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

# Uma Técnica para Compilar Sistemas Configuráveis com `#ifdefs` Baseada no Impacto da Mudança

Larissa Nadja Braz Brasileiro

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

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

Rohit Gheyi
(Orientador)

Campina Grande, Paraíba, Brasil

# Resumo

Sistemas condifuráveis tipicamente usam `#ifdefs` para denotar variabilidade. Gerar e compilar todas as configurações de um sistema pode consumir tempo. Uma alternativa consiste em usar analisadores sintáticos conscientes de variabilidade, como TypeChef. Porém, eles podem não escalar. Na prática, desenvolvedores podem utilizar estratégias de amostragem (*sampling*) para compilar apenas um subconjunto das configurações. Este trabalho propõe uma técnica para compilar sistemas configuráveis com `#ifdefs` baseada no impacto da mudança através da análise apenas das configurações impactadas por uma mudança de código. A técnica foi implementada em uma ferramenta chamada CHECKCONFIGMX, que reporta os novos erros de compilação introduzidos pela transformação. Um estudo empírico foi realizado para avaliar 3,913 transformações aplicadas aos 14 maiores arquivos dos sistemas configuráveis BusyBox, Apache HTTPD, e Expat. CHECKCONFIGMX encontrou 595 errors de compilação de 20 tipos introduzidos por 41 desenvolvedores em 214 *commits* (5.46% das transformações analisadas). No estudo realizado, a ferramenta reduziu ao menos 50% (uma média de 99%) o número de configurações compiladas em comparação à abordagem exaustiva, sem considerar *feature models*. CHECKCONFIGMX pode ajudar os desenvolvedores a reduzir o esforço de avaliar transformações de granularidade fina aplicadas a sistemas configuráveis com `#ifdefs`.

**Abstract**

Configurable systems typically use `#ifdefs` to denote variability. Generating and compiling all configurations may be time-consuming. An alternative consists of using variability-aware parsers, such as TypeChef. However, they may not scale. In practice, developers can use sampling strategies to compile only a subset of the configurations. We propose a change-centric approach to compile configurable systems with `#ifdefs` by analyzing only configurations impacted by a code change. We implement it in a tool called CHECKCONFIGMX, which reports the new compilation errors introduced by the transformation. We perform an empirical study to evaluate 3,913 transformations applied to the 14 largest files of BusyBox, Apache HTTPD, and Expat configurable systems. CHECKCONFIGMX finds 595 compilation errors of 20 types introduced by 41 developers in 214 commits (5.46% of the analyzed transformations). In our study, it reduces at least 50% (an average of 99%) the number of compiled configurations by comparing with the exhaustive approach without considering a feature model. CHECKCONFIGMX may help developers to reduce compilation effort to evaluate fine-grained transformations applied to configurable systems with `#ifdefs`.

# Agradecimentos

A Deus por me amparar nos momentos difíceis, me dar força interior para superar as dificuldades e mostrar os caminho nas horas incertas;

A minha mãe, Ester Braz, por todo amor, dedicação e companheirismo. Por me apoiar em todos os meus sonhos e agindo sem medir esforços para que se tornassem realidade;

Agradeço ao meu esposo Pedro Barbosa, pelo amor, amizade e apoio em todos os momentos. Obrigada por estar sempre presente, vibrando com minha felicidade e me dando forças nos momentos difíceis;

Aos meus avós, Eliezer e Rilda Braz, e à minha irmã, Safira Braz, pelo grande carinho, amor, preocupação, apoio e conselhos. Aos meus tios-irmãos por sempre me encorajarem e torcerem por mim;

Meus sinceros agradecimentos ao meu orientador, Rohit Gheyi, por me mostrar os passos da pesquisa, sempre com muita paciência, dedicação, empenho, vontade de ensinar e orientar. Agradeço por ter acreditado no meu potencial e por todas as oportunidades que me concedeu. Sem ele esse trabalho não seria possível;

Agradeço a Melina Mongiovi, grande amiga que acompanhou toda minha caminhada para realização deste trabalho. Obrigada pelo constante incentivo e ajuda durante o mestrado;

Aos professores Leopoldo Teixeira e Márcio Ribeiro pelas sugestões e contribuições neste trabalho;

Agradeço a todos os amigos do SPG, que contribuíram de alguma forma neste trabalho. Agradeço especialmente a Gustavo Soares e Reudismam Rolim pelas boas conversas, trocas de ideias e momentos de descontração;

Aos professores e funcionários da COPIN e do DSC;

A Capes pelo apoio e suporte financeiro fornecidos a este trabalho.

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG**

B823t Brasileiro, Larissa Nadja Braz.

Uma técnica para compilar sistemas configuráveis com #ifdefs baseada no impacto da mudança / Larissa Nadja Braz Brasileiro. – Campina Grande, 2016.

89f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

"Orientação: Prof. Rohit Gheyi".

1. Ciência da Computação. 2. Sistemas Configuráveis. 3. #ifdefs – Sistemas Configuráveis. I. Gheyