam informações de utilização de softwares durante o processo de desenvolvimento. Dessa forma, a ferramenta não pode ser utilizada para analisar

dados de projetos que não a utilizaram desde o início. A *Release History Data-Base* (RHDB) (Fischer et al. [2003]) foi a primeira ferramenta a combinar dados de sistemas de controle de versão e sistemas de rastreamento de erros. German [2004] propôs uma ferramenta similar à RHDB, a Softchange. Essa ferramenta é capaz de fazer inferências a partir do código fonte após o término da extração.

Finalmente, as ferramentas Alithea (Gousios & Spinellis [2009]) e Kenyon (Bevan et al. [2005]) são plataformas que realizam o pré-processamento de dados de vários tipos de sistemas de controle de versão. São capazes de exportar dados para outras ferramentas que podem ser executadas automaticamente. A base de dados da Kenyon foi desenvolvida especificamente para estudos de instabilidade de software, enquanto a Alithea é uma ferramenta mais genérica.

2.5 Técnicas de Identificação e Predição de Erros de Software

Para efetuar a predição de erros em software pode-se utilizar métricas de produto isoladamente ou em conjunto com métricas de processo (D'Ambros et al. [2010]). Para se utilizar exclusivamente métricas de produto é necessário apenas o código fonte do projeto e a contagem de erros de cada módulo ou arquivo. Quando se utiliza conjuntamente, métricas de processo, é necessário que se tenha acesso aos sistemas de controle de versão e de rastreamento de erros nos quais o software foi desenvolvido. A seguir, serão apresentados, os principais trabalhos que utilizam as duas estratégias.

2.5.1 Métricas de Produto

As técnicas de predição de erro baseadas apenas em métricas de produto assumem que o estado atual das métricas, extraídas do código fonte, influenciam a ocorrência de erros futuros. Basili et al. [1996] utilizaram as métricas de Chidamber e Keremer (CK), apresentadas na Tabela 2.1, em oito implementações de um sistema de gerenciamento de informações. O trabalho discute as principais vantagens e desvantagens de utilização das métricas CK, sendo que várias delas foram úteis para identificar módulos propensos a erro.

Ohlsson & Alberg [1996] utilizaram diversas métricas de grafos, dentre as quais a complexidade ciclomática, em um software de telecomunicações da Ericsson. No sistema, composto por 130 módulos, 60% dos erros estavam localizados em apenas 20% dos módulos. O modelo desenvolvido no trabalho foi capaz de identificar 20% dos módulos nos quais 47% dos erros estavam localizados.

Emam et al. [2001b] utilizaram as métricas CK conjuntamente com as métricas de acoplamento de Briand (Briand et al. [1999]) em um sistema comercial implementado em Java. Esse trabalho utilizou regressão logística[1] para selecionar um subconjunto de métricas capaz de identificar os módulos mais propensos a erros. As métricas CK também foram utilizadas por Subramanyam & Krishnan [2003] em sistemas desenvolvidos em Java e C++. Para tanto, foi aplicada regressão linear[2] por mínimos quadrados. Um trabalho similar, proposto por Gyimóthy et al. [2005] analisou o software Mozilla. Na análise, foram empregadas técnicas de regressão logística, regressão linear e aprendizado de máquina (Mitchell [1997]).

Nagappan & Ball [2005a] empregaram técnicas de regressão linear e análise de discriminantes (Klecka [1980]) em 199 módulos do Windows Server 2003. O sistema de análise estática foi projetado para efetuar a classificação dos módulos em dois grupos: alta e baixa qualidade. Um percentual alto (82,91%) foi corretamente classificado. Nagappan et al. [2006] utilizaram técnicas de regressão linear e regressão logística conjuntamente com análise de componentes principais[3] para estudar várias métricas de código fonte em cinco projetos desenvolvidos pela Microsoft. Os resultados mostraram que é possível criar modelos eficientes para cada um dos projetos, mas nenhum dos preditores obteve um bom desempenho em todos os projetos. Zimmermann et al. [2007] utilizaram regressão logística para classificar os módulos do software Eclipse como propensos ou não propensos a erros. Janes et al. [2006] analisaram dados de cinco projetos industriais. As técnicas utilizadas foram regressão de poisson[4], regressão binomial negativa[5] e regressão inflada de zeros (Hilbe [2007]). A última técnica apresentou melhores resultados, pois modela bem grandes quantidades de zeros (módulos sem erros), como ocorre

[1]Para definição dessa técnica, ver Seção 3.3.5
[2]Para definição dessa técnica, ver Seção 3.3.2
[3]Para definição dessa técnica, ver Seção 3.3.1
[4]Para definição dessa técnica, ver Seção 3.3.3
[5]Para definição dessa técnica, ver Seção 3.3.4

nos projetos estudados.

2.5.2 Métricas de Produto e Processo

Quando se utiliza conjuntamente, métricas de produto e métricas processo, é necessário ter acesso ao sistema de controle de versões do projeto. Nagappan & Ball [2005b] apresentam indicações que medições de mudanças relativas são melhores preditores de erros que mudanças absolutas. Por exemplo, ao invés de se utilizar o número de mudanças de linhas de código (LC) absolutas é melhor utilizar:

$$\frac{\text{Número de mudanças de LC}}{\text{LC}} \tag{2.1}$$

Os resultados apresentados foram obtidos analisando o histórico de versões do Windows Server 2003 e empregando técnicas de regressão linear, análise de componentes principais e análise de discriminantes.

Hassan [2009] introduziu o conceito de entropia de mudanças, uma medida de complexidade de mudanças. A entropia de mudanças pode ser definida como:

$$H_n(P) = -\sum_{k=1}^{n}(p_k \cdot log_2 p_k) \tag{2.2}$$

onde $p_k \geq 0, \forall k \in 1, 2, ..., n$ e $\sum_{k=1}^{n} p_k = 1$.

p_k pode ser definido como:

$$\frac{\text{Número de alterações no intervalo}}{\text{Número total de pontos no intervalo}} \tag{2.3}$$

A entropia de mudanças foi comparada com o número de mudanças e o número de erros e apresentou resultados melhores. A relevância da entropia de mudanças foi avaliada em seis sistemas de código aberto: FreeBSD, NetBSD, OpenBSD, KDE, KOffice e PostgreSQL. Hassan & Holt [2005] propuseram um algoritmo para construir uma lista de módulos propensos a erros. Eles utilizaram, de maneira independente, quatro métricas para determinar essa lista: Módulos frequentemente modificados (MFM); Módulos recentemente modificados (MRM); Módulos frequentemente corrigidos (MFC); Módulos recentemente corrigidos (MRC). Os módulos MRM e MRC apresentam maior número de erros. Uma

abordagem similar é o *cache* de erros proposto por Kim et al. [2007]. Esse trabalho também utiliza as quatro métricas: MFM, MRM, MFC e MRC, mas estas, analisadas conjuntamente. A análise foi realizada em sete sistemas de código aberto: Apache, PostgresSQL, Subversion, Mozilla, JEdit, Columba e Eclipse.

Moser et al. [2008] utilizaram várias métricas (mudança de código, erros passados, refatorações, número de autores, tamanho e idade do arquivo, etc.) para realizar a predição da ausência ou presença de erros nos arquivos do Eclipse. As técnicas utilizadas foram regressão logística, Naive Bayes e árvore de decisão (Mitchell [1997]). Ostrand et al. [2005] utilizaram métricas de mudança e erros passados para identificar módulos propensos a erros em dois projetos industriais. A técnica empregada foi a regressão binomial negativa. Modelos não lineares (Seber & Wild [2003]) baseados nas métricas de erros e mudanças foram propostos por Bernstein et al. [2007]. Seis *plugins* do Eclipse são utilizados para validar os modelos. A técnica empregada foi árvore de decisão.

2.6 Métricas de Complexidade de Hardware

As métricas de complexidade apresentadas nesta Subseção já foram utilizadas em desenvolvimento de circuitos digitais, ou seja, em linguagens de descrição de hardware. Não foi encontrado nenhum trabalho que utilize métricas de código fonte para identificar módulos HDL propensos a erros. A seguir, as métricas de complexidade serão discutidas dentro de dois contextos distintos: acadêmico e industrial.

Schafers [1995] propõe adaptações de métricas de engenharia de software para estimar a complexidade de projetos de circuitos integrados em Verilog HDL. Neste trabalho, foram estudadas mais de cem métricas de complexidade. Com o objetivo de definir as métricas que melhor refletem a complexidade, cinco desenvolvedores classificaram trinta e oito projetos em Verilog HDL, de acordo com: estrutura, conexão, concorrência, qualidade do código, dados, legibilidade do código (*readability*) e facilidade de manutenção (*maintainability*). Após esta classificação, foi calculada a correlação entre as métricas automaticamente extraídas e a complexidade definida manualmente pelos desenvolvedores.

Protheroe & Pessolano [2000] propõem uma metodologia de extração automática de métricas de implementações RTL (em VHDL) e circuitos sintetizados. O objetivo deste trabalho é definir parâmetros de qualidade de um projeto de circuito integrado, utilizando métricas extraídas automaticamente tanto da implementação RTL, quanto do circuito sintetizado. As métricas apresentadas no trabalho são: linhas de código, complexidade ciclomática, número de sinais de E/S, número de portas lógicas após a síntese, custo de teste e confiabilidade.

Mastretti [1995] apresenta a ferramenta SAVE que extrai métricas de complexidade, eficiência e facilidade de síntese (*synthesizability)* de implementações VHDL. As métricas de complexidade apresentadas, são: complexidade ciclomática (McCabe [1976]), nível de aninhamento e fluxo de informações.

Holzer & Rupp [2006] empregam métricas de hardware e software para auxiliar na escolha de módulos a serem simulados. A linguagem de descrição de hardware utilizada é SystemC. As principais classes de métricas, são: métricas de estrutura (número de classes, número de instâncias, profundidade hierárquica, etc), métricas funcionais (extraídas de grafos de controle de fluxo), complexidade ciclomática, grau de paralelismo, controle (número de `if`s, `while`s, etc) e utilização de memória.

A Tabela 2.2 resume as principais métricas de complexidade utilizadas em projetos de circuitos integrados.

No contexto industrial, a empresa **Cistel Technology Inc.** utilizou diversas métricas para comparar a complexidade de implementações Verilog em nível RTL e comportamental (Cistel Technology Inc. [1997, 1998]). As métricas de Complexidade Ciclomática, Linhas de Código e Comentários já foram utilizadas nos trabalhos resumidos na Tabela 2.2. Outras métricas de complexidade utilizadas, são:

- Número de operandos;
- Número de operandos, excluindo assinalamentos (`=,<=`);
- Número de expressões;
- Número de expressões de controle (`if,for, while, repeat, case`);

Tabela 2.2. Métricas de complexidade hardware.

Métrica	**Descrição**
Fração de comentários	$\frac{\text{Número de bytes de comentários}}{\text{Número de bytes de código}}$ (Schafers [1995]).
Tamanho	Número de bytes e linhas de código (Schafers [1995]; Protheroe & Pessolano [2000]).
Estrutura	Hierarquia representada por um grafo, no qual cada instância é um nodo. Pode-se calcular a altura e as larguras absolutas, relativas e médias do grafo G (Schafers [1995]), sendo que a métrica original de software foi proposta por Blaschek [1985]. Número de classes, instâncias e profundidade (Holzer & Rupp [2006]).
Sincronização	Número de variáveis de sincronização no código (Schafers [1995]).
Fluxo	Controle de fluxo utilizando caminhos de execução acíclicos (Schafers [1995]) e complexidade ciclomática (Schafers [1995]; Protheroe & Pessolano [2000]; Mastretti [1995]; Holzer & Rupp [2006]), sendo que os equivalentes em software foram propostos, respectivamente, por Nejmeh [1988] e McCabe [1976]. Número de `if`s, `while`s, etc (Holzer & Rupp [2006]).
Halstead (Halstead [1977])	Mede dificuldade ou volume (Schafers [1995]).
Joergensen (Lehner et al. [1992])	Métrica legibilidade de código (Schafers [1995]).
Sinais	Número de sinais de E/S e de sinais internos (Protheroe & Pessolano [2000]).
Nível de aninhamento	Profundidade máxima de declarações aninhadas (Mastretti [1995]).
Fluxo de informações	Troca de informações entre processos (Mastretti [1995]).
Grau de paralelismo	Proposta por Holzer & Rupp [2006].
Acessos à memória	Proposta por Holzer & Rupp [2006].

- Número de partições em um módulo (`always, initial`);
- Número de registradores;
- Número de módulos instanciados;
- Número de portas unidirecionais;
- Número de portas bidirecionais;
- Número de entradas para outros módulos;
- Número de saídas de outros módulos;
- Número de vezes que os clocks são usados;
- Número máximo de estados em um módulo;
- Número total de clocks;
- Número de assinalamentos bloqueantes;
- Número de assinalamentos não-bloqueantes.

Finalmente, a ferramenta ARDID (Torroja et al. [1999]), incorporada ao fluxo de desenvolvimento da Cadence, realiza uma série de análises no código HDL:

- Lista de sensibilidade (sincronização): Aponta erros e omissões nas listas de sensibilidade;
- Estilo de codificação: Influencia a manutenibilidade e os resultados de síntese;
- Utilização de objetos: Lista objetos declarados, mas não utilizados;
- Sinais de alta impedância: Em algumas metodologias de desenvolvimento ou fabricação de circuitos integrados a utilização de sinais de alta impedância não é permitida ou recomendada;
- Clock e reset: São os sinais mais importantes do circuito integrado. Análise objetiva identificação de *glitches*;

- Objetos e saídas conectados a flip-flops ou latches;
- Conectividade.

Capítulo 3

Metodologia para Identificação de Módulos HDL Propensos a Erros

Este Capítulo discute a metodologia utilizada para estimar a propensão de módulos HDL (*Hardware Description Language*) a possuírem erros. A Figura 3.1 apresenta as principais etapas desse processo. Inicialmente, durante a etapa **1**, diversas métricas são extraídas de um repositório de desenvolvimento de circuito integrado. Essas métricas podem ser de produto (extraídas do código fonte HDL) ou de processo (extraídas histórico do repositório). O repositório de desenvolvimento do circuito integrado é composto pelo sistema de controle de versão e pelo sistema de rastreamento de erros. Na etapa **2**, as métricas que melhor se correlacionam com a propensão dos módulos a erros são selecionadas. A seguir, na etapa **3**, o modelo de predição de erros é construído e ajustado. As técnicas utilizadas incluem análise de componentes principais e seleção dos componentes que tenham maior significância. Finalmente, na etapa **4**, o modelo deve ser validado, comparando os resultados do preditor com os dados de erros observados. A seguir, cada uma das etapas da metodologia proposta é descrita com mais detalhes nas Seções 3.1, 3.2, 3.3 e 3.4.

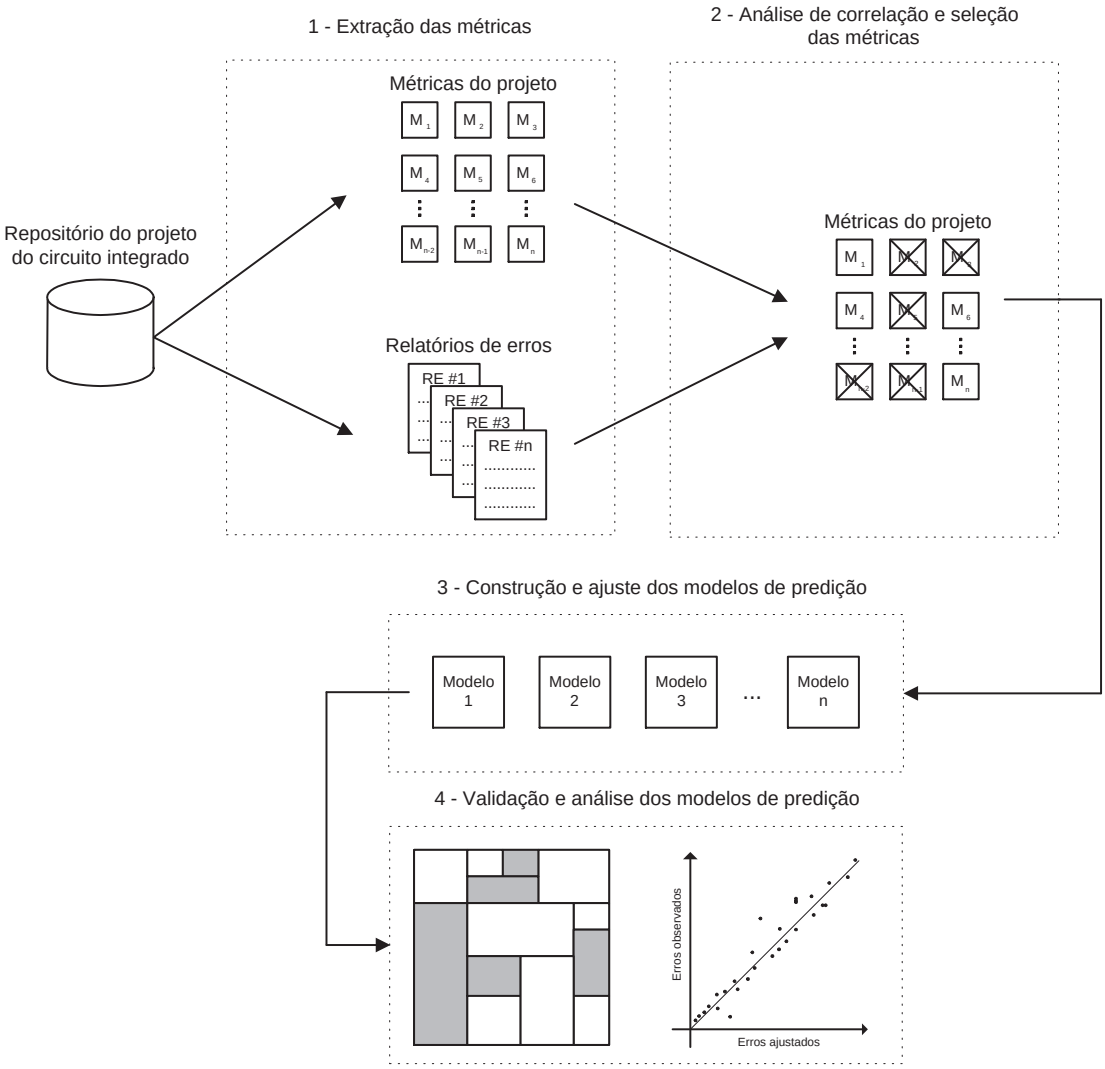

Figura 3.1. Visão geral da metodologia proposta.

3.1 Extração das Métricas para Construção do Modelo

A Figura 3.2 apresenta o processo de extração das métricas e relatórios de erros do repositório de desenvolvimento do circuito integrado. Os sistemas de controle de versão possuem um comando que retorna o histórico de modificações

de cada um dos arquivos do projeto. Dessa forma, cada versão de cada um dos arquivos do projeto do circuito integrado é extraída e pode ter suas métricas de produto processadas. As métricas de processo também podem ser extraídas do histórico de modificações do sistema de controle de versão. Em seguida, as métricas de produto e processo são armazenadas em uma base de dados com objetivo de facilitar a etapa de análise de correlação. É importante combinar e sincronizar as informações dos sistemas de rastreamento de erros e de controle de versão. Dessa forma, tem-se o número de erros de cada módulo do projeto em cada versão.

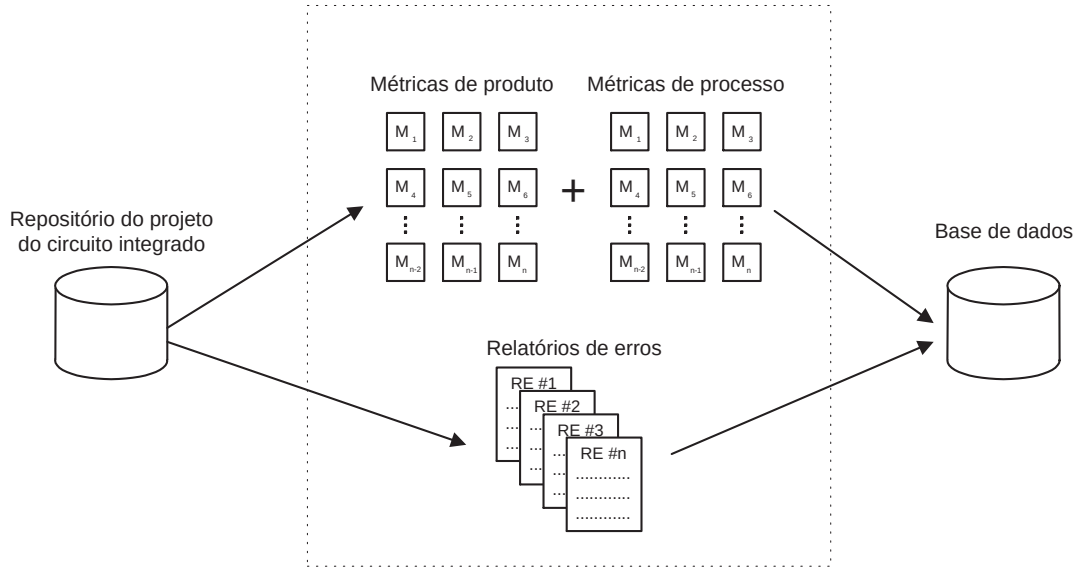

Figura 3.2. Extração das métricas e relatórios de erros do repositório.

A extração das métricas por versões permite que se estude a evolução dos projetos ao longo do tempo. Além disso, é possível estudar versões específicas do projeto ou apenas a versão final. No caso de projetos de circuitos integrados, a análise da versão final do projeto do circuito integrado é mais fácil pois, ao contrário de projetos de software, não existem repositórios de código aberto adequados para a extração de métricas por versões.

3.2 Análise de Correlação das Métricas com os Erros

Para a seleção das métricas do modelo, uma análise de correlação entre as métricas e os erros deve ser realizada. As métricas que têm maior correlação com o número de erros são selecionadas. A Figura 3.3 ilustra esta etapa. Os coeficientes de Pearson, Spearman ou Kendall (Cohen et al. [2003]) podem ser utilizados para calcular a correlação entre as métricas e os erros. Esses coeficientes de correlação já foram utilizados em outros trabalhos da área de engenharia de software (Zimmermann [2008]; Fenton & Pfleeger [1998]). O coeficiente de correlação de Pearson necessita que os dados estejam distribuídos normalmente e que a associação entre os elementos seja linear. Ao contrário, os coeficientes de correlação de Spearman e Kendall são técnicas robustas que podem ser utilizadas mesmo quando a associação entre os valores não for linear.

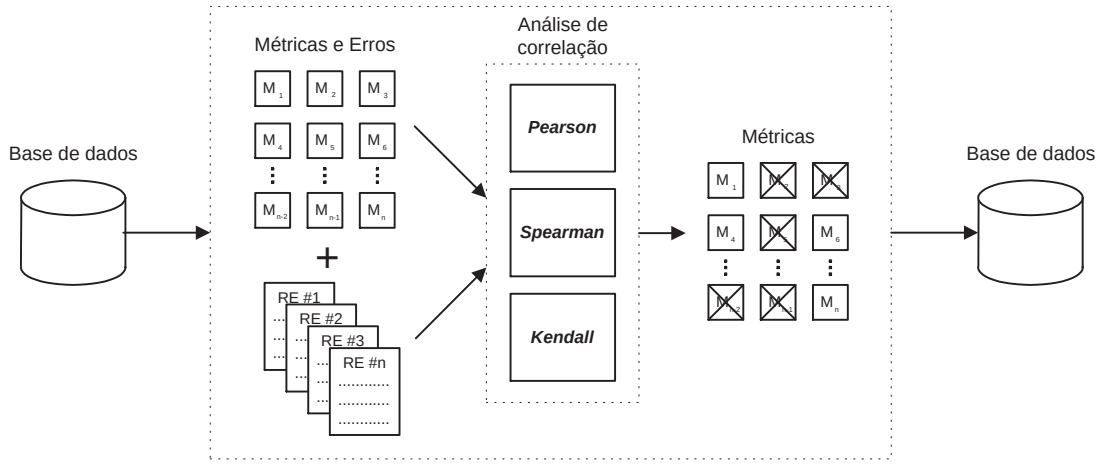

Figura 3.3. Análise de correlação das métricas.

O valores dos coeficientes de correlação de Pearson, Spearman e Kendall estão compreendidos no intervalo entre -1 e $+1$. Valores mais próximos a -1 indicam maior correlação negativa, enquanto valores mais próximo a $+1$ indicam correlação positiva. Valores próximos a 0 indicam que não existe correlação. A seguir, serão apresentados maiores detalhes sobre os coeficientes de correlação de

Pearson, Spearman e Kendall.

3.2.1 Coeficiente de Correlação de Pearson

O coeficiente de correlação de Pearson é muito utilizado em dados que seguem a distribuição normal. Para estudar a correlação entre dois atributos, x e y formam-se pares (x_i,y_i) onde i é o atributo. É importante observar que tanto x quanto y devem seguir uma distribuição normal ou pelo menos próxima. O número total de pares é n e, para cada atributo, deve-se calcular a média e a variância. As médias de x e y são representadas, respectivamente, por $\bar{x}$ e $\bar{y}$. As variâncias de x e y são representadas, respectivamente, por $var(x)$ e $var(y)$. Dessa forma, o coeficiente de correlação de Pearson é representado por:

$$r = \sum_{i=1}^{n} \frac{(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{n \cdot var(x) \cdot var(y)}} \tag{3.1}$$

O coeficiente de correlação de Pearson varia entre -1 e $+1$. Caso r seja $+1$, x e y têm uma correlação positiva perfeita, ou seja, quando x aumenta seu valor, y também aumenta seu valor de maneira linear. Da mesma forma, quando o valor de r é -1 uma correlação negativa perfeita (x aumenta, y diminui linearmente) existe. O valor de r igual a 0 significa que não existe correlação entre os valores de x e y, ou seja, quando x aumenta, y pode tanto aumentar quanto diminuir seu valor. Testes estatísticos podem ser usados para verificar se o valor calculado de r é significantemente diferente de zero a um certo grau de significância, ou seja, o cálculo de r deve ser acompanhado por um teste que indique o nível de confiança da associação.

3.2.2 Coeficiente de Correlação de Spearman

O coeficiente de correlação de Spearman não necessita que a distribuição dos dados seja normal e, portanto, é uma técnica mais robusta. O cálculo do coeficiente de correlação de Spearman é similar ao do coeficiente de correlação de Pearson, mas os valores de x e y são baseados em ranqueamentos dos atributos. Assim sendo, os valores dos atributos são ordenados em ordem crescente e é assinalado 1 para o menor valor, 2 para o segundo menor e, assim, sucessivamente. Se um ou

mais valores de x e y são iguais, é atribuída a média desses valores. Por exemplo, se os dois menores valores são 20 cada um, será atribuído o valor 1.5, que é a média de 1 e 2.

3.2.3 Coeficiente de Correlação de Kendall

A teoria do coeficiente de correlação de Kendall é diferente. Para cada dois pares de valores de atributos, (x_i,y_i) e (x_j,y_j), caso exista uma correlação positiva entre os atributos, quando x_i é maior que x_j é muito provável que y_i seja maior que y_j. Da mesma forma, se existe uma correlação negativa, quando x_i é menor que x_j é muito provável que y_i seja menor que y_j. O coeficiente de correlação de Kendall, τ, é baseado na comparação dos pares x e y no vetor de atributos.

O coeficiente de correlação de Kendall é importante porque pode ser generalizado para fornecer coeficientes de correlação parciais. Esses valores parciais são úteis quando a relação entre dois atributos for causada, na verdade, pela relação de ambos com um terceiro atributo. Maiores detalhes de utilização dos coeficientes parciais podem ser encontradas em Siegel & Castellan [1988].

3.3 Construção e Ajuste do Modelo de Predição

Nesta etapa, modelos de regressão multivariada são construídos, conforme ilustra a Figura 3.4. A variável dependente é o número de erros. Na regressão multivariada têm-se duas ou mais variáveis independentes que, no caso, são as métricas. Nas regressões multivariadas a correlação entre as variáveis independentes pode causar instabilidades, diminuindo significantemente o desempenho do modelo. Esse problema é conhecido como multicolinearidade e normalmente aumenta muito a variância na estimação da variável dependente. Uma forma de resolver o problema é a utilização de análise de componentes principais (*Principal Component Analysis* - PCA) (Jackson [2003]). No PCA, um pequeno número de combinações lineares não correlacionadas são selecionadas para serem utilizadas na regressão. As combinações são independentes e não sofrem do problema da multicolinearidade.

Após a aplicação da análise de componentes principais, a regressão multivariada é executada com novos componentes independentes gerados. Assim, as variáveis independentes utilizadas na regressão multivariada não são mais as métricas e sim os componentes principais. Cada um dos componentes principais é um vetor de tamanho n, onde n é o número de métricas.

Com o modelo construído, deve-se verificar a significância estatística de cada um dos componentes principais utilizados na regressão multivariada. Em estatística, um resultado é significante e, portanto, tem significância estatística, se for improvável que tenha ocorrido por acaso. Dessa forma, a significância mede a importância de cada uma das variáveis independentes, através de um teste de hipótese nula, ou seja, atribui o valor zero à variável e observa o resultado no modelo. Valores de significância superiores a 95% são considerados bons. O último procedimento dessa etapa é a remoção dos componentes principais do modelo com significâncias inferiores a 95%.

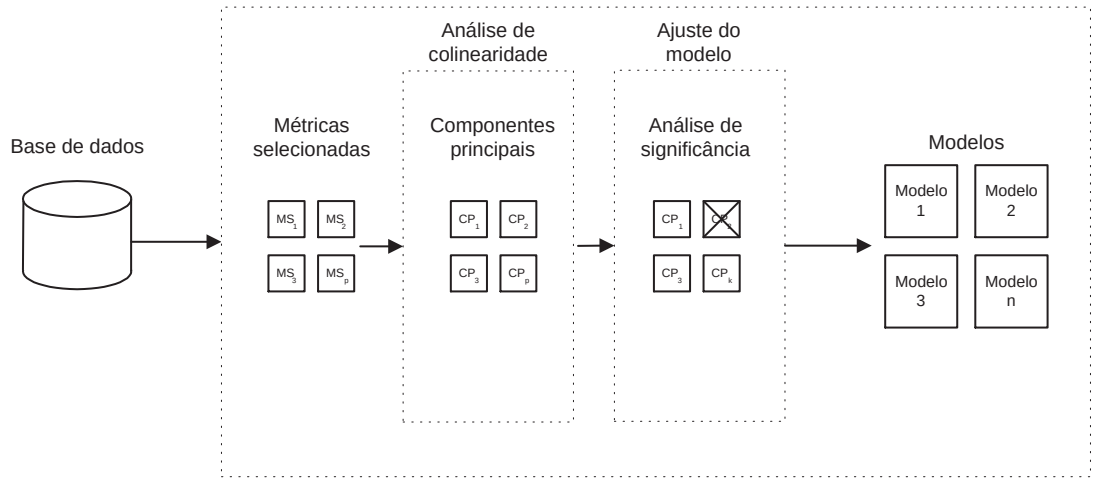

Figura 3.4. Construção do modelo de predição.

A seguir serão apresentados mais detalhes sobre a análise de componentes principais e os seguintes técnicas de regressão: Linear, Poisson, Negativa Binomial e Logística.

3.3.1 Análise de Componentes Principais

A análise de componentes principais tem o objetivo de explicar a variância e a covariância de um conjunto de variáveis através de combinações lineares dessas variáveis. Apesar de serem necessários p componentes para reproduzir a variância total do sistema, normalmente k componentes principais podem substituir as p variáveis. Nesse caso, os k componentes principais têm, aproximadamente, a mesma quantidade de informação que as p variáveis originais. Assim sendo, os k componentes principais podem substituir as p variáveis e o conjunto original de dados composto por n observações de p variáveis, é reduzido a um conjunto de dados composto por n observações de k componentes principais (Johnson & Wichern [2001]). Quando a distribuição de probabilidades das variáveis é normal, os componentes principais, além de não correlacionadas, são independentes e têm distribuição normal. Entretanto, a suposição de normalidade não é requisito necessário para que a técnica de análise de componentes principais possa ser utilizada.

Algebricamente, os componentes principais são combinações lineares de p variáveis aleatórias: $X_1, X_2, ..., X_p$. Geometricamente, estas combinações lineares representam a seleção de um novo sistema de coordenadas, obtido através da rotação do sistema original com $X_1, X_2, ..., X_p$, sendo os eixos de coordenadas. Os novos eixos representam as direções com o máximo de variância e fornecem uma descrição simplificada da covariância. Os componentes principais dependem unicamente da matrix de covariância $\sum$ (ou da matrix de correlação ρ) de $X_1, X_2, ..., X_p$.

Sejam $X' = [X_1, X_2, ..., X_p]$, o vetor de variáveis aleatórias, $\sum$, a matriz de covariância e $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_p \geq 0$, os autovalores. Considere as seguintes combinações lineares:

$$\begin{aligned} Y_1 &= a_1'X = a_{11}X_1 + a_{12}X_2 + ... + a_{1p}X_p \\ Y_2 &= a_2'X = a_{21}X_1 + a_{22}X_2 + ... + a_{2p}X_p \\ &\vdots \\ Y_p &= a_p'X = a_{p1}X_1 + a_{p2}X_2 + ... + a_{pp}X_p \end{aligned} \tag{3.2}$$

Dessa forma, a variância de Y_i e a covariância de Y_i e Y_k são

(Johnson & Wichern [2001]):

$$var(Y_i) = a_i' \sum a_i, \text{ com } i = 1, 2, ..., p \tag{3.3}$$

$$cov(Y_i, Y_k) = a_i' \sum a_k, \text{ com } i, k = 1, 2, ..., p \tag{3.4}$$

Os componentes principais são as combinações lineares não correlacionadas de $Y_1, Y_2, ..., Y_p$ cujas variâncias sejam as maiores possíveis. O primeiro componente principal é a combinação linear com o máximo da variância, ou seja, $var(Y_1) = a_1' \sum a_1$ é maximizado. A variância de $var(Y_1) = a_1' \sum a_1$ pode ser aumentada multiplicando a_1 por uma constante. De forma a eliminar essa indeterminação é conveniente restringir os coeficientes do vetor ao valor 1. Dessa forma, é possível definir:

Primeira componente principal = combinação linear de $a_1'X$ que maximiza :

$$var(a_1'X) \text{ , sendo que } a_1'a_1 = 1 \tag{3.5}$$

Segunda componente principal = combinação linear de $a_1'X$ que maximiza :

$$\begin{aligned} var(a_2'X) \text{ , sendo que } a_2'a_2 = 1 \text{ e} \\ cov(a_1'X, a_2'X) = 0 \end{aligned} \tag{3.6}$$

i-ésima componente principal = combinação linear de $a_i'X$ que maximiza :

$$var(a_i'X) \text{ , sendo que } a_i'a_i = 1 \text{ e}$$
$$cov(a_i'X, a_k'X) = 0 \text{ para } k < i \tag{3.7}$$

3.3.2 Modelo de Regressão Linear

A regressão linear é muito utilizada para expressar uma associação através de uma fórmula linear. A técnica da regressão linear é baseada em um gráfico de dispersão. Cada par de atributos é expresso como um ponto, (x_i,y_i), e essa técnica calcula a reta que melhor se enquadra entre os pontos. Dessa forma, o objetivo é expressar um atributo y (variável dependente), em termos do atributo x (variável independente), em uma equação da forma:

$$y = a + bx \tag{3.8}$$

Na regressão linear, desenha-se uma reta a partir de cada ponto verticalmente superior ou inferior à reta de tendência, representando a distância entre o ponto e a reta, sendo que deve-se minimizar essa distância. A reta final é a que minimiza essas distâncias. A distância para cada ponto é chamada residual e a fórmula para calcular a reta da regressão linear minimiza a soma de quadrados dos residuais. Pode-se expressar o residual para um dado ponto, como:

$$r_i = y_i - a - bx_i \tag{3.9}$$

A minimização da soma dos quadrados dos residuais leva às seguintes equações para a e b:

$$b = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2} \tag{3.10}$$

$$a = \bar{y} - b\bar{x} \tag{3.11}$$

A técnicas dos mínimos quadrados não leva em consideração se os dados seguem a distribuição normal. Entretanto, para realizar qualquer teste estatístico relacionado aos parâmetros da regressão (por exemplo, determinar se o valor de b é significantemente diferente de 0), é necessário assumir que esses residuais seguem a distribuição normal. A abordagem dos mínimos quadrados deve ser usada

com cuidado quando existem muitos valores grandes ou atípicos, pois esses pontos podem distorcer as estimativas de a e b.

A regressão linear necessita distribuição normal e variância constante dos erros. Com dados reais, a maioria das premissas normalmente não são completamente satisfeitas. Nesses casos, técnicas de regressão alternativas devem ser adotadas (Freund & Simon [1996]).

3.3.3 Modelo de Regressão de Poisson

A distribuição de Poisson é particularmente adequada para contar eventos que ocorrem no tempo (Lloyd [1999]; King [1988]). No modelo de regressão de Poisson (MRP), a distribuição de Poisson determina a probabilidade de uma contagem, na qual a média da distribuição é uma função de variáveis independentes. MRP tem sido usado em engenharia de software para modelar o número de erros (Graves et al. [2000]) e esforço (Briand & Wüst [2001]).

O MRP necessita de equidispersão, ou seja, igualdade entre a variância condicional e a média condicional da variável dependente. Quando as premissas para o MRP não são satisfeitas, por exemplo, no caso de alta variância condicional na variável dependente, ou seja, alta probabilidade de pequenas ou altas contagens, pode-se usar a distribuição binomial negativa (King [1988]; Briand & Wüst [2001]).

Um processo de Poisson é um modelo que descreve a ocorrência de eventos ao longo do tempo (Papoulis [1991]) e, assume que a ocorrência de um evento em um intervalo depende apenas do tamanho do intervalo e não da história do processo. Uma distribuição de Poisson é uma distribuição dos números de eventos resultantes de um processo de Poisson.

A distribuição de Poisson para uma variável dependente y, e um vetor de n variáveis independentes $x = (x_1, ..., x_n)$ é dada por:

$$Pr(y|x) = \frac{e^{-\mu}.\mu^y}{y!} \tag{3.12}$$

onde μ é o valor da média condicional de y, $\mu = E(y|x) = \mu(y|x)$.

Na prática, a variância condicional da variável dependente no modelo é normalmente maior que a sua média condicional. Nesse caso, a variável dependente

é afetada por superdispersão que é causada pela incapacidade da distribuição de Poisson de capturar a heterogeneidade nos dados. Superdispersão compromete seriamente a qualidade do ajuste do modelo (Lloyd [1999]), resultando em uma significância do preditor super estimada (Cameron & Trivedi [1986]).

A heterogeneidade dos dados é levada em consideração pelo MRP através das variáveis independentes. O objetivo da análise estatística é encontrar funções de regressão lineares simples que modelem com precisão o comportamento dos dados.

A função de regressão exponencial é normalmente utilizada no MRP (Lloyd [1999]; Long [1997]) e corresponde à multiplicação das médias. A média condicional é dada por:

$$\mu(y|x) = e^{\beta_0+\beta_1 x_1+...+\beta_n x_n} = e^{x\beta} \tag{3.13}$$

onde β é o vetor de parâmetros do modelo.

3.3.4 Modelo de Regressão Binomial Negativo

O modelo de regressão binomial negativo (MRBN) é uma extensão do modelo de regressão de Poisson que permite que a variância condicional da variável dependente exceda a média condicional. O MRBN já foi utilizado para lidar com dados superdispersos em engenharia de software (Briand & Wüst [2001]).

O MRBN pode ser derivado da distribuição de Poisson, levando-se em consideração a heterogeneidade não observada (Briand & Wüst [2001]). No MRBN, a média μ é substituída pela variável aleatório $\tilde{\mu}$:

$$\tilde{\mu} = e^{x\beta+\varepsilon} \tag{3.14}$$

onde ε é o erro aleatório não correlacionado com x, ou seja, a heterogeneidade não observada.

A relação entre $\tilde{\mu}$ e o μ original é:

$$\tilde{\mu} = e^{x\beta}e^{\varepsilon} = \mu e^{\varepsilon} \tag{3.15}$$

Assumindo que $E(\varepsilon) = 0$, a contagem esperada após adicionar novas fontes de variação é a mesma do MRP, ou seja, $E(\tilde{\mu}) = \mu$. Para simplificar, a fórmula é

reescrita como $\tilde{\mu} = \mu\delta$, onde $\delta = e^{\varepsilon}$.

No MRBN, para uma dada combinação de variáveis independentes existe uma distribuição de $\tilde{\mu}$ ao invés de um único valor. Consequentemente, a função de distribuição de probabilidade para $\delta = e^{\varepsilon}$ deve ser especificada para determinar a probabilidade para a variável dependente. A distribuição resultante é uma combinação da distribuição de Poisson e a distribuição de probabilidade para δ, é:

$$Pr(y|x) = \int_0^{\infty} [Pr(y|x.\delta).Pr(\delta)]d\delta \tag{3.16}$$

Devido à forma fechada do resultado e a propensão para representar processos de Poisson, a distribuição gama com parâmetro positivo v, g_v é normalmente utilizada para δ:

$$g_v(\delta) = \frac{v^v}{\Gamma(v)}\delta^{v-1}e^{-\delta v} \; para \; v > 0 \tag{3.17}$$

onde Γ é a função gama de Euler:

$$\Gamma(x) = \int_0^{\infty} t^{x-1}.e^{-t}dt \tag{3.18}$$

A distribuição binomial negativa resultante é dada por (Long [1997]):

$$Pr(y_i, x_i) = \frac{\Gamma(y_i + v)}{y! \; . \; \Gamma(v)}.\left(\frac{v}{v + \mu_i}\right)^{\mu}.\left(\frac{\mu}{v + \mu_i}\right)^{y_i} \tag{3.19}$$

Para a distribuição negativa, a média condicional da variável dependente é a mesma do MRP, enquanto a variância condicional da variável dependente é quadrática na média μ:

$$var(y|x) = \mu\left(1 + \frac{\mu}{v}\right) \tag{3.20}$$

Já que v é positivo e μ para variáveis de contagem também é positivo, a variância excede a média condicional da distribuição original de Poisson. O termo $\alpha = \frac{1}{v}$ é normalmente referenciado como parâmetro de dispersão, já que o aumento no valor de α causa o aumento do valor da variância condicional de y. Consequentemente, um valor baixo de α representa um nível baixo de super dispersão.

Com um aumento no valor de α a probabilidade de valores iguais a zero na distribuição binomial negativa aumenta. Para um valor de α suficientemente grande, o modo condicional para todos os valores da variável independente é 0. A distribuição binomial negativa corrige três causas de diminuição de desempenho do modelo de Poisson:

1. A variância da variável dependente que segue a distribuição binomial negativa excede a variância correspondente à distribuição de Poisson para uma dada média.

2. Um aumento no valor da variância no modelo binomial negativo resulta em uma probabilidade mais alta de contagens pequenas.

3. A probabilidade de contagens grandes é maior.

3.3.5 Modelo de Regressão Logístico

O modelo de regressão logístico difere das distribuições anteriormente descritas, pois os valores da variável dependente são binários (Hosmer & Lemeshow [2000]). Considerando o modelo de regressão logístico multivariável, tem-se um conjunto de p variáveis, no vetor $x' = (x_1, x_2, ..., x_p)$. A probabilidade condicional é expressa por $P(Y = 1|x) = \pi(x)$. O modelo de regressão logístico multivariável pode ser representado pelas seguintes equações:

$$g(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p \tag{3.21}$$

$$\pi(x) = \frac{e^{g(x)}}{1 + e^{g(x)}} \tag{3.22}$$

Assuma que se tem n observações $(x_i, y_i), i = 1, 2, ..., n$. Para estimar os parâmetros do modelo, $\beta' = (\beta_0, \beta_1, ..., \beta_p)$, utiliza-se o método da máxima verossimilhança (*maximum likehood*), expresso por:

$$\sum_{i=1}^{n} [y_i - \pi(x_i)] = 0 \tag{3.23}$$

$$\sum_{i=1}^{n} x_{ij}[y_i - \pi(x_i)] = 0 \tag{3.24}$$

para $j = 1, 2, ..., p$.

A solução dessas equações é expressa por $\hat{\beta}$. Assim, o valor ajustado para a regressão logística multivariável é $\hat{\pi}(x_i)$, calculado a partir de $\hat{\beta}$ e x_i.

3.4 Validação do Modelo de Predição

A validação é realizada para verificar se o modelo é capaz de realizar a predição em novos dados ou dados que não tenham sido utilizados na construção do modelo (Harrel [2001]). Existem duas estratégias de validação de modelos: interna e externa. Na validação interna, o ajuste e validação do modelo é realizado em um subconjunto dos dados de maior interesse. Em uma abordagem mais rigorosa na validação externa, utiliza-se dois conjuntos de dados completamente diferentes, ou seja, dois projetos de hardware ou software distintos. Em se tratando de projetos de software, esse tipo de validação não alcança bons resultados, pois as especificidades de cada projeto impedem que um único modelo seja eficiente em vários projetos (Nagappan et al. [2006]; Zimmermann et al. [2009]).

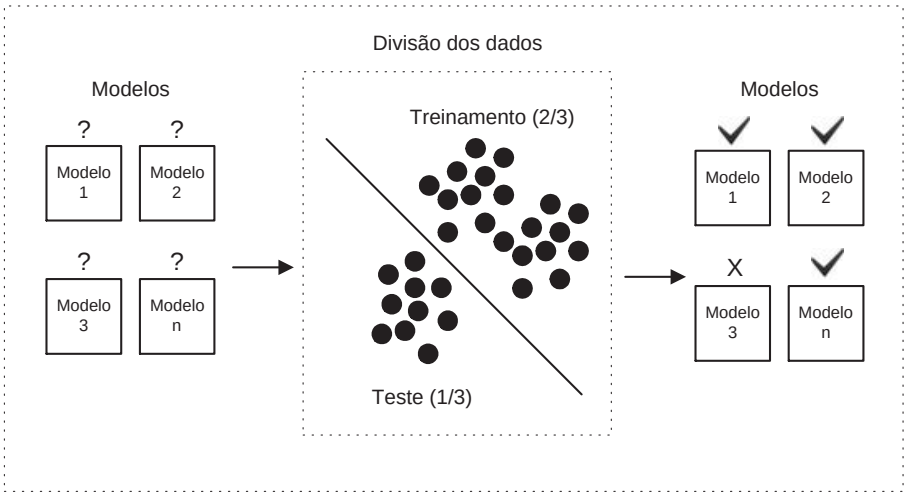

Figura 3.5. Validação do modelo de predição.

Uma estratégia menos rigorosa da validação externa, a divisão de dados (*data splitting*), é apresentada na Figura 3.5. Essa estratégia de validação é muito utilizada na predição de erros de módulos de software (Munson & Khoshgoftaar [1992]; Zimmermann [2008]). Na divisão de dados, é realizada uma partição aleatória dos dados em dois grupos: treinamento e teste. O grupo de treinamento é utilizado para ajustar os valores da regressão. Em seguida, o modelo criado com os dados de treinamento é avaliado utilizando os dados do grupo de teste. Como a divisão dos dados é aleatória, é interessante repetir o procedimento diversas vezes e avaliar a média dos dados gerados.

Para avaliar o poder de predição do modelo, pode-se utilizar o índice R^2 (regressão linear) ou χ^2 (regressão de Poisson ou binomial negativa). Além disso, uma análise gráfica dos dados ajustados e observados também pode ser útil para avaliar a eficiência do modelo. O cálculo do coeficiente de correlação de Spearman entre os dados ajustados e os dados observados permite a comparação do poder de predição de diferentes modelos.

Para analisar os resultados do modelo de regressão logístico, é utilizada a notação de avaliação de classificação binária. A Tabela 3.1 apresenta a matriz de confusão (*confusion matrix*). A matriz de confusão confronta os resultados obtidos no modelo com os resultados reais. Os módulos de baixo risco são não propensos a erros e os módulos de alto risco são propensos a erros.

Tabela 3.1. Notação utilizada na matriz de confusão.

		Resultado do Modelo	
		Baixo Risco	Alto Risco
Resultado Real	Baixo Risco	n_{11}	n_{12}
	Alto Risco	n_{21}	n_{22}

A partir da notação da matriz de confusão é possível calcular a sensibilidade (*sensibility*) do modelo, definida como:

$$s = \frac{n_{22}}{n_{21} + n_{22}} \tag{3.25}$$

A sensibilidade é o percentual de módulos de alto risco que foram corretamente classificados. A especificidade (*specificity*) do modelo, o percentual de

módulos de baixo risco que foram corretamente classificados, é definida como:

$$f = \frac{n_{11}}{n_{11} + n_{12}} \tag{3.26}$$

Para que o modelo tenha um bom desempenho, tanto a sensibilidade quanto a especificidade devem ter valores altos. Um valor baixo de especificidade indica que existem muitos módulos de baixo risco classificados como módulos de alto risco. Já um valor baixo de sensibilidade indica que existem muitos módulos de alto risco classificados como módulos de baixo risco. O coeficiente de Youden [1950] é utilizado para combinar os índices s e f e, dessa forma, apresentar uma medida de desempenho de modelos de classificação binários (Emam et al. [2001a]). O coeficiente de Youden é definido como:

$$J = s + f - 1 \tag{3.27}$$

O coeficiente J pode variar de -1 a $+1$, sendo que $+1$ é o melhor resultado e, -1, o pior. Um modelo que classifica como alto/baixo risco com probabilidade de $0,5$ conduziria a um valor de J igual a 0. Assim sendo, valores de J maiores que 0, apresentam um desempenho melhor do que um classificador com probabilidade de acerto de $0,5$.

3.5 Considerações Finais

Este Capítulo apresentou técnicas estatísticas utilizadas em diversos trabalhos de engenharia de software. Para a realização dos experimentos, são utilizadas algumas dessas técnicas. A escolhas de quais técnicas utilizar é feita com base nos critérios de adequação dos dados à técnica, maior utilização da técnica em outros trabalhos, e de simplicidade da técnica. Dessa forma, são utilizadas as seguintes técnicas:

- Análise de correlação das métricas com erros: coeficiente de correlação de Spearman;
- Construção e ajuste do modelo de predição: modelos de regressão linear, de Poisson, binomial negativa e logístico;

- Validação do modelo de predição: divisão de dados, coeficiente R^2 (regressão linear), coeficiente χ^2 (regressão de Poisson e binomial negativa) e coeficiente de Youden (regressão logística).

Capítulo 4

Ambiente Experimental: Métricas, Dados e Ferramentas

O primeiro desafio deste trabalho é ter acesso a dados de projeto de circuitos integrados. O ideal seria utilizar dados de projetos comerciais mas, infelizmente, o acesso ao repositório de desenvolvimento de circuitos integrados é considerado pela indústria como altamente confidencial. Ao longo do desenvolvimento do trabalho foram realizadas diversas tentativas nesse sentido, sendo que, em nenhuma delas o acesso aos dados foi autorizado. Portanto, são utilizados projetos de circuitos integrados desenvolvidos por alunos e repositórios de código aberto. Projetos desenvolvidos por alunos já foram utilizados em diversos trabalhos (Campenhout et al. [2000]; Velev [2003]; Campenhout et al. [1998]), assim como os projetos de código aberto (Constantinides et al. [2008]). A seguir, são apresentados os 3 componentes principais do ambiente experimental: métricas, dados e ferramentas.

4.1 Métricas

A primeira métrica utilizada no trabalho é gerada por uma ferramenta chamada *HDL Complexity Tool* - HCT (Maurer [2009]). A ferramenta processa arquivos Verilog e extrai a métrica CCM (Complexidade Ciclomática de McCabe). Essa métrica de complexidade foi proposta por McCabe [1976], com o objetivo de estimar a legibilidade e a testabilidade de um software. A complexidade ciclomá-

tica pode ser definida como o número de caminhos de execução independentes de um programa.

O segundo conjunto de métricas é extraído de uma ferramenta comercial de verificação de circuitos digitais, o JasperGold™ (Jasper Design Automation, Inc. [2010]). O comando `get_design_info` extrai as seguintes métricas:

- Linhas de comentários (`JG_COMMENT_LINES`);
- Flip Flops do módulo após a síntese (`JG_FLOPS`);
- Latches do módulo após a síntese (`JG_LATCHES`);
- Portas lógicas do módulo após a síntese (`JG_GATES`);
- Sinais do módulo (`JG_NETS`);
- Portas de E/S do módulo (`JG_PORTS`);
- Linhas do código fonte do módulo (`JG_RTL_LINES`);
- Outros módulos instanciados pelo módulo (`JG_RTL_INSTANCES`).

Finalmente, o último conjunto de métricas é extraído por um *parser* Verilog de código aberto, o VParser (Snyder [2010]). As métricas geradas pelo VParser são contagens de ocorrências de: palavras-chave (`VP_Keywords`), operadores (`VP_Operators`), símbolos (`VP_Symbols`), comentários (`VP_Comments`), atributos (`VP_Attributes`), números (`VP_Numbers`) e cadeias de caracteres (`VP_Strings`). As palavras-chave são palavras reservadas da linguagem Verilog e, muitas delas, podem estar correlacionadas à complexidade do módulo. Assim, optou-se por estudar algumas palavras-chave específicas, que são apresentadas na Tabela 4.1. Os operadores indicam operações lógicas e aritméticas. Símbolos são os nomes utilizados para referenciar os sinais no código fonte. Atributos e números são valores numéricos utilizados no código fonte.

Tabela 4.1. Palavras-chave utilizadas como métricas.

endmodule	input	module	output	inout
reg	always	begin	end	if
initial	negedge	posedge	wire	case
default	else	endcase	or	assign
for	integer	while	function	do
bit	final	forever	casez	casex

4.2 Dados

4.2.1 Projetos Desenvolvidos por Alunos

Os projetos desenvolvidos por alunos foram implementados na forma de trabalhos práticos no decorrer das disciplinas "Arquitetura de Computadores" (Mestrado/Doutorado) e "Organização de Computadores II" (Graduação), ofertadas pelo Departamento de Ciência da Computação da UFMG e consistem na implementação de um subconjunto de instruções do processador RISC MIPS32 (MIPS Technologies [2005]).

4.2.1.1 Histórico

Foram realizadas diversas tentativas de utilização de dados de projeto de alunos até que a especificação do trabalho prático atingisse o grau de maturidade adequado. As principais características das especificações dessas tentativas, ao longo de três semestres são descritas abaixo:

- **2^o semestre de 2007**: Utilizou-se a ferramenta de gerenciamento de versão CVS. A especificação do trabalho era composta pelo documento de descrição do conjunto de instruções do MIPS (MIPS Technologies [2005]) e pelos livros adotados no curso (David A. Patterson, John L. Hennessy [2005]; Hennessy & Patterson [2003]). Cada grupo era responsável por projetar o caminho de dados, implementá-lo e realizar os testes. Os erros foram reportados preenchendo um formulário em um arquivo texto. Apesar de esta ser a metodologia tradicionalmente adotada, tinha-se como resultado implementações completamente distintas e, na maioria das vezes, ineficientes. Dessa

forma era muito difícil comparar as implementações de diferentes grupos. Outra desvantagem dessa abordagem é a dificuldade de processar automaticamente os erros relatados e a extração das métricas dos módulos.

- **1^o semestre de 2008**: A ferramenta de gerenciamento de erros Bugzilla (The Bugzilla Team [2008]) foi utilizada. Apesar das informações sobre os erros poderem ser automaticamente recuperadas, infelizmente, devido à complexidade desta ferramenta, os alunos foram incapazes de cadastrar os erros de projeto de maneira satisfatória.

- **2^o semestre de 2008**: A partir desse semestre a ferramenta de controle de versão utilizada passou a ser o SVN. As informações sobre os erros foram anexadas aos comentários de `commit` do repositório. Para tanto, foi criada uma linguagem de descrição de erros e ferramentas de auxílio ao preenchimento de relatórios de erros.

- **2^o semestre de 2009**: Foram realizados treze trabalhos no total, sendo que cinco são utilizados na Seção de Resultados.

4.2.1.2 Especificação do Trabalho Prático

Após a realização dos trabalhos práticos descritos anteriormente, ficou definido que seria fornecido aos estudantes uma especificação do processador MIPS. Assim sendo é possível comparar as implementações de diferentes grupos que normalmente são compostos por 4 ou 5 alunos. A especificação do processador MIPS é dividida em 17 módulos e descreve as interfaces e os respectivos comportamentos dos sinais de cada módulo. A seguir é apresentada uma breve descrição de cada um dos módulos:

1. Unidade Lógico Aritmética (ULA): agrupa as operações aritméticas soma e subtração, e lógicas, Negação, E, Ou, Não ou e Ou Exclusivo.

2. Unidade de Comparação (Comparador): utilizada para resolução de instruções de desvio.

3. Unidade de Controle (Controle): responsável por gerar a maior parte dos sinais que configuram o processamento de uma instrução em seu caminho de dados.

4. Cache de Dados: do tipo mapeamento direto e de tamanho 64 kB. Possui opção de escrita na cache e a atualização no módulo Memória RAM. A política de atualização é *write-back* e *write-allocate*.

5. Unidade de Decodificação de Instruções (Decodificação): Nesse estágio do *pipeline* são recuperados os registradores, gerados os sinais de controle e calculados os desvios para instruções do tipo *branch* e *jump*. Nesse módulo são instanciadas a Unidade de Comparação e a Unidade de Controle.

6. Unidade de Execução (Execução): nesse estágio do *pipeline* são realizadas as operações de instruções lógico aritméticas e, também, o cálculo de endereços utilizados em *loads* e *stores*. Esta implementação do MIPS possui apenas instruções inteiras de 32 bits. Não são implementadas instruções de multiplicação. Neste módulo são instanciadas a Unidade Lógico Aritmética e a Unidade de Deslocamento.

7. Unidade de Busca de Instruções (Busca): responsável pelo estágio de busca do *pipeline*.

8. Unidade de Repasse (Repasse): resolve os conflitos de dependências de dados dos estágios do *pipeline*.

9. Unidade de Gerenciamento de Cache (Geren. Cache): interliga os estágios de busca de instruções e acesso a dados do MIPS às caches de instruções e dados. Instancia a Cache de Dados e a Cache de Instruções.

10. Unidade de Gerenciamento de Memória (Geren. Memória): responsável pela interface entre os sinais do microprocessador e a memória. Todas as requisições de leitura e escrita do processador ao bloco de memória externo passam pela unidade de gerenciamento de memória.

11. Cache de Instruções: do tipo mapeamento direto e de tamanho 64 kB.

12. Unidade de Memória (Memória): estágio do *pipeline* no qual são realizadas as instruções de leitura (*load*) escrita (*store*) em memória. O acesso é intermediado pela Unidade de Gerenciamento de Memória.

13. Unidade Principal do MIPS (MIPS): neste módulo são instanciados os estágios de *pipeline* e unidades auxiliares como registradores e lógica de repasse.

14. Memória RAM: composta por um bloco comum para instruções e dados. Nesta implementação são utilizados 2^{16} blocos de 8 bits, permitindo o endereçamento de 64 kB de memória.

15. Banco de Registradores (Registradores): composto por 32 registradores de 32 bits. Esses registradores podem ser acessados por 3 barramentos distintos: 2 de leitura e 1 de escrita.

16. Unidade de Deslocamento (Deslocamento): unidade capaz de efetuar deslocamentos de dados em até 32 bits.

17. Unidade de Escrita (Escrita): estágio do *pipeline* responsável pelo armazenamento dos resultados no Banco de Registradores.

No Apêndice B, Subseção B.1 é apresentada, como exemplo, a especificação do módulo ULA entregue aos estudantes. A Subseção B.2 ilustra um exemplo de implementação em Verilog da especificação da ULA.

De forma a distribuir o tempo de desenvolvimento do projeto de maneira mais homogênea, os módulos foram agrupados e prazos de entrega foram definidos conforme pode-se observar na Tabela 4.2. O tempo total dos prazos fixados é de 15 semanas.

Além das especificações, são fornecidos testes para cada um dos módulos implementados. Os testes instanciam os módulos, aplicam estímulos às entradas e verificam se as saídas se comportam conforme esperado. Para automatizar o processo e também para que os alunos não tenham acesso ao conjunto de testes utilizado, um sistema de correção automático é utilizado. Esse sistema, apresentado na Figura 4.1, permite que os alunos enviem as implementações e visualizem o resultado dos testes. A Subseção B.3 do Apêndice B apresenta um exemplo de teste da ULA.

Tabela 4.2. Prazos e módulos de cada entrega.

Entrega	Módulos	Prazo
1	ULA, Registradores, Memória RAM	2 semanas
2	Deslocamento, Comparador	1 semana
3	Geren. Memória	1 semana
4	Busca	1 semana
5	Controle	1 semana
6	Decodificação, Repasse	2 semanas
7	Execução, Memória, Escrita	2 semanas
8	MIPS	2 semanas
9	Cache Dados, Cache Instruções	1 semana
10	Geren. Cache	2 semanas

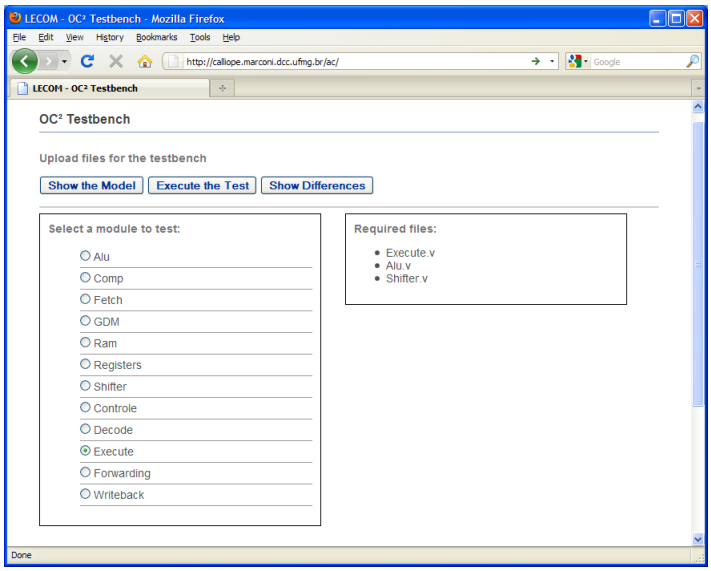

Figura 4.1. Sistema automático de testes.

4.2.1.3 Relatórios de Erros

Ao longo do desenvolvimento do processador, os alunos devem relatar os erros identificados. Para automatizar o processo de recuperação das informações de erros foi desenvolvida a Bug Language. A Bug Language é uma linguagem desenvolvida para descrever erros em projetos de circuitos integrados, marcando explicitamente os módulos e sinais envolvidos. Essa linguagem foi desenvolvida com o objetivo de ser facilmente processada, além de natural para ser utilizada

pelo desenvolvedor, sem o auxílio de ferramentas externas. Toda informação é anexada à revisão durante o procedimento de `commit`, tornando menos provável que o usuário esqueça de preencher o relatório de erros ou até mesmo que detalhes sobre este evento sejam perdidos. Neste sentido, a informação armazenada se torna mais precisa e confiável, sendo capaz de fornecer estatísticas de erros de cada módulo.

Todos os arquivos envolvidos no `commit` devem possuir o seu próprio bloco de descrição. Um bloco de descrição possui as seguintes informações:

- Nome do arquivo;
- Nome do módulo;
- Motivação do `commit`:
  - Como o erro foi identificado (apenas quando a motivação for correção de erro - `bugfix`);
  - Sinais envolvidos no erro (apenas quando a motivação for correção de erro - `bugfix`).
- Comentários.

A Figura 4.2 apresenta uma visão geral da linguagem com os possíveis blocos que podem ser utilizados para descrever um `commit`. O nome do arquivo é especificado pela palavra reservada `@file`. Esta palavra reservada é utilizada para definir o arquivo ao qual o bloco de descrição se refere. Caso um arquivo não seja código-fonte HDL, pode-se inibir o preenchimento do bloco de descrição da alteração. O módulo no qual a modificação foi realizada é especificado pela palavra reservada `@module`. Essa palavra reservada é utilizada para diferenciar entre módulos no mesmo arquivo, aumentando assim a granularidade para recuperação de estatísticas. Desta forma, um único arquivo pode conter vários blocos de descrição de alteração.

A motivação do `commit` é descrita pela palavra reservada `@motivation`, além de uma das seguintes razões:

- `resume`: Continuação do projeto;

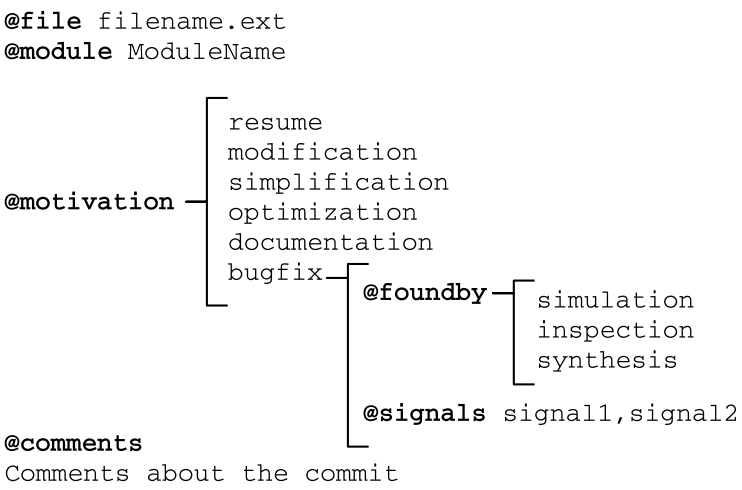

Figura 4.2. Visão geral da Bug Language.

- `modification`: Modificação do projeto;
- `simplification`: Simplificação no projeto;
- `optimization`: Otimização do projeto;
- `documentation`: Geração ou atualização de documentação;
- `bugfix`: Correção de erro.

No caso da motivação ser do tipo `bugfix`, duas outras palavras reservadas devem ser utilizadas:

- `@foundby`: Descreve como o erro foi identificado, dentre as opções: por simulação (`simulation`), inspeção do código (`inspection`) ou durante a síntese (`synthesis`);
- `@signals`: Esta é uma palavra reservada opcional e deve ser utilizada para descrever os sinais envolvidos no erro.

O palavra reservada `@comments` é utilizada para comentários em linguagem natural que não serão processados.

4.2.2 Projetos de Código Aberto

Atualmente, existem diversos repositórios de projetos de circuitos integrados de domínio público. O site `opencores.org` (Opencores [2011]) armazena vários projetos de processadores e núcleos de protocolos de comunicação, co-processamento, criptografia, dentre outros. Entretanto, a utilização das informações dos projetos disponibilizados no `opencores.org` apresenta diversos obstáculos. A maioria dos projetos possui poucas revisões, pois normalmente o projeto é implementado por poucos desenvolvedores e apenas disponibilizado no site. Além disso, dificilmente são encontradas informações sobre a ocorrência de erros nos módulos, o que inviabiliza sua utilização nesse trabalho.

O projeto do processador OpenSPARC, da Sun Microsystems (adquirida pela Oracle em 2009), foi disponibilizado ao domínio público em 2008. Apesar de possuir poucas revisões disponíveis, e não utilizar uma ferramenta de gerenciamento de erros automática, alguns módulos contém erros comentados (Constantinides et al. [2008]). Assim sendo, é possível obter a informação de quantos erros foram corrigidos em cada um desses módulos.

O OpenSPARC T1 é um multiprocessador em um único chip. Cada processador OpenSPARC T1 contém oito núcleos físicos de processamento SPARC. Cada processador físico SPARC possui suporte de hardware para quatro processadores virtuais. Esses quatro processadores virtuais executam instruções concorrentemente em um escalonamento circular.

Cada núcleo físico SPARC possui: cache de instruções de 16 Kbytes, cache de dados de 8 Kbytes, TLB (*Translation Lookaside Buffer*) de dados de 64 entradas e TLB de instruções de 64 entradas. Os oito núcleos SPARC são conectados a uma cache L2 de 3 MBytes e, em seguida, a um controlador de memória RAM dinâmica DDR2. O *pipeline* de cada um dos núcleos é composto por seis estágios: Busca, Alternância, Decodificação, Execução, Memória e Escrita de Dados.

4.3 Ferramentas

Para implementar o processo descrito no Capítulo 3 (Metodologia) diversas ferramentas foram desenvolvidas ou utilizadas. Para auxiliar na utilização da Bug

Language, duas ferramentas, Bug Reporter e Bug Ruler (Subseção 4.3.1) foram disponibilizadas aos alunos durante o desenvolvimento do processador MIPS. A ferramenta EyesOn (Subseção 4.3.2) foi desenvolvida para realizar a extração das métricas de produto e processo do sistema de controle de versões e do sistema de rastreamento de erros. Finalmente, a ferramenta *R* (Crawley [2007]), utilizada para construir e validar os modelos de predição é apresentada na Subseção 4.3.3.

4.3.1 Bug Reporter e Bug Ruler

Quanto maior é a quantidade de texto a ser digitada pelo desenvolvedor, maior é a resistência de preencher o relatório de erros. As ferramentas Bug Reporter e Bug Ruler foram desenvolvidas para auxiliar o desenvolvedor a preencher as informações sobre os erros, em Bug Language, diminuindo, consideravelmente, a quantidade de texto a ser digitada.

O Bug Reporter foi implementado em Java por questões de compatibilidade. Essa ferramenta funciona como um editor de texto chamado assim que o `commit` é efetuado. A interface gráfica do Bug Reporter é apresentada na Figura 4.3.

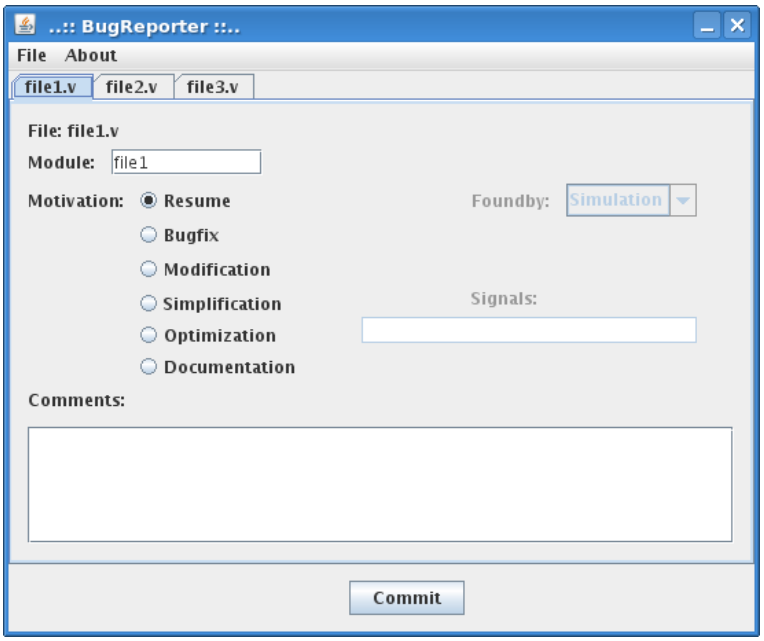

Figura 4.3. Interface gráfica do Bug Reporter.

Após ser chamado pelo evento de `commit`, o Bug Reporter lê e processa os arquivos gerados pelo SVN, identificando quais arquivos foram modificados. A interface gráfica é então apresentada com algumas informações já preenchidas. A utilização desta ferramenta reduz drasticamente a quantidade de texto digitada, sendo necessário que o desenvolvedor digite apenas os sinais e comentários. Assim que o formulário é preenchido, o arquivo é salvo e enviado como comentário do `commit`. Caso o terminal utilizado pelo desenvolvedor não tenha acesso ao modo gráfico X, o Bug Reporter inicializa uma versão em modo texto do programa e pergunta pela informação necessária. O Bug Ruler possui uma interface gráfica similar ao Bug Reporter, sendo que a única diferença é que, ao invés de efetuar o `commit`, o Bug Ruler apenas gera o texto, em Bug Language, que deve ser incluído no comentário do `commit`. Esta ferramenta é muito útil, quando o desenvolvedor utiliza clientes gráficos que não permitem a utilização do Bug Reporter.

4.3.2 EyesOn

A ferramenta EyesOn foi desenvolvida para extrair, armazenar e visualizar métricas de produto e processo de projetos HDL. Atualmente, a ferramenta EyesOn é capaz de extrair informações de erros escritas em Bug Language e armazenadas em `commits`. Além disso, várias métricas de complexidade podem ser extraídas, como, por exemplo, complexidade ciclomática, linhas de código, número de flip-flops, latches, instâncias, etc. As métricas de complexidade extraídas pela ferramenta EyesOn são descritas na Seção 4.1. Para cada revisão armazenada no sistema de controle de versão, a ferramenta extrai todo o conjunto de métricas de produto e processo. Em seguida, essas informações são armazenadas em um banco de dados MySQL que pode ser facilmente acessado por outras ferramentas.

A Figura 4.4 apresenta dois cenários de utilização da ferramenta EyesOn. No primeiro cenário, a evolução das métricas no decorrer do tempo pode ser visualizada em diversos formatos, como, por exemplo: gráficos de linha e barra, gráficos de radar e diagramas de calor (*heatmap*). O segundo cenário é o de maior relevância para esse trabalho, pois as métricas são exportadas para serem utilizadas em outras ferramentas. Maiores detalhes sobre a ferramenta EyesOn podem ser encontrados em Nacif et al. [2011b].

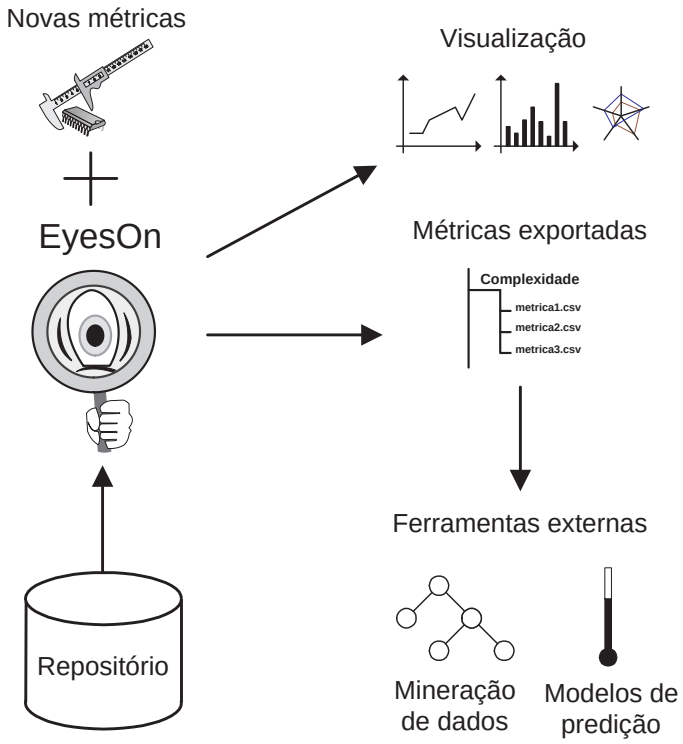

Figura 4.4. Cenários de utilização da ferramenta EyesOn.

4.3.3 R

O projeto R (Crawley [2007]) é um sistema aberto para computação estatística e visualização gráfica. O ambiente de trabalho do R é composto por: interpretador da linguagem, depurador e acesso a diversas funções. O interpretador do ambiente R é capaz de executar *scripts*. A principal vantagem de se utilizar o R nesse trabalho é que essa ferramenta possui um grande número de técnicas estatísticas já implementadas, como, por exemplo: modelos de regressão lineares e não lineares, análise de séries temporais, testes estatísticos clássicos (paramétricos e não paramétricos), e também funções básicas como média, desvio padrão, cálculo de coeficientes de correlação. Além disso, o R é capaz de gerar gráficos de pontos, linhas, histogramas, etc. O R possui também a funcionalidade de acesso a bancos de dados MySQL através do pacote RMySQL. Dessa forma, os dados gerados pela ferramenta EyesOn podem ser facilmente acessados.

Capítulo 5

Resultados

Neste Capítulo são apresentados dois estudos de caso. O primeiro estudo de caso utiliza métricas de produto e processo em dados de processadores MIPS desenvolvidos por alunos. O segundo estudo de caso utiliza métricas de produto nos dados do processador OpenSPARC, que é um projeto de código aberto.

5.1 Estudo de Caso 1: MIPS

O primeiro estudo de caso utiliza os dados dos processadores MIPS, desenvolvidos por alunos e descritos na Seção 4.2.1. Apesar de todos os alunos terem sido orientados a relatar erros, e o cumprimento dessa regra ter sido levado em consideração na avaliação, infelizmente muitos grupos não relataram os erros satisfatoriamente. Dessa forma, para realizar esse estudo de caso foram escolhidos os 5 projetos que relataram o maior número de erros ao longo do semestre. Como esses projetos foram desenvolvidos em um ambiente controlado, é possível ter acesso às métricas de número de `commits` e erros. Os valores finais dessas métricas para os 5 projetos selecionados são apresentados na Figura 5.1.

5.1.1 Extração das Métricas

Inicialmente deve-se extrair as métricas de cada um dos módulos do projeto. Nesse estudo de caso, o número total de módulos é 85, pois cada um dos 5 projetos

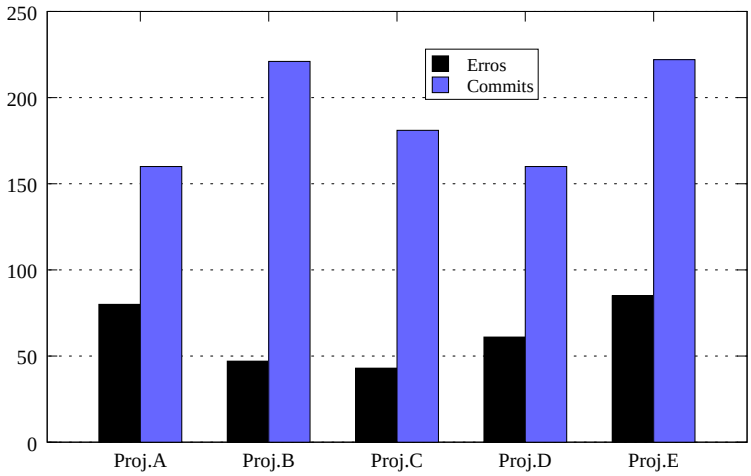

Figura 5.1. Erros e commits por projeto.

possui 17 módulos. Conforme descrito na Seção 4.3.2, a ferramenta EyesOn utiliza 3 ferramentas externas para extrair as métricas. As métricas geradas pelo HCT e pelo VParser não necessitam que os módulos passem pelo processo de síntese. Ao contrário, as métricas geradas pelo JasperGold™ são extraídas após o processo de síntese dos módulos. Infelizmente nem todos os módulos dos projetos desenvolvidos pelos alunos podem ser sintetizados e as métricas extraídas pelo JasperGold™ para esses módulos não podem ser geradas. A maioria dos problemas de síntese são em módulos de memória.

A Tabela 5.1 apresenta o resultado da síntese para os projetos A, B, C, D e E. Os módulos que passaram pelo processo de síntese e que, consequentemente, possuem as métricas extraídas pelo JasperGold™ foram marcados com o símbolo ✓e são numerados de 1 a 44. De forma a simplificar as análises estatísticas dos próximos passos, optou-se por considerar apenas esses módulos.

A Tabela 5.2 apresenta os valores mínimos, máximos, médios e desvios-padrão das métricas extraídas dos 44 módulos. Os nomes das métricas possuiem os prefixos HCT, JG e VP para as métricas geradas, respectivamente, pelas ferramentas HCT, JasperGold™ e Verilog Parser. No total foram estudadas 47 métricas, sendo uma gerada pela ferramenta HCT, 8 pelo JasperGold™, 37 pelo VParser e, finalmente, o número de `commits` de cada módulo. As métricas `VP_while`, `VP_function`, `VP_do`, `VP_bit`, `VP_final`, `VP_forever`, `VP_casez` e `VP_casex` não

Tabela 5.1. Resultado da síntese dos módulos dos Projetos A, B, C, D e E.

Módulo	**Proj. A**	**Proj. B**	**Proj. C**	**Proj. D**	**Proj. 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)

possuem ocorrências e, portanto, não são apresentadas na Tabela 5.2.

A métrica `HCT_MCCABE` varia pouco. As principais métricas de complexidade extraídas pelo JasperGold™ apresentam uma grande variação. Exemplos dessas métricas são `JG_GATES`, `JG_NETS`, `JG_FLOPS` e, `JG_RTL_LINES`. Essa variação se deve às diferentes complexidades dos módulos. A métrica `VP_Keyword` apresenta o número de palavras-chave diferentes em cada módulo. As métricas `VP_module` e `VP_endmodule` possuem apenas uma ocorrência, pois cada arquivo Verilog possui um único módulo.

5.1.2 Análise de Correlação das Métricas com os Erros

Nessa etapa é realizado o cálculo dos coeficientes de correlação entre os erros e cada uma das métricas. Assim, é possível escolher um subconjunto de métricas que possua maior correlação com os erros e utilizá-lo nos modelos. Tanto as métricas, quanto os erros não seguem a distribuição normal e, dessa forma, não é possível se utilizar o coeficiente de correlação de Pearson. Assim, foi utilizado o coeficiente

Tabela 5.2. Estatística descritiva das métricas extraídas do conjunto de 44 módulos selecionados do MIPS.

Métrica	Min	Max	Média	Desvio-padrão
HCT_MCCABE	1	5	1,09	0,61
JG_COMMENT_LINES	0	4	0,09	0,61
JG_FLOPS	0	2.112	239,47	483,72
JG_LATCHES	0	33	1,53	7,03
JG_GATES	38	3.8604	5.442,14	7.203,72
JG_NETS	19	5.353	857,16	1.383,88
JG_PORTS	4	45	16,56	12,26
JG_RTL_LINES	21	2.000	233,44	401,25
JG_RTL_INSTANCES	0	7	0,40	1,22
VP_Keywords	5	18	13,33	3,39
VP_Operators	7	20	13,56	3,53
VP_Symbols	5	105	27,53	27,16
VP_Comments	0	45	9,72	10,75
VP_Attributes	3	115	21,09	25,89
VP_Numbers	3	115	21,09	25,89
VP_Strings	0	7	0,40	1,22
VP_endmodule	1	1	1	0
VP_input	3	23	7,37	4,77
VP_module	1	1	1	0
VP_output	1	36	9,07	10,52
VP_inout	0	1	0,12	0,32
VP_reg	0	26	5,63	7,09
VP_always	0	3	1,23	0,72
VP_begin	0	84	6,49	12,91
VP_end	0	84	6,49	12,91
VP_if	0	18	3,09	4,32
VP_initial	0	1	0,02	0,15
VP_posedge	0	5	0,95	1,17
VP_negedge	0	1	0,49	0,51
VP_wire	0	43	4,28	9,81
VP_case	0	4	0,81	1,07
VP_default	0	4	0,51	0,94
VP_else	0	14	2,19	3,25
VP_endcase	0	4	0,81	1,07
VP_or	0	5	1,33	1,07
VP_assign	0	83	8,05	14,31
VP_for	0	2	0,19	0,45
VP_integer	0	1	0,14	0,35
Commits	2	26	10,65	6,41

de correlação de Spearman, que pode ser empregado em dados que não seguem a distribuição normal.

A Tabela 5.3 apresenta os coeficientes de correlação de Spearman das métricas com os erros observados nos módulos ordenados da maior correlação para a menor. As métricas dentro do intervalo compreendido entre 1 e 15 apresentam correlações com significância maior que 99%.

Pode-se observar que a métrica que possui maior correlação com os erros é o número de `commits` realizados no módulo, ou seja, o número de modificações

Tabela 5.3. Correlações entre as métricas e o número de erros nos módulos do MIPS.

Número	Métrica	Correlação Spearman
1	Commits	0,849
2	VP_negedge	0,592
3	JG_PORTS	0,537
4	VP_Symbols	0,514
5	VP_input	0,494
6	VP_Keywords	0,470
7	VP_Comments	0,453
8	JG_RTL_LINES	0,441
9	JG_NETS	0,440
10	VP_wire	0,419
11	VP_output	0,380
12	JG_GATES	0,424
13	VP_assign	0,370
14	VP_Operators	0,389
15	VP_else	0,367
16	VP_output	0,366
17	VP_if	0,360
18	VP_assign	0,350
19	VP_begin	0,334
20	VP_end	0,334
21	VP_posedge	0,323
22	VP_reg	0,311
23	JG_FLOPS	0,310
24	VP_inout	0,250
25	JG_RTL_INSTANCES	0,238
26	VP_Strings	0,238
27	VP_always	0,191
28	VP_case	-0,176
29	VP_endcase	-0,176
30	VP_default	-0,146
31	VP_or	0,129
32	VP_for	0,084
33	JG_LATCHES	-0,069
34	HCT_MCCABE	-0,048
35	JG_COMMENT_LINES	-0,048
36	VP_initial	-0,048
37	VP_integer	-0,037

sofridas pelo módulo está fortemente correlacionado com o número de erros. Em seguida, tem-se o número de ocorrências da palavra-chave `neg_edge`, que na linguagem Verilog denota sincronização na borda de descida. Essa correlação pode ser explicada pela maior complexidade de sincronização, pois os módulos do MIPS que utilizam sincronização na borda de descida, também o fazem na borda de subida. As métricas `JG_PORTS`, `VP_input` e `VP_output` indicam complexidade de E/S.

5.1.3 Construção e Validação do Modelo

Os modelos são construídos e validados com a ajuda do software R. Neste estudo de caso, a variável dependente do modelo de regressão é o número de erros em cada módulo. A Figura 5.2 apresenta o histograma desses dados. Pode-se observar que os dados da variável dependente não seguem a distribuição normal, sendo que a maioria dos módulos possui poucos ou nenhum erro. Apenas 2 módulos possuem mais que 10 erros.

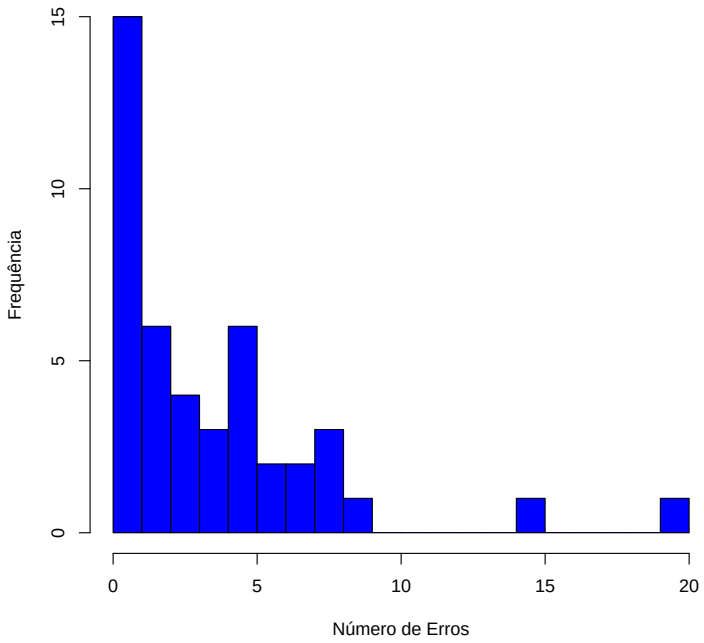

Figura 5.2. Histograma dos erros dos módulos selecionados do MIPS.

Este estudo de caso apresenta os resultados dos modelos de regressão Linear, Poisson, Negativo Binomial e Logístico. Para cada técnica de regressão são desenvolvidos 4 modelos com 37, 15, 10 e 5 métricas, respectivamente. Para formar esses subconjuntos de métricas são escolhidas as que têm maior correlação com o número de erros. Assim, é possível comparar o desempenho de uma mesma classe de modelo, utilizando diferentes números de métricas.

São calculados para cada modelo os componentes principais e selecionados os componentes que têm significância maior que 99% (McCullagh & Nelder [1989]). A única exceção é o modelo de regressão lógistico no qual são mantidos componentes principais com significância maior que 95%.

Para a validação dos modelos é utilizada a técnica de divisão de dados, na qual os dados são agrupados aleatoriamente em dois conjuntos: treinamento e teste. O conjunto de treinamento é composto por $\frac{2}{3}$ dos módulos (29) e o conjunto de teste contém $\frac{1}{3}$ dos módulos (15). Assim, o modelo é ajustado com os dados de treinamento e seu desempenho é avaliado com os dados de teste. A divisão aleatória dos dados é repetida 100 vezes e são apresentados os valores médios.

5.1.3.1 Modelo Linear

O desempenho do modelo de regressão linear é apresentado na Tabela 5.4. O coeficiente de determinação, R^2, mede a qualidade do modelo em relação à sua habilidade de estimar corretamente os valores da variável dependente (número de erros). O R^2 indica quanto da variância da variável dependente é explicada pela variância das variáveis independentes (métricas). Seu valor está no intervalo de 0 a 1: Quanto maior, mais explicativo é o modelo. Além do coeficiente de determinação, o coeficiente de correlação de Spearman também é calculado entre os valores observados e ajustados. Pode-se observar que apesar do valor de R^2 não mostrar bons resultados, o coeficiente de correlação de Spearman retorna valores próximos a $0,8$ para os modelos que possuem 5 e 15 métricas. O valores médios da validação são similares aos valores do modelo que utiliza todas as observações.

Tabela 5.4. Desempenho dos modelos lineares no MIPS.

Modelo	Todos os dados		Validação	
	Cor. Spearman	R^2	Cor. Spearman	R^2
37 métricas	0,629	0,633	0,590	0,633
15 métricas	0,818	0,688	0,765	0,688
10 métricas	0,726	0,496	0,666	0,496
05 métricas	0,840	0,599	0,789	0,599

A Figura 5.3 apresenta um gráfico de dispersão dos resultados observados (eixo Y) e dos valores ajustados pelo modelo (eixo X). Quanto mais próximos da

reta os pontos estiverem, melhor é o desempenho do modelo. Pode-se observar que para os módulos com maior quantidade de erros, o modelo linear retornou valores bem menores de erros.

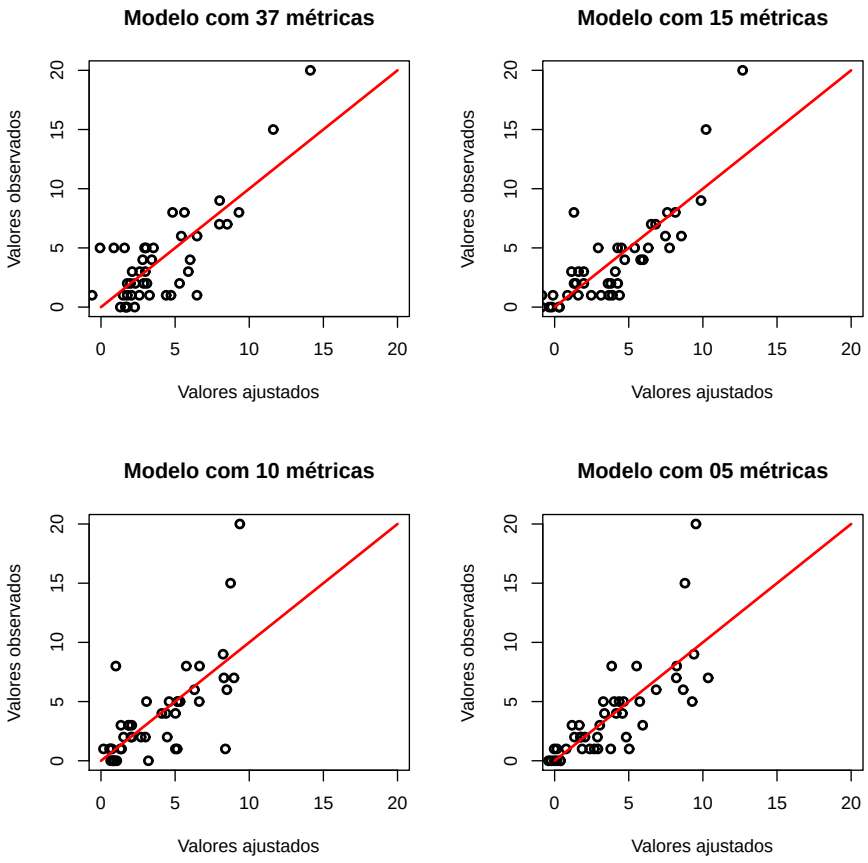

Figura 5.3. Comparação dos resultados dos modelos lineares com os valores observados nos módulos do MIPS.

Os quatro modelos obtiveram resultados semelhantes. Assim sendo, bastariam as 5 métricas mais correlacionadas com os erros (`Commits`, `VP_negedge`, `JG_PORTS`, `VP_Symbols` e `VP_input`) para o modelo de regressão linear. Os valores baixos para R^2 sugerem que o modelo linear não seria o mais adequado realizar a regressão nesses dados.

5.1.3.2 Modelo de Poisson

A Tabela 5.5 apresenta os resultados do modelo de regressão de Poisson. Para avaliar o desempenho do modelo de regressão de Poisson, o coeficiente χ^2 é calculado. O coeficiente χ^2 mede se os dados que estão sendo analisados possuem uma distribuição similar a amostras aleatórias da distribuição de Poisson. Valores acima de $0,05$ são considerados bons. Além do coeficiente χ^2 são calculados os coeficientes de correlação de Spearman entre os dados observados e ajustados.

Considerando a utilização de todos os dados, os coeficientes de correlação de Spearman dos modelos apresentam valores muito bons, acima de $0,8$. Esses valores caem um pouco quando os modelos utilizam os dados de validação, mas o modelo com 5 métricas permanece com o valor superior a $0,8$. Os coeficientes χ^2 apresentam bons valores tanto para os modelos que utilizam todos os dados quanto para os que utilizam os dados de validação. É interessante observar que os modelos que utilizam os dados de validação apresentam um χ^2 maior.

Tabela 5.5. Desempenho dos modelos de Poisson no MIPS.

Modelo	Todos os dados		Validação	
	Cor. Spearman	χ^2	Cor. Spearman	χ^2
37 métricas	0,831	0,336	0,741	0,390
15 métricas	0,823	0,268	0,781	0,363
10 métricas	0,812	0,227	0,728	0,325
05 métricas	0,847	0,205	0,818	0,323

A Figura 5.4 apresenta um gráfico de dispersão comparando os dados gerados pelo modelo com os dados observados. Pode-se observar que a distribuição de Poisson é mais adequada para modelar o comportamento da variável dependente (número de erros). Os quatro modelos apresentaram resultados similares e, portanto, a utilização do modelo com apenas 5 métricas também se mostra viável com o modelo de regressão de Poisson.

5.1.3.3 Modelo Binomial Negativo

A Tabela 5.6 apresenta o desempenho dos modelos binomiais negativos. Por se tratar de uma variação da distribuição de Poisson, o índice χ^2 é utilizado. Dentre os 4 modelos, o modelo com 5 métricas alcança os melhores resultados.

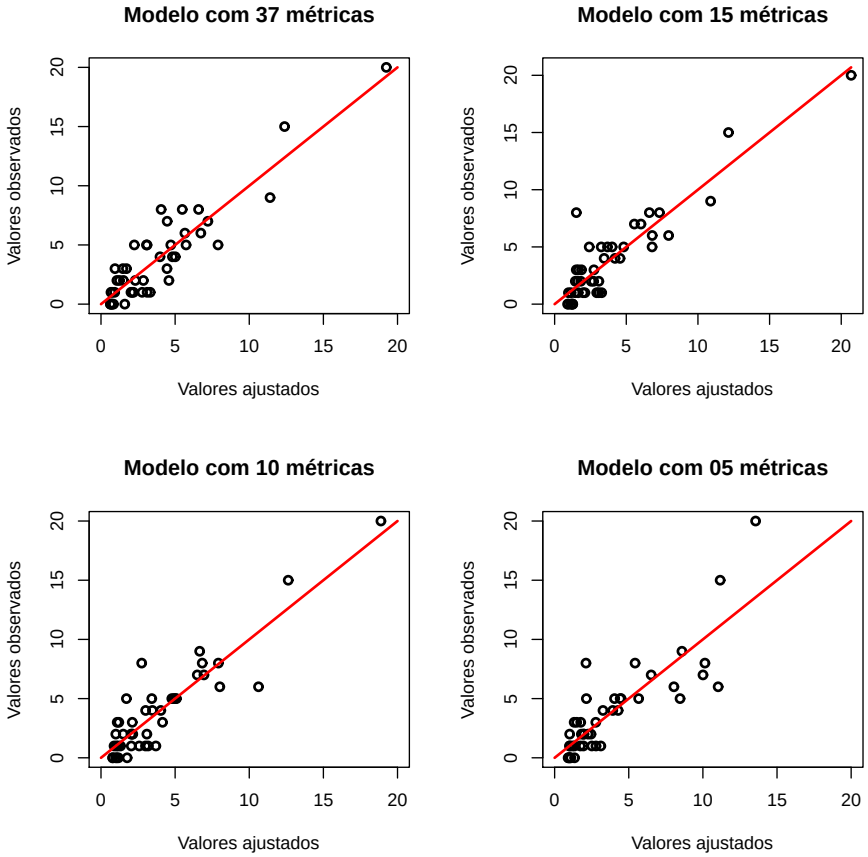

Figura 5.4. Comparação dos resultados dos modelos de Poisson com os valores observados nos módulos do MIPS.

Tabela 5.6. Desempenho dos modelos binomiais negativos no MIPS.

Modelo	Todos os dados		Validação	
	Cor. Spearman	χ^2	Cor. Spearman	χ^2
37 métricas	0,831	0,336	0,728	0,382
15 métricas	0,823	0,268	0,780	0,369
10 métricas	0,812	0,227	0,717	0,312
05 métricas	0,835	0,387	0,798	0,357

A Figura 5.5 apresenta um gráfico de dispersão dos resultados observados dos valores ajustados pelo modelo. De uma maneira geral, o desempenho dos modelos binomiais negativos é similar ao desempenho dos modelos de Poisson.

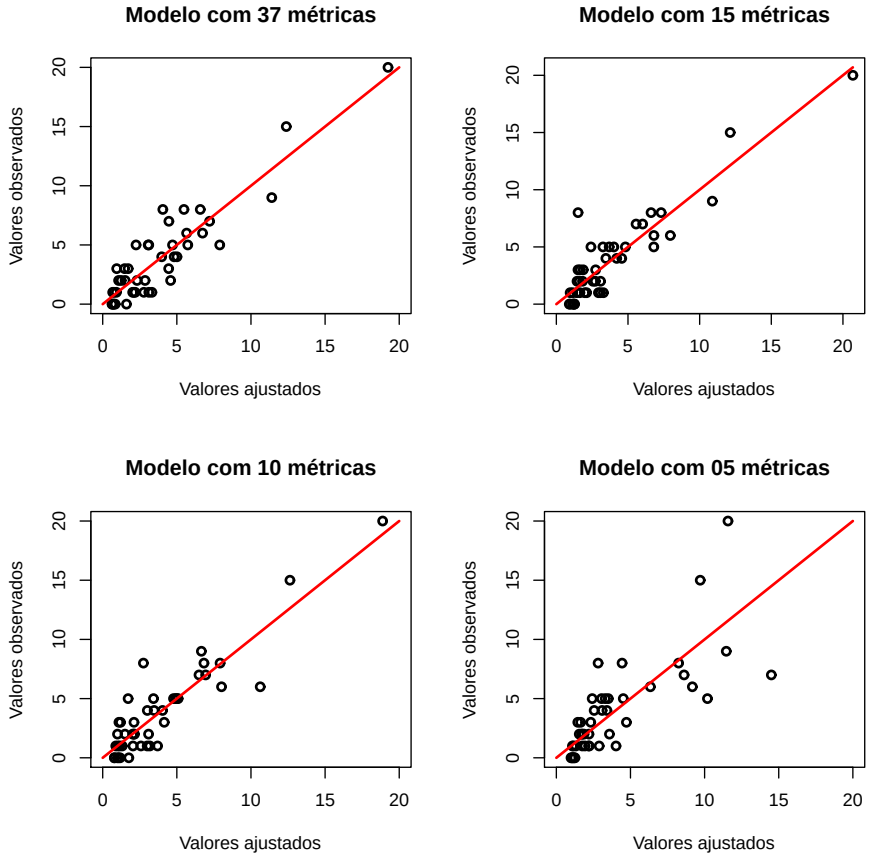

Figura 5.5. Comparação dos resultados dos modelos binomiais negativos com os valores observados nos módulos do MIPS.

5.1.3.4 Modelo Logístico

Ao contrário dos modelos apresentados até agora, o modelo de regressão logístico classifica os dados em 2 grupos. Como o objetivo deste trabalho é identificar as características dos módulos que possuem maior propensão a erros, os dois grupos foram definidos como: módulos que possuem número menor ou igual à média do número de erros (0) e módulos que possuem número maior de erros que a média de erros (1).

Para analisar os resultados do modelo, além dos coeficientes de correlação de Spearman, é utilizada a notação de classificação binária. Para tanto, devem ser construídas as matrizes de confusão e calculados os coeficientes s, f e J, conforme

descrito na Seção 3.4.

A Tabela 5.7 apresenta as correlações de Spearman do modelo com todos os dados e com os dados de validação. Os resultados dos dados de validação foram melhores, o que indica que o modelo tem uma boa capacidade de predição. Além disso, os modelos com menor número de métricas (10 e 5) apresentam melhor desempenho. A Tabela 5.7 também apresenta os valores do coeficiente de Youden (J). É interessante observar que os valores do coeficiente J estão muito próximos aos valores do coeficiente de correlação de Spearman.

Tabela 5.7. Desempenho dos modelos logísticos no MIPS.

Modelo	Todos os dados		Validação
	Cor. Spearman	J	Cor. Spearman
37 métricas	0,627	0,600	0,717
15 métricas	0,564	0,547	0,713
10 métricas	0,704	0,707	0,654
05 métricas	0,659	0,640	0,695

5.1.3.5 Comparação dos modelos

O objetivo deste trabalho é identificar quais módulos são mais propensos a erros utilizando métricas de produto e processo. Assim sendo, é feita uma comparação da eficiência com a qual os modelos selecionam um subconjunto de módulos responsável por 80% do total de erros. Os modelos linear, Poisson e binomial negativo apresentam uma estimativa da quantidade de erros de cada módulo, mas, nessa comparação, o único aspecto relevante é a lista de erros ordenada, gerada pelos modelos.

A Figura 5.6 ilustra, com um exemplo de 10 módulos, como a comparação é realizada. A Figura 5.6 (a) apresenta como o valor utilizado como referência da comparação é calculado. Com os dados reais de erros, é realizada uma ordenação decrescente e são selecionados os módulos com maior número de erros, até alcançar 80% do total de erros. No exemplo, o número total de erros é 52 e, 80% desse valor corresponde a 42 erros. Assim, os módulos selecionados são 8, 4, 5, 7 e 10, correspondendo a 50% do total. Esses módulos contém 42 (80,77%) erros.

Módulo	Erros Reais
8	20
4	7
5	6
7	5
10	4
1	3
3	3
9	2
2	1
6	1

(a)

Módulo	Erros do Modelo	Erros Reais
8	13,55	20
5	8,04	6
4	6,51	7
7	4,04	5
6	3,26	1
10	2,79	4
3	1,76	3
1	1,49	3
9	1,01	2
2	1,00	1

(b)

Figura 5.6. (a) Seleção dos módulos baseada nos erros reais (referência). (b) Seleção dos módulos baseada no modelo.

Na Figura 5.6 (b), os módulos são ordenados de acordo com os valores ajustados pelo modelo que está sendo comparado. Em seguida, os módulos são selecionados sequencialmente até que um valor maior ou igual 80% dos erros seja atingido. Neste exemplo foram selecionados os módulos 8, 5, 4, 7, 6 e 10 (60% do total). Nestes módulos estão localizados 43 (82,69%) erros.

A Tabela 5.8 e a Figura 5.7 apresentam o número de módulos selecionados pelos modelos linear, Poisson e binomial negativo, sendo que para cada um foram construídos modelos com 37, 15, 10 e 5 métricas. O número total de erros dos módulos do MIPS é 169, e, 80% desses erros correspondem ao valor de 136 erros. No caso real, utilizado como referência de comparação, foram necessários 19 módulos, ou seja, 43,18% do total.

Os módulos selecionados pelos modelos foram, no pior caso, um pouco menos que 60% do total, ou seja, não tão distante dos 43,18% do valor de referência. O modelo que apresentou melhor desempenho foi o linear com 5 métricas, selecionando apenas 22 módulos com pelo menos 80% dos erros. Em segundo lugar, os modelos Poisson com 37 métricas, e binomial negativo com 37 e 5 métricas, selecionaram 23 módulos. O mais interessante dessa comparação é constatar que

Tabela 5.8. Desempenho dos modelos para selecionar módulos que contenham 80% dos erros do MIPS.

Modelo	Métricas	Módulos	Percentual
Real	-	19	43,18%
Linear	37	25	56,82%
Linear	15	26	59,09%
Linear	10	25	56,82%
Linear	05	22	50,00%
Poisson	37	23	52,27%
Poisson	15	26	59,09%
Poisson	10	24	54,55%
Poisson	05	25	56,82%
Binomial Negativo	37	23	52,27%
Binomial Negativo	15	26	59,09%
Binomial Negativo	10	24	54,55%
Binomial Negativo	05	23	52,27%

apenas um conjunto de 5 métricas mais correlacionadas com os erros é suficiente para identificar, com boa precisão, os módulos mais propensos a erros. Levando-se em consideração a média dos resultados, o modelo binomial negativo se mostrou ligeiramente mais eficiente.

No caso dos modelos de regressão logísticos, não é possível efetuar a ordenação decrescente, pois os módulos são classificados em dois grupos: alto risco, com mais erros que a média e baixo risco, com menos erros que a média. Assim, utiliza-se a densidade de erros, d, definida como:

$$d = \frac{\sum \text{Erros módulos alto risco}}{\sum \text{Número módulos alto risco}} \tag{5.1}$$

Dessa forma, um valor alto da densidade de erros significa que foram classificados como alto risco apenas módulos que, de fato, possuem um número grande de erros. A Tabela 5.9 e a Figura 5.8 apresentam os valores de d para os modelos logísticos com 37, 15, 10 e 5 métricas. O valor real $(7,16)$ é utilizado como referência. Todos os modelos apresentam resultados bem próximos ao valor de referência. O modelo que apresenta o melhor resultado utiliza 10 métricas, sendo que o segundo melhor utiliza 5 métricas. Mais uma vez pode-se observar o bom

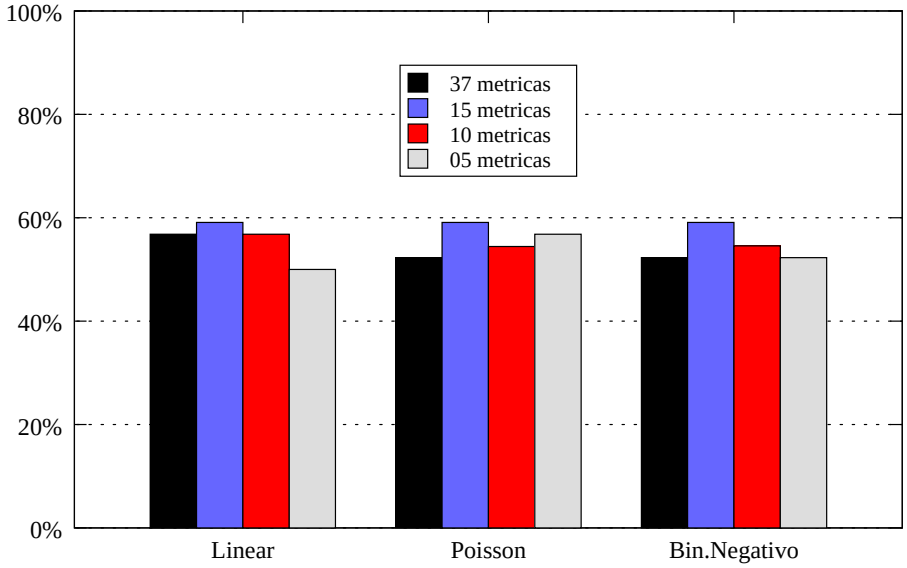

Figura 5.7. Gráfico de barras do desempenho dos modelos para selecionar módulos que contenham 80% dos erros do MIPS.

desempenho do modelo construído apenas com 5 métricas.

Tabela 5.9. Densidade de erros dos módulos selecionados pelos modelos logístico no MIPS.

Modelo	Métricas	d
Real	-	$7,16$
Logístico	37	$5,21$
Logístico	15	$5,18$
Logístico	10	$5,83$
Logístico	05	$5,39$

5.1.4 Influência do número de modificações e erros

Nesta Seção, são consideradas apenas duas métricas de processo: número de modificações (`commits`) e número de erros. Com base nessas métricas é calculada, dinamicamente, uma lista de módulos mais propensos a erros (Nacif et al. [2011a]). É importante ressaltar que a lista de módulos propensos a erro é recalculada sempre que uma métrica é atualizada, ou seja, a lista é atualizada várias vezes ao longo do processo de desenvolvimento do circuito integrado.

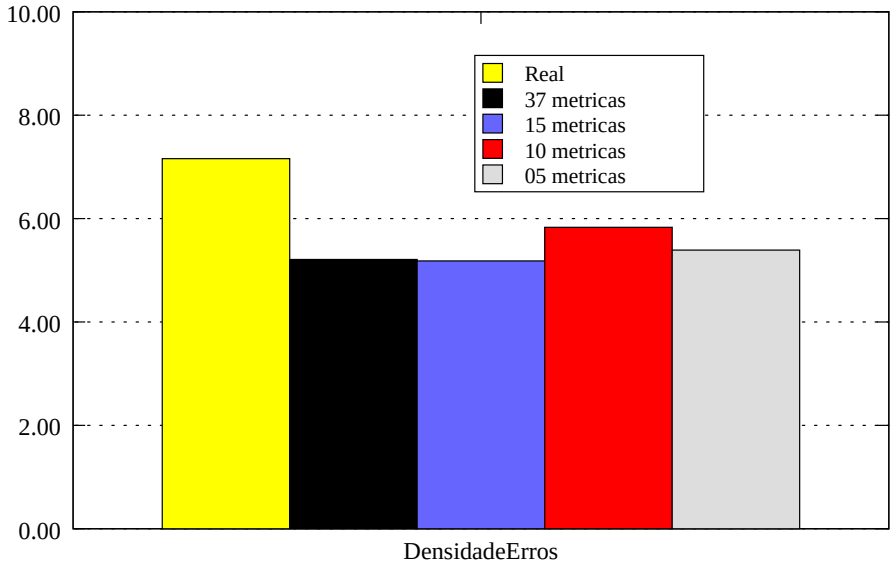

Figura 5.8. Gráfico de barras da densidade de erros dos módulos selecionados pelos modelos logístico no MIPS.

O **Proj. D** do MIPS desenvolvido pelos alunos é utilizado para gerar os dados apresentados nesse estudo de caso. Esse projeto possui 160 modificações (`commits`) e por volta de 50 erros reportados. O algoritmo utilizado na construção da lista de módulos propensos a erros é apresentado na Figura 5.9. Esse algoritmo é baseado na idéia de uma *cache* na qual são mantidos os módulos mais frequentemente modificados ou corrigidos. Na ocorrência de erros, é verificado se o módulo no qual o erro foi relatado está na lista de módulos propensos a erro. Em caso positivo, acontece um acerto (*hit*), caso contrário acontece uma falta (*miss*) e o módulo é incluído na lista. No algoritmo, a lista de módulos propensos a erros é representada pela variável *ListaModulos*.

O algoritmo de cálculo da lista de módulos propensos a erros pode utilizar duas políticas de substituição:

- Primeiro a entrar, primeiro a sair (FIFO - *First In, First Out*): Nessa política, é levado em consideração apenas a ordem na qual os módulos foram incluídos na lista;
- Mais frequentemente modificados ou corrigidos: Nessa caso, é levado em consideração a frequência com a qual os módulos presentes na lista foram

```
1: inicializacao ListaModulos
2: for i ← 1 to |Revisoes| do
3:    Modulo ← Revisoes[i]
4:    if Modulo nao esta na lista then
5:       if |ListaModulos| = TamanhoListaModulos then
6:          Substituir Modulo
7:       else
8:          Incluir Modulo
9:       end if
10:   end if
11: end for
```

Figura 5.9. Algoritmo para cálculo da lista de módulos propensos a erro.

modificados ou corrigidos.

Para o cálculo do desempenho do algoritmo, foram realizadas simulações com o tamanho da lista igual a 3 e 5, que corresponde a 23,08% (3/13) e 38,46% (5/13) dos 13 módulos que tiveram erros relatados, e 17,65% (3/17) e 29,41% (5/17) dos 17 módulos modificados durante o desenvolvimento do MIPS. Para ser utilizado como parâmetro de comparação, foi definida a probabilidade de acerto aleatória, que é calculada de acordo com o tamanho da lista e o número total de módulos.

As Figuras 5.10 e 5.11 apresentam o desempenho do algoritmo de cálculo da lista de módulos propensos a erro, considerando o tamanho da lista igual a 3. Na Figura 5.10 tem-se o desempenho do algoritmo com as políticas de substituição FIFO e Mais Frequentemente Modificado (MFM). Inicialmente, o desempenho do algoritmo aleatório é superior, pois o número de módulos conhecidos é menor que o tamanho da lista. A partir da décima revisão, o algoritmo atinge uma taxa de acerto que se mantém por volta de 80%. O desempenho do algoritmo que utiliza política de substituição FIFO é ligeiramente superior ao MFM. O desempenho do algoritmo aleatório cai progressivamente e por volta de revisão 130 atinge 20%. Na Figura 5.11, o desempenho do algoritmo com as políticas de substituição FIFO e Mais Frequentemente Corrigidos (MFC) é apresentado. Nesse caso, o algoritmo atinge uma taxa de acerto próxima a 55%.

As Figuras 5.12 e 5.13 apresentam o desempenho dos algoritmos com o tamanho da lista igual a 5. Como esperado, o desempenho do algoritmo com o tamanho

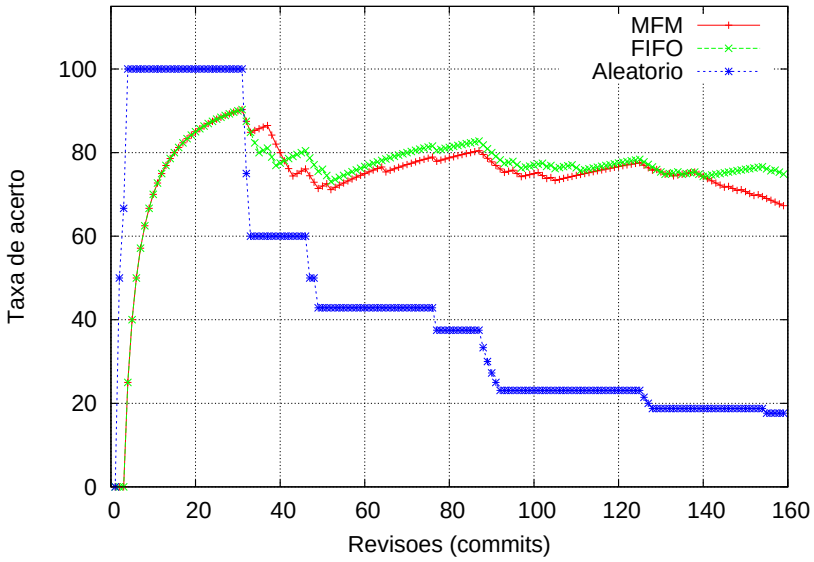

Figura 5.10. Desempenho do algoritmo considerando a lista de módulos propensos a erro com tamanho 3 e política de substituição mais frequentemente modificado.

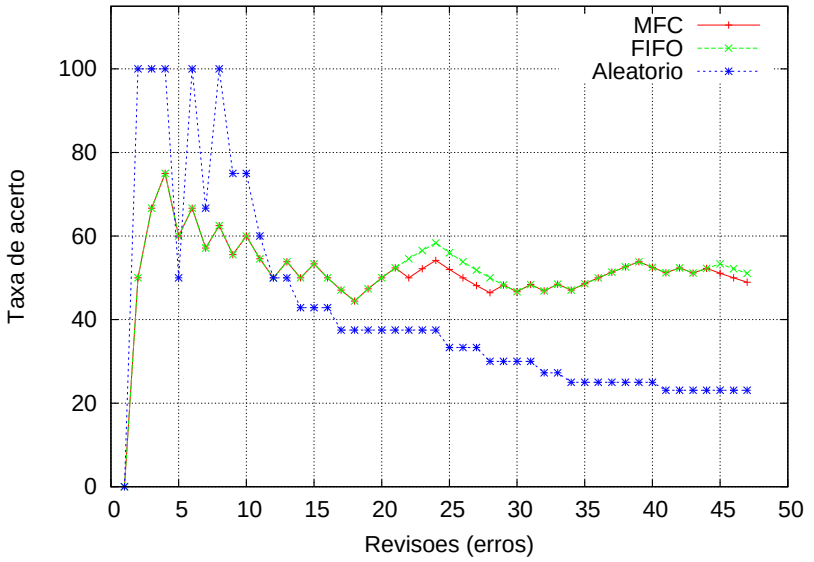

Figura 5.11. Desempenho do algoritmo considerando a lista de módulos propensos a erro com tamanho 3 e política de substituição mais frequentemente corrigido.

da lista maior foi melhorado. O desempenho dos algoritmos baseados em módulos mais frequentemente modificados e corrigidos alcançam uma taxa de acerto de,

respectivamente, 85% e 60%.

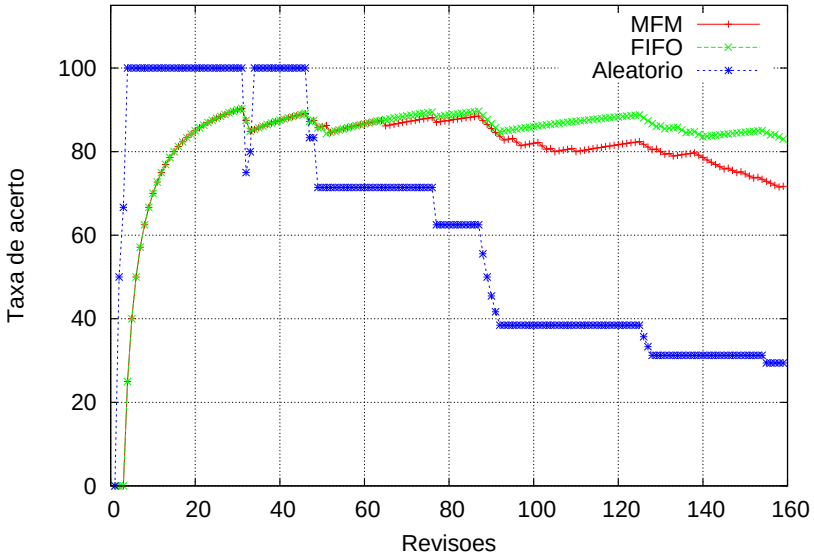

Figura 5.12. Desempenho do algoritmo considerando a lista de módulos propensos a erro com tamanho 5 e política de substituição mais frequentemente modificado.

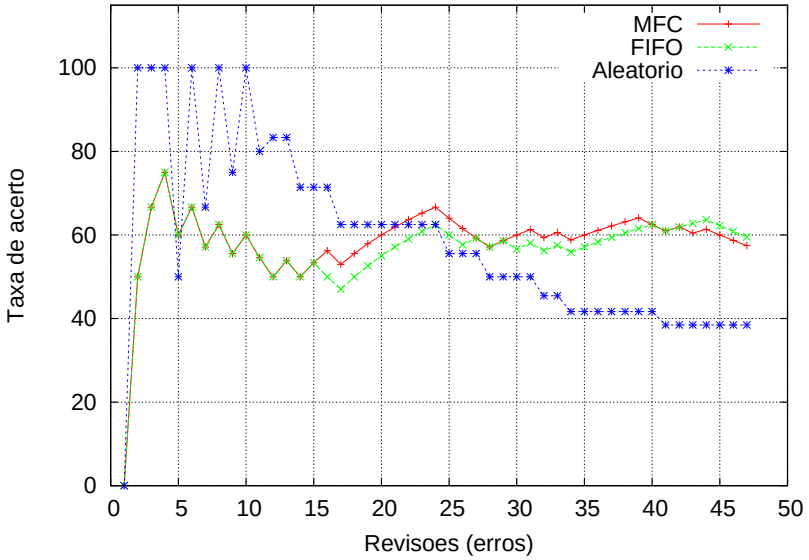

Figura 5.13. Desempenho do algoritmo considerando a lista de módulos propensos a erro com tamanho 5 e política de substituição mais frequentemente corrigido.

O algoritmo de cálculo da lista de módulos propensos a erro, apesar de simples, demonstra a relação entre o número de alterações sofridas pelos módulos e a ocorrência de erros. Além disso, os resultados do algoritmo com módulos mais recentemente corrigidos sugerem que, módulos com histórico problemático, tendem a continuar apresentando erros ao longo do desenvolvimento do circuito integrado.

5.2 Estudo de Caso 2: OpenSPARC

O segundo estudo de caso utiliza os dados do processador de código aberto OpenSPARC, descrito na Seção 4.2.2. Alguns módulos do processador possuem erros comentados no código-fonte, conforme ilustra a Figura 5.14. Nesse exemplo, é apresentado um trecho de código do módulo `tlu_tcl`. A linha 1.105 indica que esse erro é o de número 3.919, sendo que a linha 1.106 se refere à versão comentada e a linha 1.107 à versão corrigida. Todos os módulos do OpenSPARC são inspecionados automaticamente, sendo selecionados módulos que totalizam 264 erros.

```
linha 1.089: assign intrpt_taken =
linha 1.090:            rstint_taken | hwint_taken | sirint_taken;
...
linha 1.105: // modified for bug 3.919
linha 1.106: // assign trap_to_redmode = trp_lvl_at_maxtlless1 & ~intrpt_taken;
linha 1.107: assign trap_to_redmode = trp_lvl_at_maxtlless1 & ~(rstint_taken | sirint_taken);
```

Figura 5.14. Exemplo de erro comentado no módulo `tlu_tcl` do OpenSPARC.

5.2.1 Extração das Métricas

Infelizmente, nem todos os módulos do OpenSPARC possuem erros comentados. Na verdade, 99% dos erros comentados estão localizados em 2 unidades: a unidade de lógica de *trap* (TLU - *Trap Logic Unit*) e a unidade de carregamento/escrita (LSU - *Load/Store Unit*). A TLU é responsável pela lógica de gerenciamento das *traps* e interrupções de software. É capaz de suportar seis níveis de *traps* em diferentes modos de permissão e 64 interrupções de software pendentes. A LSU é responsável pelo processamento de todas as instruções que acessam a memória de

dados. Essa unidade realiza a interconexão entre as unidades funcionais do núcleo do OpenSPARC e outros subsistemas de memória. Além disso, fazem parte da LSU, a TLB de dados e a memória cache L1. Maiores detalhes sobre a micro-arquitetura do OpenSPARC podem ser encontradas em Sun Microsystems, Inc. [2006].

A Tabela 5.10 apresenta os 17 módulos da TLU utilizados neste estudo de caso.

Tabela 5.10. Módulos da TLU.

(1) `sparc_tlu_dec64`	(7) `sparc_tlu_intctl`	(13) `sparc_tlu_intdp`
(2) `sparc_tlu_penc64`	(8) `sparc_tlu_zcmp64`	(14) `tlu`
(3) `tlu_addern_32`	(9) `tlu_hyperv`	(15) `tlu_incr64`
(4) `tlu_misctl`	(10) `tlu_mmu_ctl`	(16) `tlu_mmu_dp`
(5) `tlu_pib`	(11) `tlu_prencoder16`	(17) `tlu_rrobin_picker`
(6) `tlu_tcl`	(12) `tlu_tdp`	

A Tabela 5.11 apresenta os 21 módulos da LSU utilizados nesse trabalho. Foram incluídos 2 módulos da unidade de processamento de fluxo (SPU - *Stream Processing Unit*), pois a funcionalidade desses módulos está ligada à LSU. Assim sendo, fazem parte deste estudo de caso 38 módulos do OpenSPARC.

Tabela 5.11. Módulos da LSU.

(1) `lsu_asi_decode`	(8) `lsu_dcache_lfsr`	(15) `lsu_dcdp`
(2) `lsu_dctl`	(9) `lsu_dctldp`	(16) `lsu_dc_parity_gen`
(3) `lsu_excpctl`	(10) `lsu_pcx_qmon`	(17) `lsu_qctl1`
(4) `lsu_qctl2`	(11) `lsu_qdp1`	(18) `lsu_qdp2`
(5) `lsu_rrobin_picker2`	(12) `lsu_stb_ctl`	(19) `lsu_stb_ctldp`
(6) `lsu_stb_rwctl`	(13) `lsu_stb_rwdp`	(20) `lsu_tagdp`
(7) `lsu_tlbdp`	(14) `spu_lsurpt`*	(21) `spu_lsurpt1`*

$*$ Módulos externos à LSU (SPU).

A Tabela 5.12 apresenta os valores mínimos, máximos, médios e desvios-padrão das métricas extraídas dos 38 módulos. O número total de métricas estudadas é 46, sendo uma gerada pela ferramenta HCT, 8 pelo JasperGold™ e 37 pelo VParser. Ao contrário do estudo de caso do MIPS, no OpenSPARC não se tem a métrica de número de modificações (`commits`).

As métricas JG_FLOPS, JG_LATCHES, VP_while, VP_function, VP_do, VP_bit, VP_final, VP_forever, VP_casez, VP_casex, VP_inout, VP_initial, VP_posedge, VP_negedge, VP_case, VP_default e VP_endcase não possuem ocorrências e, portanto, não são apresentadas na Tabela 5.12.

A ausência de ocorrências de algumas métricas relacionadas a circuitos sequenciais se deve ao fato de que as unidades TLU e LSU são completamente combinacionais não tendo, portanto, sincronização por borda (VP_posedge e VP_negedge) e memória (JG_FLOPS e JG_LATCHES).

As métricas VP_module e VP_endmodule aparecem apenas uma vez, pois cada arquivo Verilog possui um único módulo. A métrica HCT_MCCABE, assim como no estudo de caso do MIPS, apresenta pouca variação. As principais métricas de complexidade extraídas pelo JasperGold^{TM} têm valores mínimos e máximo bem distantes, o que significa que a complexidade dos módulos é diversificada. O número total de linhas de código dos módulos (JG_RTL_LINES) se aproxima de 54.000. A métrica VP_assign possui valores elevados, pois essa construção, em Verilog, é responsável pela implementação de circuitos combinacionais.

5.2.2 Análise de Correlação das Métricas com os Erros

Para o cálculo das correlações entre os erros e as métricas é utilizado o coeficiente de Spearman, pois os dados não seguem a distribuição normal. A Tabela 5.13 apresenta os coeficientes de correlação de Spearman das métricas com os erros observados nos módulos ordenados da maior correlação para a menor. As métricas dentro do intervalo compreendido entre 1 e 15 apresentam correlações com significância maior que 99%.

A métrica que mais se correlaciona com os erros é a VP_wire, fortemente ligada a JG_NETS, a segunda maior correlação. Dessa forma, o número de sinais existentes no circuito tem relação com o número de erros. Em seguida, tem-se o número de linhas de comentários (VP_Comments), o que poderia ser explicado pelo maior cuidado dos desenvolvedores em comentar módulos notadamente problemáticos. O número de linhas de código (JG_RTL_LINES), também apresenta uma boa correlação com o número de erros. O número de símbolos (VP_Symbols), que denota o nome dado aos sinais, é a quinta maior correlação. Também aparecem na

Tabela 5.12. Estatística descritiva das métricas extraídas do conjunto de 38 módulos selecionados do OpenSPARC.

Métrica	Min	Max	Média	Desvio-padrão
HCT_MCCABE	1	3	1,13	0,47
JG_COMMENT_LINES	23	165	45,24	30,98
JG_GATES	0	4.334	861,79	1.083,41
JG_NETS	10	6.496	850,03	1.355,44
JG_PORTS	2	448	68,37	87,3
JG_RTL_LINES	9	7.832	1.418,79	1.908,59
JG_RTL_INSTANCES	0	302	46,45	69,76
VP_Keywords	5	13	6,82	2,19
VP_Operators	7	23	14,68	4,06
VP_Symbols	3	1782	250,87	355,92
VP_Comments	21	1684	315,79	392,92
VP_Attributes	1	121	27,87	29,28
VP_Numbers	1	121	27,87	29,28
VP_Strings	0	3	1,29	0,98
VP_endmodule	1	1	1	0
VP_input	1	177	38,16	43,17
VP_module	1	1	1	0
VP_output	1	262	28,92	46,48
VP_reg	0	15	0,53	2,44
VP_always	0	3	0,21	0,58
VP_begin	0	3	0,32	0,77
VP_end	0	3	0,32	0,77
VP_if	0	2	0,13	0,41
VP_wire	0	588	78,05	121,65
VP_else	0	2	0,11	0,39
VP_or	0	8	0,26	1,33
VP_assign	0	1.012	124,42	209,13
VP_for	0	2	0,11	0,39
VP_integer	0	2	0,11	0,39

lista métricas relacionadas ao número de pinos de E/S (JG_PORTS, VP_Output e VP_Input).

Tabela 5.13. Correlações entre as métricas e o número de erros nos módulos do OpenSPARC.

Número	Métrica	Correlação Spearman
1	VP_wire	0,880
2	JG_NETS	0,875
3	VP_Comments	0,875
4	JG_RTL_LINES	0,859
5	VP_Symbols	0,855
6	VP_assign	0,845
7	JG_PORTS	0,835
8	VP_output	0,830
9	VP_Operators	0,790
10	JG_RTL_INSTANCES	0,775
11	VP_input	0,771
12	VP_Attributes	0,666
13	VP_Numbers	0,666
14	VP_Strings	0,642
15	JG_GATES	0,628
16	VP_if	-0,344
17	HCT_MCCABE	-0,294
18	VP_else	-0,294
19	VP_for	-0,294
20	VP_integer	-0,294
21	VP_reg	-0,247
22	VP_always	-0,247
23	VP_begin	-0,246
24	VP_end	-0,246
25	VP_Keywords	-0,136
26	JG_COMMENT_LINES	0,077
27	VP_or	0,041

5.2.3 Construção e Validação do Modelo

A Figura 5.15 apresenta o histograma do número de erros do OpenSPARC. Pode-se observar que os dados da variável dependente não seguem a distribuição normal, sendo que a maioria dos módulos possui poucos ou nenhum erro. Apenas 6 módulos possuem mais que 10 erros. Mais da metade dos módulos (20) não possuem erros.

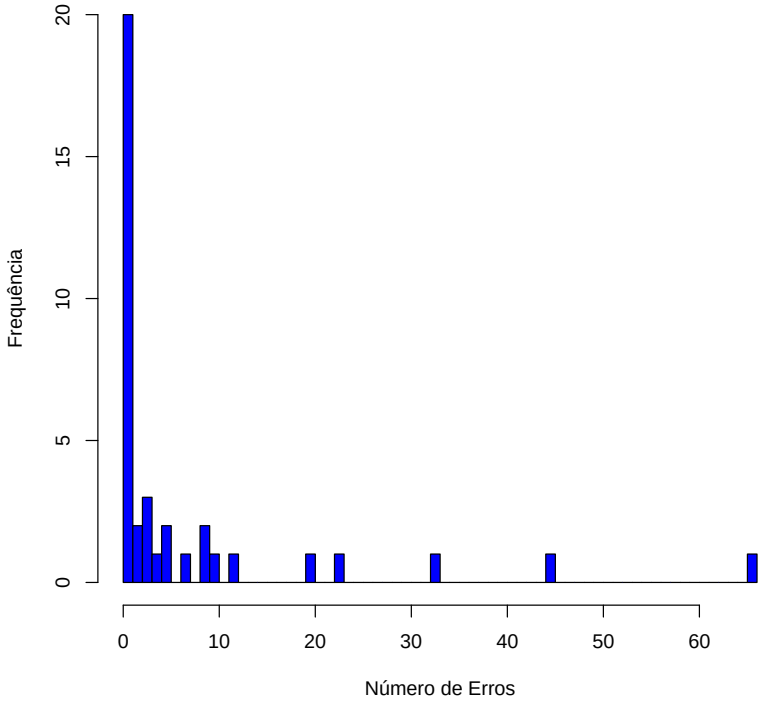

Figura 5.15. Histograma dos erros dos módulos selecionados do OpenSPARC.

Este estudo de caso apresenta os resultados dos modelos de regressão Linear, Poisson, Negativo Binomial e Logístico. Para cada técnica de regressão são desenvolvidos 4 modelos com, respectivamente, 27, 15, 10 e 5 métricas. Para cada modelo, são calculados os componentes principais e mantidos os componentes que têm significância maior que 99%. Para a validação dos modelos é utilizada a técnica de divisão de dados, na qual os dados são agrupados aleatoriamente em dois conjuntos: treinamento ($\frac{2}{3}$) e teste ($\frac{1}{3}$). A divisão aleatória dos dados é repetida 100 vezes e são apresentados os valores médios.

5.2.3.1 Modelo Linear

O desempenho do modelo de regressão linear é apresentado na Tabela 5.14. Pode-se observar que o valor do coeficiente de correlação de Spearman apresenta

valores muito bons, próximos a $0,90$, para todos os modelos. O coeficiente R^2 também apresenta bons resultados, tendo valores próximos a $0,80$ para os modelos com 5, 10 e 15 métricas. Ao se utilizar os dados de validação, o desempenho do modelo permanece praticamente inalterado. O coeficiente de correlação de Spearman sofre uma pequena queda, enquanto os valores de R^2 não se alteram.

Tabela 5.14. Desempenho dos modelos lineares no OpenSPARC.

Modelo	Todos os dados		Validação	
	Cor. Spearman	R^2	Cor. Spearman	R^2
27 métricas	0,893	0,641	0,867	0,641
15 métricas	0,884	0,754	0,849	0,754
10 métricas	0,872	0,791	0,807	0,791
05 métricas	0,871	0,726	0,772	0,726

A Figura 5.16 apresenta o gráfico de dispersão dos resultados observados e dos valores ajustados pelo modelo. Pode-se observar que o modelo linear foi capaz de ajustar os erros de maneira bem semelhante aos erros observados. Os valores de R^2 próximos a $0,80$ sugerem que os dados deste estudo de caso podem ser satisfatoriamente modelados em uma distribuição linear.

5.2.3.2 Modelo de Poisson

A Tabela 5.15 apresenta os resultados do modelo de regressão de Poisson. São calculados os coeficientes χ^2 e os coeficientes de correlação de Spearman para os modelos com 27, 15, 10 e 5 métricas. Considerando a utilização de todos os dados, os coeficientes de correlação χ^2 dos modelos apresentam valores muito ruins, o que significa que a distribuição de Poisson não é a mais adequada para modelar os dados. Apesar disso, os coeficientes de correlação de Spearman apresentam bons resultados para todos os modelos ($> 0,80$). Em relação aos resultados de validação, a alteração no desempenho foi muito pequena. Mais uma vez o modelo com 5 métricas tem um desempenho superior.

A Figura 5.17 apresenta gráficos de dispersão comparando os dados gerados pelo modelo com os dados observados. Os gráficos apresentam bons resultados, em coerência com os dados do coeficiente de correlação de Spearman.

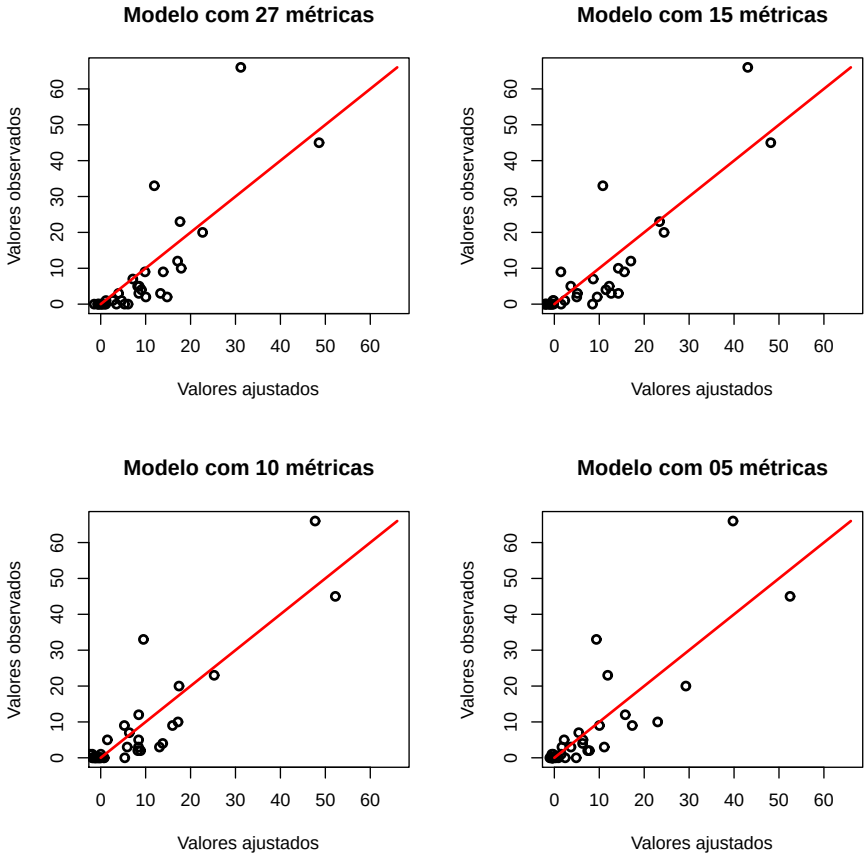

Figura 5.16. Comparação dos resultados dos modelos lineares com os valores observados nos módulos do OpenSPARC.

Tabela 5.15. Desempenho dos modelos de Poisson no OpenSPARC.

Modelo	Todos os dados		Validação	
	Cor. Spearman	χ^2	Cor. Spearman	χ^2
27 métricas	0,850	0	0,771	0,013
15 métricas	0,830	0	0,799	0,009
10 métricas	0,820	0	0,806	0,020
05 métricas	0,830	0	0,799	0,002

5.2.3.3 Modelo Binomial Negativo

A Tabela 5.16 apresenta o desempenho dos modelos binomiais negativos. De uma maneira geral, o desempenho dos modelos binomiais negativos é similar

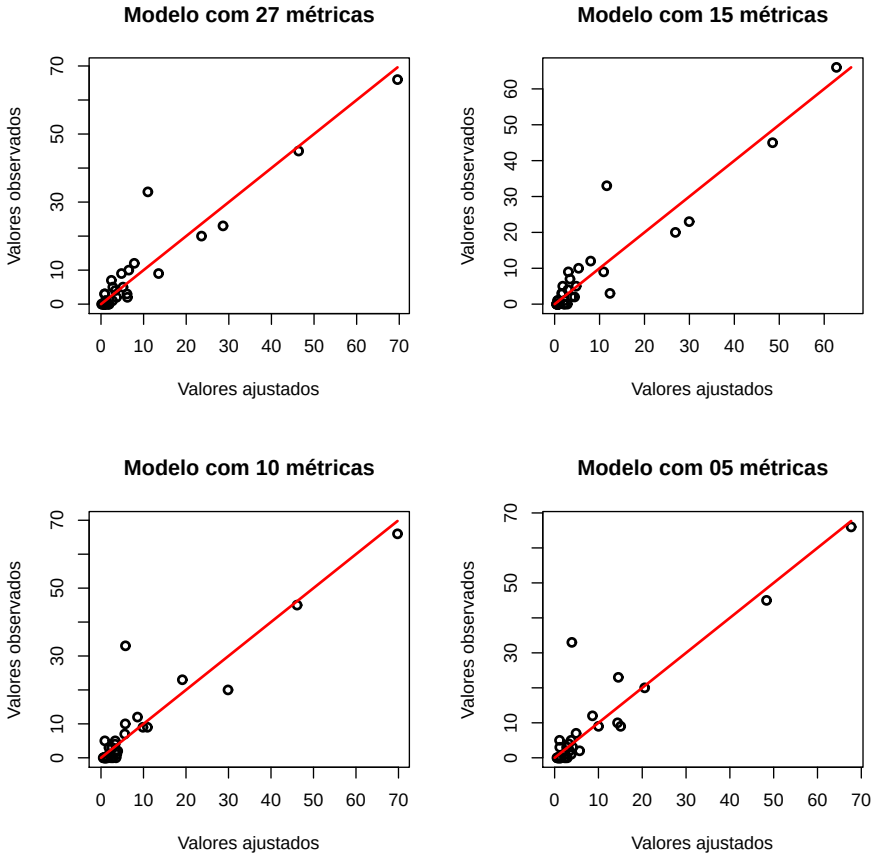

Figura 5.17. Comparação dos resultados dos modelos de Poisson com os valores observados nos módulos do OpenSPARC.

ao desempenho dos modelos de Poisson, mas o valor do coeficiente χ^2 apresenta resultados bem melhores. Dessa forma, o modelo binomial negativo é melhor para representar os dados desse estudo de caso que o modelo de Poisson.

Tabela 5.16. Desempenho dos modelos binomiais negativos no OpenSPARC.

Modelo	Todos os dados		Validação	
	Cor. Spearman	χ^2	Cor. Spearman	χ^2
27 métricas	0,887	0,568	0,866	0,482
15 métricas	0,854	0,404	0,824	0,365
10 métricas	0,887	0,433	0,865	0,381
05 métricas	0,882	0,411	0,859	0,383

A Figura 5.18 apresenta um gráfico de dispersão dos resultados observados e dos valores ajustados pelo modelo. É interessante observar que o maior valor observado (66) foi ajustado para valores muito maiores: 2.400 nos modelos com 27 e 5 métricas e 4.000 no modelo com 10 métricas. Para melhor visualização, esses pontos não foram plotados no gráfico.

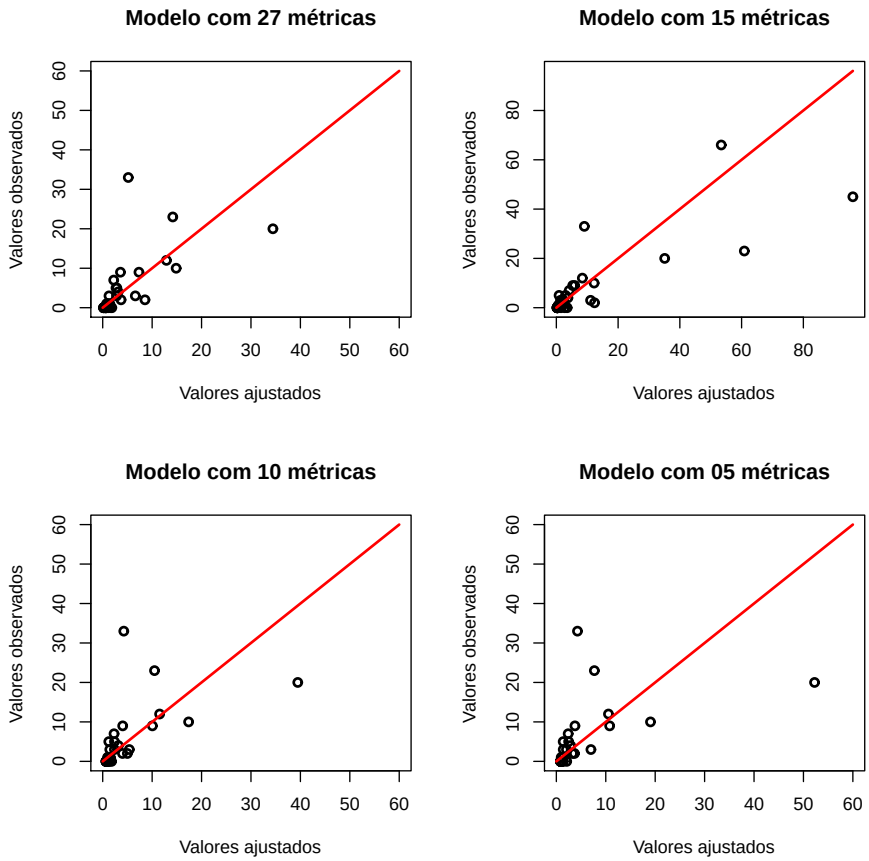

Figura 5.18. Comparação dos resultados dos modelos binomiais negativos com os valores observados nos módulos do OpenSPARC.

5.2.3.4 Modelo Logístico

O modelo de regressão logístico classifica os dados em 2 grupos: módulos que possuem número menor ou igual à média do número de erros (0) e módulos que possuem número maior de erros que a média de erros (1). Para analisar

os resultados do modelo, além dos coeficientes de correlação de Spearman, são utilizados os coeficientes de Youden (J), conforme descrito na Seção 3.4.

Tabela 5.17 apresenta as correlações de Spearman do modelo com todos os dados e com os dados de validação. Os resultados dos dados de validação sofreram pouca alteração, o que indica que o modelo tem uma boa capacidade de predição. Além disso, o modelo com 5 métricas apresenta o melhor desempenho. Assim como o estudo de caso do MIPS, os valores dos coeficientes J acompanham os valores do coeficiente de correlação de Spearman.

Tabela 5.17. Desempenho dos modelos logísticos no OpenSPARC.

Modelo	Todos os dados		Validação
	Cor. Spearman	J	Cor. Spearman
27 métricas	0,664	0,750	-0,060
15 métricas	0,664	0,750	0,660
10 métricas	0,701	0,786	0,679
05 métricas	0,740	0,821	0,693

5.2.3.5 Comparação dos modelos

Nessa Seção, é feita uma comparação da eficiência com a qual os modelos selecionam um subconjunto de módulos responsável por 80% do total de erros. Na comparação, o único aspecto relevante é a ordem com a qual os modelos geram as listas de módulos. Maiores detalhes podem ser encontrados na Seção 5.1.3.5.

A Tabela 5.18 e a Figura 5.19 apresentam o número de módulos selecionados pelos modelos linear, Poisson e binomial negativo, sendo que para cada um foram construídos modelos com 27, 15, 10 e 5 métricas. O número total de erros dos módulos do OpenSPARC é 264, e, 80% destes erros correspondem ao valor de 212 erros. No caso real, utilizado como referência de comparação, foram necessários 8 módulos, ou seja, 21,05% do total.

Os módulos selecionados pelos modelos foram, no pior caso, 31,58% do total, ou seja, muito próximo de 21,05% do valor de referência. O modelo que apresentou melhor desempenho foi o Poisson com 27 e 10 métricas. Apesar disso, o modelo de Poisson com 5 métricas apresentou o pior desempenho, juntamente com o linear com 15 métricas. Os modelos binomiais negativos apresentaram resultados

similares com os 4 conjuntos de métricas. Com exceção do modelo de Poisson, um conjunto de 5 métricas mais correlacionadas com os erros é suficiente para identificar, com boa precisão, os módulos mais propensos a erros. Levando-se em consideração a média dos resultados, os modelos de Poisson e binomial negativo se mostraram mais eficientes.

Tabela 5.18. Desempenho dos modelos para selecionar módulos que contenham 80% dos erros do OpenSPARC.

Modelo	Métricas	Módulos	Percentual
Real	-	8	21,05%
Linear	27	10	26,32%
Linear	15	12	31,58%
Linear	10	9	23,68%
Linear	05	10	26,32%
Poisson	27	8	21,05%
Poisson	15	9	23,68%
Poisson	10	8	21,05%
Poisson	05	12	31,58%
Binomial Negativo	27	10	26,32%
Binomial Negativo	15	9	23,68%
Binomial Negativo	10	10	26,32%
Binomial Negativo	05	9	23,68%

A Tabela 5.19 e a Figura 5.20 apresentam os valores da densidade de erros, definido na Seção 5.1.3.5, para os modelos logísticos com 27, 15, 10 e 5 métricas. O valor real $(23,40)$ é utilizado como referência. Todos os modelos apresentam resultados similares, cerca de 70% do valor de referência. O modelo que apresenta o melhor resultado utiliza 5 métricas.

Tabela 5.19. Densidade de erros dos módulos selecionados pelos modelos logístico no OpenSPARC.

Modelo	Métricas	d
Real	-	$23,40$
Logístico	27	$15,18$
Logístico	15	$15,18$
Logístico	10	$15,81$
Logístico	05	$16,67$

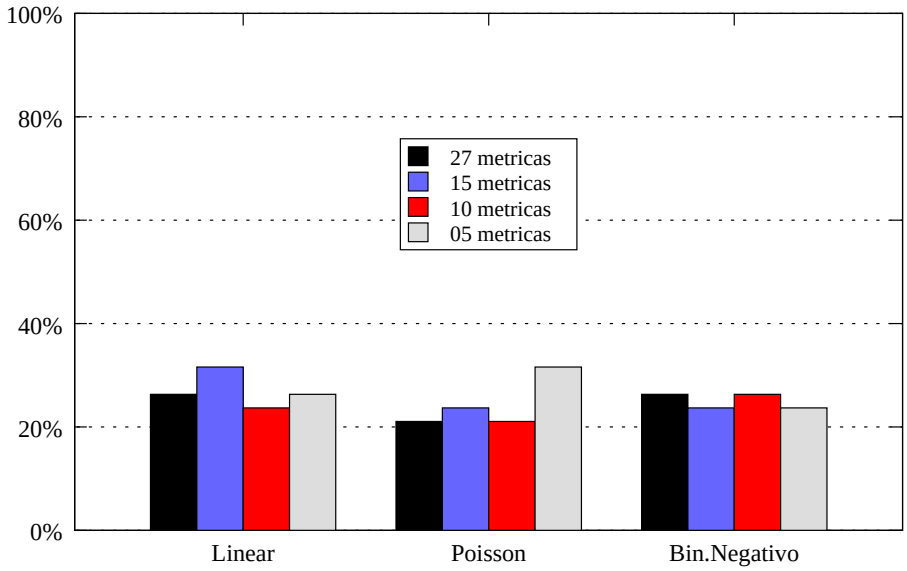

Figura 5.19. Gráfico de barras do desempenho dos modelos para selecionar módulos que contenham 80% dos erros do OpenSPARC.

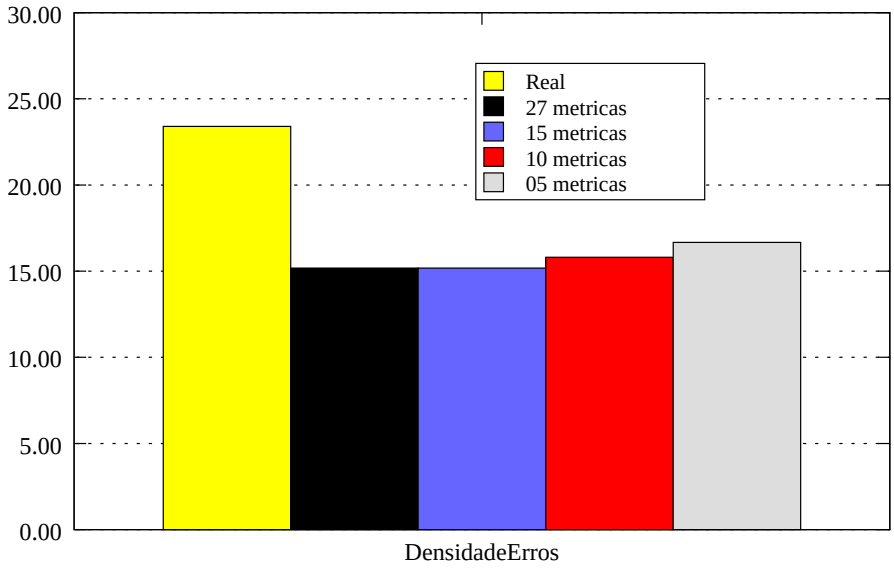

Figura 5.20. Gráfico de barras da densidade de erros dos módulos selecionados pelos modelos logístico no OpenSPARC.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Conclusões

A utilização de métricas de produto e de processo para identificar módulos de software propensos a erros é uma área de pesquisa estabelecida e, ainda sim, promissora. A cada dia, o processo e os desafios de desenvolvimento de software e de circuitos integrados se assemelham mais. Assim sendo, este trabalho demonstrou a importância e a utilidade das informações armazenadas em sistemas de controle de versão de projetos de circuitos integrados. Da mesma forma como é feito em sistemas de software, diversos estudos podem ser realizados com as métricas extraídas desses sistemas. Foi proposto o termo "evolução de hardware" (*hardware evolution*) para se referir à pesquisa que utiliza informações de sistemas de controle de versão de projetos de circuitos integrados.

A ferramenta EyesOn foi desenvolvida para automatizar o processo de extração de métricas de produto e processo. Essa ferramenta de código aberto permite que sejam incluídas novas métricas e outros sistemas de controle de versão e rastreamento de erros. Assim, a ferramenta EyesOn pode ser utilizada para facilitar o estudo de diferentes aspectos da evolução de hardware.

Os resultados apresentados neste trabalho demonstram que a metodologia proposta é capaz de construir modelos de predição de erros utilizando métricas extraídas de sistemas de controle de versão de projetos de circuitos integrados. Estes modelos apresentaram bom desempenho até mesmo com um número redu-

zido de métricas (5). Nos estudos de caso dos processadores MIPS e OpenSPARC, o modelo de predição mais simples (Linear) apresentou bom desempenho sendo, portanto, desnecessária a utilização de modelos de predição mais sofisticados como Poisson e Binomial Negativo. A ordenação do número de erros dos módulos construída a partir dos modelos se mostrou muito próxima da ordenação observada. A regressão logística também apresentou bons resultados ao classificar os módulos como propensos ou não propensos a conter erros. Devido às peculiaridades existentes em cada projeto de circuito integrado, foram construídos modelos distintos para os processadores MIPS e OpenSPARC.

O número de alterações, disponível apenas no estudo de caso do processador MIPS, se mostrou extremamente correlacionado com o número de erros. Sendo assim, foi proposto um algoritmo que mantém uma lista de módulos mais frequentemente modificados. Quando um erro é identificado, a lista é verificada e caso o módulo esteja na lista é contabilizado um acerto. O algoritmo proposto foi capaz de atingir, ao longo do processo de desenvolvimento do projeto MIPS, uma taxa de acerto próxima a 80%. Assim, o número de alterações sofridas pelos módulos pode ser utilizada para determinar, dinamicamente, módulos mais propensos a conter erros.

6.2 Trabalhos Futuros

Este trabalho demonstrou que a utilização de métricas de produto e métricas de processo para identificar módulos propensos a erro é viável. Contudo, o universo de métricas, dados, ferramentas e abordagens de análise pode ser ampliado. Nesse sentido, as principais propostas de continuação do trabalho são:

- **Mais métricas**: As métricas utilizadas neste trabalho foram extraídas por ferramentas de código livre e comerciais. Um caminho promissor para a continuação do trabalho é o estudo da correlação de novas métricas com a ocorrência de erros. Em especial, propõe-se a investigação de métricas de testabilidade de circuitos integrados e legadas de sistemas de software. Exemplos de métricas de testabilidade são os índices SCOAP (Goldstein & Thigpen [1988]), COP (Brglez [1984]) e PREDICT (Seth et al. [1985]). Seria inte-

ressante adaptar para linguagens de descrição de hardware, as métricas já utilizadas em sistemas de software como, por exemplo, as métricas propostas por Chidamber & Kemerer [1994], Halstead [1977] e Henry & Kafura [1981].

- **Mais dados**: A utilização de projetos desenvolvidos por alunos se mostrou útil para gerar dados para este trabalho. Esta prática pode ser estendida para o desenvolvimento de outros projetos de circuitos integrados. Pode-se utilizar também novos projetos de código aberto, desde que existam informações sobre a ocorrência de erros nos módulos. Finalmente, seria muito interessante empregar a metodologia proposta em um repositório de circuitos integrados industriais.

- **Mais ferramentas**: Quanto mais automatizado for o processo de extração das métricas e construção e validação do modelo, mais rápido pode se observar a eficácia de determinada métrica. Assim, se faz necessário o desenvolvimento de novas ferramentas para agilizar esse processo. Além disso, pretende-se investir em ferramentas de visualização das métricas.

- **Mais abordagens de análise**: Este trabalho utilizou técnicas estatísticas relativamente simples para construir os modelos de predição. Seria interessante avaliar o desempenho de outras técnicas como mineração de dados e aprendizado de máquina. Além disso, propõe-se o estudo do desempenho de modelos construídos em diferentes versões do projeto e também a análise de eficácia dos modelos em diferentes projetos.

Referências Bibliográficas

Albrecht, A. J. (1979). Measuring Application Development Productivity. Em Press, I. B. M., editor, *IBM Application Development Symp.*, pp. 83--92.

Albrecht, A. J. & Gaffney, J. E. (1983). Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Trans. Softw. Eng.*, 9:639--648.

Amor, J. J.; Robles, G. & Gonzalez-Barahona, J. M. (2006). Effort estimation by characterizing developer activity. Em *Proceedings of the 2006 international workshop on Economics driven software engineering research*, EDSER '06, pp. 3--6, New York, NY, USA. ACM.

Avizienis, A.; Laprie, J.-C.; Randell, B. & Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33.

Basili, V. R.; Briand, L. C. & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751--761.

Bayazit, A. A. & Malik, S. (2005). Complementary use of runtime validation and model checking. Em *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pp. 1052--1059, Washington, DC, USA. IEEE Computer Society.

Beizer, B. (1995). The pentium bug - an industry watershed. Testing Techniques Newsletter, TTN.

Bentley, B.; Baty, K.; Normoyle, K.; Ishii, M. & Yogev, E. (2004). Verification: What works and what doesn't. Em *Proceedings of Design Automation Conference*.

Bergeron, J. (2003). *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Springer.

Bernstein, A.; Ekanayake, J. & Pinzger, M. (2007). Improving defect prediction using temporal features and non linear models. Em *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE '07, pp. 11--18, New York, NY, USA. ACM.

Bevan, J.; Whitehead Jr., E. J.; Kim, S. & Godfrey, M. (2005). Facilitating software evolution research with kenyon. Em *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pp. 177--186, New York, NY, USA. ACM.

Blaschek, G. (1985). Statische programmanalyse. *Elektronische Rechenanlagen*, 27(2):89--95.

Brglez, F. (1984). On testability of combinational networks. Em *Proceedings of International Symposium on Circuits and Systems*, pp. 221--225.

Briand, L.; Daly, J. & Wust, J. (1999). A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1):91 –121.

Briand, L. C. & Wüst, J. (2001). The impact of design properties on development cost in object-oriented systems. Em *Proceedings of the 7th International Symposium on Software Metrics*, METRICS '01, pp. 260--, Washington, DC, USA. IEEE Computer Society.

Cameron, A. C. & Trivedi, P. K. (1986). Econometric models based on count data: Comparasions and applications of some estimators and tests. *Journal of Applied Econometrics*, 1:29--93.

Campenhout, D. V.; Al-Asaad, H.; Hayes, J. P.; Mudge, T. & Brown, R. B. (1998). High-level design verification of microprocessors via error modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 3(4):581--599.

Campenhout, D. V.; Mudge, T. & Hayes, J. P. (2000). Collection and analysis of microprocessors design errors. *IEEE Design & Test of Computers*, 17(4):51--60.

Canfora, G. & Cerulo, L. (2005). Impact analysis by mining software and change request repositories. Em *Software Metrics, 2005. 11th IEEE International Symposium*, pp. 9 pp. –29.

Card, D. N. & Glass, R. L. (1990). *Measuring Software Design Quality*. Prentice-Hall, Inc.

Cardoso, T. N. C.; do Val, C. G.; Nacif, J. A. M.; Fernandes, A. O. & Coelho Jr., C. N. (2008). Assertion based fault-tolerant processor: How to recover from design errors. Em *Proceedings of IFIP International Conference on Very Large Scale Integration System on a Chip VLSI-SoC*, pp. 445–450.

Cardoso, T. N. C.; Nacif, J. A. M.; Fernandes, A. O. & Coelho Jr., C. N. (2009). Bugtracer: A system for integrated circuit development tracking and statistics retrieval. Em *LATW '09: Proceedings of the 10th IEEE Latin-American TestWorkshop*. IEEE Computer Society.

Carter, H. B. & Hemmady, S. G. (2007). *Metrics Driven Design Verification*. Springer.

Chatman, V. V. (1995). Change-points: A proposal for software productivity measurement. *Journal of Systems and Software*, 31(1):71 – 91.

Chidamber, S. R. & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476--493.

Cistel Technology Inc. (1997). Verilog Source Code Complexity Analysis.

Cistel Technology Inc. (1998). Verilog Source Code Complexity Analysis.

Cohen, J.; Cohen, P.; West, S. G. & Aiken, L. S. (2003). *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Lawrence Erlbaum, Associate Publishers.

Coleman, D.; Ash, D.; Lowther, B. & Oman, P. (1994a). Using metrics to evaluate software system maintainability. *Computer*, 27(8):44 –49.

Coleman, D.; Ash, D.; Lowther, B. & Oman, P. (1994b). Using metrics to evaluate software system maintainability. *Computer*, 27(8):44--49.

Constantinides, K.; Mutlu, O. & Austin, T. (2008). Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation. Em *Proceedings of the 41st International Symposium on Microarchitecture (MICRO)*.

Crawley, M. J. (2007). *The R Book*. Wiley.

D'Ambros, M.; Lanza, M. & Robbes, R. (2010). An extensive comparison of bug prediction approaches. Em *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pp. 31 –41.

David A. Patterson, John L. Hennessy (2005). *Organização e Projeto de Computadores: A Interface Hardware/Software*. Campus.

Easterbrook, S.; Singer, J.; Storey, M.-A. & Damian, D. (2008). *Guide to Advanced Empirical Software Engineering*, capítulo Selecting Empirical Methods for Software Engineering Research, pp. 285--311. Springer London.

Emam, K. E.; Benlarbi, S.; Goel, N. & Rai, S. N. (2001a). Comparing case-based reasoning classifiers for predicting high risk software components. *J. Syst. Softw.*, 55:301--320.

Emam, K. E.; Melo, W. & Machado, J. C. (2001b). The prediction of faulty classes using object-oriented design metrics. *J. Syst. Softw.*, 56:63--75.

Fenton, N. E. & Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co.

Fischer, M.; Pinzger, M. & Gall, H. (2003). Populating a release history database from version control and bug tracking systems. Em *ICSM '03: Proceedings of the International Conference on Software Maintenance*, p. 23, Washington, DC, USA. IEEE Computer Society.

FLOSS Metrics Consortium (2010). FLOSS Metrics Final Report. Disponível em: `http://flossmetrics.org`.

Foster, H.; Krolnik, A. C. & Lacey, D. J. (2004). *Assertion-Based Design*. Kluwer Academic Publishers.

Frantzeskou, G.; Stamatatos, E.; Gritzalis, S. & Katsikas, S. (2006). Effective identification of source code authors using byte-level information. Em *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pp. 893--896, New York, NY, USA. ACM.

Freund, J. E. & Simon, G. A. (1996). *Statistics: A First Course*. Prentice Hall.

German, D. (2004). Mining CVS repositories, the softChange experience. Em *Proc. Int'l Workshop on Mining Software Repositories (MSR)*, pp. 17--21.

German, D. M.; Di Penta, M.; Gueheneuc, Y.-G. & Antoniol, G. (2009). Code siblings: Technical and legal implications of copying code between applications. Em *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pp. 81--90, Washington, DC, USA. IEEE Computer Society.

Glass, R. L.; Vessey, I. & Ramesh, V. (2002). Research in software engineering: An analysis of the literature. *Journal of Information and Software Technology*, 44(8):491--506.

Goldstein, L. H. & Thigpen, E. L. (1988). Scoap: Sandia controllability/observability analysis program. Em *Papers on Twenty-five years of electronic design automation*, 25 years of DAC, pp. 397--403, New York, NY, USA. ACM.

Gousios, G. & Spinellis, D. (2009). A platform for software engineering research. *Mining Software Repositories, International Workshop on*, 0:31–40.

Graves, T. L.; Karr, A. F.; Marron, J. S. & Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653--661.

Gyimóthy, T.; Ferenc, R. & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31:897–910.

Halstead, M. H. (1977). *Elements of Software Science*. Elsevier.

Hampton, M. & Petithomme, S. (2007). Leveraging a commercial mutation analysis tool for research. Em *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pp. 203--209, Washington, DC, USA. IEEE Computer Society.

Harrel, F. E. (2001). *Regression Modeling Strategies*. Springer.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. Em *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 78--88, Washington, DC, USA. IEEE Computer Society.

Hassan, A. E. & Holt, R. C. (2005). The top ten list: Dynamic fault prediction. Em *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 263--272, Washington, DC, USA. IEEE Computer Society.

Hatton, L. (1998). Does oo sync with how we think? *Software, IEEE*, 15(3):46 –54.

Hennessy, J. L. & Patterson, D. A. (2003). *Arquitetura de Computadores: Uma Abordagem Quantitativa*. Campus.

Henry, S. & Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510--518.

Hilbe, J. M. (2007). *Negative Binomial Regression*. Cambridge University Press.

Holzer, M. & Rupp, M. (2006). Static code analysis of functional descriptions in systemc. Em *DELTA '06: Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications*, pp. 243--248, Washington, DC, USA. IEEE Computer Society.

Hosmer, D. W. & Lemeshow, S. (2000). *Applied Logistic Regression*. John Wiley & Sons, Inc.

Howison, J.; Conklin, M. & Crowston, K. (2006). FLOSSmole: A Collaborative Repository for FLOSS Research Data and Analyses. *International Journal of Information Technology and Web Engineering*, 1(3).

hui Chang, K.; Markov, I. L. & Bertacco, V. (2009). *Functional Design Errors in Digital Circuits: Diagnosis, Correction and Repair*. Springer.

IEEE (1990). *IEEE Standard 610.12 - Glossary of software engineering terminology*. IEEE.

Intel (2011). Intel identifies chipset design error, implementing solution. Disponível em: `http://www.intel.com/pressroom`.

ISO/IEC (2004). 9126: 2004 software engineering - product quality - quality model. Relatório técnico, International Organization for Standardization.

ISO/IEC (2006). 17726: 2006 software engineering - software life cycle processes - maintenance. Relatório técnico, International Organization for Standardization.

ITRS (2009). International technology roadmap for semiconductors - 2009 edition. http://www.itrs.net.

Jackson, E. J. (2003). *A Users Guide to Principal Components*. John Wiley & Sons Inc.

Janes, A.; Scotto, M.; Pedrycz, W.; Russo, B.; Stefanovic, M. & Succi, G. (2006). Identification of defect-prone classes in telecommunication software systems using design metrics. *Information Sciences*, 176(24):3711 – 3734.

Jasper Design Automation, Inc. (2010). JasperGold Verification System and JasperCore Command Reference Manual.

Johnson, P. M.; Kou, H.; Paulding, M.; Zhang, Q.; Kagawa, A. & Yamashita, T. (2005). Improving software development management through software project telemetry. *IEEE Softw.*, 22:76--85.

Johnson, R. A. & Wichern, D. W. (2001). *Applied Multivariate Statiscal Analysis*. Prentice Hall.

Kan, S. H. (2003). *Metrics and Models in Software Quality Engineering*. Addison Wesley Profesional.

Kim, S.; Whitehead Jr., E. J. & Bevan, J. (2005). Analysis of signature change patterns. *SIGSOFT Softw. Eng. Notes*, 30:1--5.

Kim, S.; Zimmermann, T.; Whitehead Jr., E. J. & Zeller, A. (2007). Predicting faults from cached history. Em *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pp. 489--498, Washington, DC, USA. IEEE Computer Society.

King, G. (1988). Statistical Models for Political Science Event Counts: Bias in Conventional Procedures and Evidence for the Exponential Poisson Regression Model. *American Journal of Political Science*, 32:838--863.

Klecka, W. R. (1980). *Discriminant Analysis*. Sage Publications.

Lehner, F.; Dumke, R. & Abran, A. (1992). *Software Metrics- Research and Practice in Software Measurement*. Gabler-Verlag.

Lloyd, C. J. (1999). *Statistical Analysis of Categorical Data*. Wiley-Interscience.

Long, J. S. (1997). Regression models for categorical and limited dependent variables. *Advanced Quantitative Techniques in the Social Sciences*, 7:Sage Publications.

Manners, D. (2008). Leakage and verification costs both continue to rise, says cadence. Disponível em: `http://www.electronicsweekly.com/Articles/2008/10/23/44769/leakage-and-verification-costs-both-continue-to-rise-says-cadence.htm`.

Mäntylä, M.; Vanhanen, J. & Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. Em *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pp. 381--, Washington, DC, USA. IEEE Computer Society.

Mastretti, M. (1995). Vhdl quality: synthesizability, complexity and efficiency evaluation. Em *EURO-DAC '95/EURO-VHDL '95: Proceedings of the conference on European design automation*, pp. 482--487, Los Alamitos, CA, USA. IEEE Computer Society Press.

Maurer, S. (2009). HCT Complexity Tool Documentation.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308--320.

McCullagh, P. & Nelder, J. A. (1989). *Generalized Linear Models*. Chapman and Hall/CRC.

Meixner, A.; Bauer, M. E. & Sorin, D. (2007). Argus: Low-cost, comprehensive error detection in simple cores. Em *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pp. 210--222, Washington, DC, USA. IEEE Computer Society.

Menzies, T.; Stefano, J. S. D. & Chris Cunanan, R. C. (2004). Mining Repositories to assist in project planning and resource allocation. Em *Proceedings of the 1rst Workshop on Mining Software Repositories*.

Michael Weiss, G. M. (2007). *Emerging Free and Open Source Software Practices*, capítulo Ecology and Dynamics of Open Source Communities. IGI Global.

MIPS Technologies (2005). MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Mockus, A. (2009). Amassing and indexing a large sample of version control systems: Towards the census of public source code history. Em *Mining Software*

Repositories, 2009. MSR '09. 6th IEEE International Working Conference on, pp. 11 –20.

Mockus, A. & Votta, L. (2000). Identifying reasons for software changes using historic databases. Em *Software Maintenance, 2000. Proceedings. International Conference on*.

Moser, R.; Pedrycz, W. & Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. Em *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pp. 181--190, New York, NY, USA. ACM.

Munson, J. C. & Khoshgoftaar, T. M. (1992). The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423--433.

Nacif, J. A. M.; Silva, T.; Tavares, A. I.; Fernandes, A. O. & Coelho Jr., C. N. (2008). Efficient allocation of verification resources using revision history information. Em *DDECS*, pp. 190–194.

Nacif, J. A. M.; Silva, T. S. F.; Vieira, L. F. M.; Vieira, A. B.; Fernandes, A. O. & Coelho Jr., C. N. (2011a). A cache based algorithm to predict hdl modules faults. Em *LATW '11: Proceedings of the 12th Latin American Test Workshop*. IEEE.

Nacif, J. A. M.; Silva, T. S. F.; Vieira, L. F. M.; Vieira, A. B.; Fernandes, A. O. & Coelho Jr., C. N. (2011b). Tracking hardware evolution. Em *ISQED '11: Proceedings of the 12th International Symposium on Quality of Electronic Design*. IEEE.

Nagappan, N. & Ball, T. (2005a). Static analysis tools as early indicators of pre-release defect density. Em *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pp. 580--586, New York, NY, USA. ACM.

Nagappan, N. & Ball, T. (2005b). Use of relative code churn measures to predict system defect density. Em *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 284--292, New York, NY, USA. ACM.

Nagappan, N.; Ball, T. & Zeller, A. (2006). Mining metrics to predict component failures. Em *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pp. 452--461, New York, NY, USA. ACM Press.

Nejmeh, B. A. (1988). Npath: a measure of execution path complexity and its applications. *Commun. ACM*, 31(2):188--200.

Ohlsson, N. & Alberg, H. (1996). Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886--894.

Opencores (2011). Opencores. Disponível em: `http://www.opencores.org`.

Ostrand, T. J.; Weyuker, E. J. & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340--355.

Papoulis, A. (1991). *Probability, Random Variables, and Stochastic Processes*. McGraw Hill College Div.

Peled, D. A. (2001). *Software reliability methods*. Springer.

Protheroe, D. & Pessolano, F. (2000). An objective measure of digital system design quality. Em *ISQED '00: Proceedings of the 1st International Symposium on Quality of Electronic Design*, p. 227, Washington, DC, USA. IEEE Computer Society.

Purushothaman, R. (2005). Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511--526. Member-Dewayne E. Perry.

Reorda, M. S.; Sterpone, L.; Violante, M.; Portela-Garcia, M.; Lopez-Ongil, C. & Entrena, L. (2006). Fault injection-based reliability evaluation of sopcs. Em *ETS '06: Proceedings of the Eleventh IEEE European Test Symposium*, pp. 75--82, Washington, DC, USA. IEEE Computer Society.

Robles, G.; Koch, S. & Gonzalez-Barahona, J. M. (2004). Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. Em

Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS), Edinburg, Scotland, UK.

Scacchi, W. (2004). Free and open source development practices in the game community. *Software, IEEE*, 21(1):59 – 66.

Schafers, M. (1995). Measuring the complexity of hdl models. pp. 113–116.

Seber, G. A. F. & Wild, C. J. (2003). *Nonlinear Regression*. Wiley-Interscience.

Seth, S. C.; Pan, L. & Agrawal, V. D. (1985). Predict–probabilistic estimation of digital circuit testability. Em *Fault-Tolerant Comput. Symp. (FTCS-15) Dig. Paper*, pp. 220--225.

Shihab, E.; Jiang, Z. M. & Hassan, A. (2009). On the use of internet relay chat (irc) meetings by developers of the gnome gtk+ project. Em *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pp. 107 –110.

Siegel, S. & Castellan, N. J. (1988). *Nonparametrics Statistics for the Behavioral Sciences*. McGraw-Hill.

Sjoberg, D. I. K.; Dyba, T. & Jorgensen, M. (2007). The future of empirical methods in software engineering research. Em *FOSE '07: 2007 Future of Software Engineering*, pp. 358--378, Washington, DC, USA. IEEE Computer Society.

Śliwerski, J.; Zimmermann, T. & Zeller, A. (2005). When do changes induce fixes? Em *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pp. 1--5, New York, NY, USA. ACM.

Snyder, W. (2010). Verilog-Perl manpage. Disponível em: `http://www.veripool.org/wiki/verilog-perl/Manual-verilog-perl`.

Stapleton, W. & Tobin, P. (2009). Verification problems in reusing internal design components. Em *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pp. 209--211, New York, NY, USA. ACM.

Subramanyam, R. & Krishnan, M. (2003). Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *Software Engineering, IEEE Transactions on*, 29(4):297 – 310.

Sun Microsystems, Inc. (2006). Opensparc^{TM}t1 microarchitecture specification. Disponível em: `http://www.opensparc.net`.

The Bugzilla Team (2008). The bugzilla guide - 3.0.4 release. Disponível em `http://www.bugzilla.org/docs/3.0/html/`.

Torroja, Y.; Lopez, C.; García, M.; Riesgo, T.; de la Torre, E. & Uceda, J. (1999). Ardid: A tool for the quality analysis of vhdl based designs. Em *Forum on Design Languages*.

Velev, M. N. (2003). Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs. Em *ITC'03: International Test Conference*, pp. 138--147.

Wagner, I.; Bertacco, V. & Austin, T. (2008). Using field-repairable control logic to correct design errors in microprocessors. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(2):380–393.

Weyuker, E. J. (1988). Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14:1357--1365.

Wiemann, A. (2007). *Standardized Functional Verification*. Springer.

Wile, B.; Goss, J. & Roesner, W. (2005). *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann.

Wohlin, C. & Wesslen, A. (2000). *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Press.

Youden, W. (1950). Index for rating diagnostics tests. *Cancer*, 3:32--35.

Zimmermann, T. (2008). *Changes and Bugs Mining and Predicting Development Activities*. Phd thesis, Saarland University.

Zimmermann, T.; Nagappan, N.; Gall, H.; Giger, E. & Murphy, B. (2009). Cross-project defect prediction. Em *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM.

Zimmermann, T.; Nagappan, N. & Zeller, A. (2008). *Software Evolution*, capítulo Predicting Bugs from History, pp. 69--88. Springer.

Zimmermann, T.; Premraj, R. & Zeller, A. (2007). Predicting defects for eclipse. Em *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pp. 9--, Washington, DC, USA. IEEE Computer Society.

Zimmermann, T. & Weißgerber, P. (2004). Preprocessing cvs data for fine-grained analysis. Em *Proceedings of the First International Workshop on Mining Software Repositories*, pp. 2--6.

Apêndice A

Publicações durante o doutorado

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.

4. BUGTRACER: A System for Integrated Circuit Development Tracking and Statistics Retrieval. In: Latin American Test Workshop, 2009, Búzios. Proceedings of the 10th Latin American Test Workshop, 2009.

5. Efficient Allocation of Verification Resources using Revision History Information. In: Design and Diagnostics of Electronic Circuits and Systems, 2008, Bratislava. Proceedings of the 2008 IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2008. p. 190-194.

6. Prediction of Design Errors using Revision History Information. In: IFIP VLSI SoC PhD Forum, 2008, Rhodes. Proceedings of 16th IFIP VLSI-SoC PhD Forum, 2008.

7. Assertion based fault-tolerant processor: How to recover from design errors. In: IFIP International Conference on Very Large Scale Integration and System-on-Chip, 2008, Rhodes. Proceedings of the 16th IFIP International Conference on Very Large Scale Integration and System-on-Chip, 2008.

8. Memory Aspects of Dual Core Processor Design. In: Chip on the Pampas Student Forum, 2008, Gramado. Proceedings of the Chip in the Pampas Student Forum, 2008.

9. On-chip Property Verification using Assertion Processors. VLSI SoC: From Systems to Chips. 1 ed. Boston: Springer, 2006, v. 200, p. 101-117.

10. High-level Automatic Verification Methodology for Computer Graphics Cores. In: Latin American Test Workshop, 2006, Buenos Aires. Proceedings of the 7th Latin American Test Workshop, 2006.

11. Signaling Chip Errors with SNMP. In: Latin American Test Workshop, 2006, Buenos Aires. Proceedings of the 7th Latin American Test Workshop, 2006.

12. System-level Dynamic Power Management Techniques for Communication Intensive Devices. In: IFIP International Conference on Very Large Scale Integration: VLSI-SoC 2006, 2006, Nice. Proceedings of the 14th International Conference on Very Large Scale Integration.

13. When bits are not bits and bytes are not bytes: Validating computer graphics cores at higher level of abstraction. In: Latin American Test Workshop, 2005, Salvador. Proceedings of the 6th Latin American Test Workshop, 2005.

14. Low Power/High Performance Self-Adapting Sensor Node Architecture. In: Emerging Techniques and Factory Automation, 2005, Catania, 2005.

15. Reconfigurable Sensor Node for Low Power/High Performance Applications. In: VLSI-SoC, 2005, Perth, 2005.

Apêndice B

Especificação do Trabalho Prático

Este Apêndice apresenta parte da especificação do trabalho prático das disciplinas Arquitetura de Computadores (mestrado/doutorado), Organização de Computadores II (graduação). O trabalho consiste na implementação de um processador MIPS32 na linguagem de descrição de hardware Verilog. A especificação completa do MIPS32 é composta por 17 módulos agrupados em 10 entregas ao longo do semestre. A Seção B.1 apresenta a especificação de um dos módulos do MIPS32, a Unidade Lógica Aritmética. Em seguida, a Seção B.2 ilustra a implementação de um dos grupos da especificação do mesmo módulo do MIPS32. Finalmente, na Seção B.3, um fragmento do testbench utilizado para verificar o módulo Unidade Lógico Aritmética é apresentado.

B.1 Especificação do Módulo - Unidade Lógica Aritmética

A unidade lógica e aritmética (ALU - *Arithmetic Logic Unit*) é um módulo que agrupa as operações aritméticas soma e subtração, e lógicas, NOT (negação), AND (e), OR (ou), NOR (negação da operação e) e XOR (ou exclusivo). Essas operações são utilizadas para calcular posições de memória ou valores de resultado que serão armazenados em registradores. A Tabela B.1 apresenta os sinais do módulo ALU. As condições de overflow são apresentadas na Tabela B.2.

Tabela B.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` é 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 B.2 para operações com sinal.

Tabela B.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

B.2 Implementação do Módulo - Unidade Lógica Aritmética

```
module Alu(

    input      [31:0] a,
    input      [31:0] b,
    input      [2:0]  op,
    input             unsig,
    output     [31:0] aluout,
    output            compout,
    output            overflow

    );

    parameter AND = 3'b000;
    parameter OR  = 3'b001;
    parameter ADD = 3'b010;
    parameter NOR = 3'b100;
    parameter XOR = 3'b101;
    parameter SUB = 3'b110;

    reg [31:0] result;
    reg alu_compout;
    reg alu_overflow;

    assign aluout   = result[31:0];
    assign compout  = alu_compout;
    assign overflow = alu_overflow;

    always@(a or b or op or unsig or result)
    begin
```

```
case(op)
    AND : result = a & b;
    OR  : result = a | b;
    ADD : result = a + b;
    NOR : result = ~(a | b);
    XOR : result = a ^ b;
    SUB : result = a - b;
    default : result = 32'bx;
endcase

if(unsig == 1)
begin
    alu_compout = a < b;
    alu_overflow = 0;
end
else
begin
    if(op == ADD)
    begin
        if(a[31] == 0 & b[31] == 0 & result[31] == 1)
            alu_overflow = 1;
        else if(a[31] == 1 & b[31] == 1 & result[31] == 0)
            alu_overflow = 1;
        else
            alu_overflow = 0;
    end
    else if(op == SUB)
    begin
        if(a[31] == 0 & b[31] == 1 & result[31] == 1)
            alu_overflow = 1;
        else if(a[31] == 1 & b[31] == 0 & result[31] == 0)
            alu_overflow = 1;
        else
```

```
                alu_overflow = 0;
            end
            else alu_overflow = 0;

            if(a[31] == 1 & b[31] == 0)
            begin
                alu_compout = 1; // a < b
            end
            else if(a[31] == 0 & b[31] == 1)
            begin
                alu_compout = 0; // a > b
            end
            else if((a[31] == 0 & b[31] == 0) | (a[31] == 1 & b[31] == 1))
            begin
                alu_compout = a < b;
            end
            else alu_compout = 1'bx;
        end
    end

endmodule
```

B.3 Fragmento de Testbench do Módulo - Unidade Lógica Aritmética

```
‘include "../rtl/Alu.v"

module tb_alu ();

  parameter AND = 3’b000;
  parameter OR  = 3’b001;
  parameter ADD = 3’b010;
  parameter NOR = 3’b100;
  parameter XOR = 3’b101;
  parameter SUB = 3’b110;

    reg               clock;
    reg               teste_clock;

    reg        [31:0] a;
    reg        [31:0] b;
    reg        [2:0]  op;
    reg               unsig;
    wire       [31:0] aluout;
    wire              compout;
    wire              overflow;

    Alu alu1(
        .a(a),
        .b(b),
        .op(op),
        .unsig(unsig),
        .aluout(aluout),
        .compout(compout),
```

```
        .overflow(overflow)
    );

    always #1 clock = ~clock;

    always@(posedge clock)
    begin
        if(teste_clock == 1)
          begin
          a = 50;
          b = 200;
          op = ADD;
          unsig = 0;
          end
    end

    always@(negedge clock)
    begin
        if(teste_clock == 1)
            begin
            $display("a:        %d", a);
            $display("b:        %d", b);
            $display("op:       %d", op);
            $display("aluout:   %d", aluout);
            $display("compout:  %b", compout);
            $display("overflow: %b", overflow);
            $finish;
            end
    end

    initial
    begin
```

```
        clock = 0;
        teste_clock = 0;

        $display("++ Operacoes aritmeticas com sinal");
        unsig = 0;

            $display("// Teste quando ocorre overflow");

            a = 2147483647;
            b = 128;
            op = ADD;
            #1
            $display("a >= 0, b >= 0 -> r < 0");
            $display("ADD e ADDI");
            $display("unsig: %b ", unsig);
            $display("a: %d  (2147483647)", a);
            $display("b: %d  (128)", b);
            $display("r: %d  (%b)", aluout, aluout);
            $display("overflow: %b", overflow);
            $display("compout:  %b", compout);
            $display;


        teste_clock = 1;

        #1000 $finish;

    end

endmodule
```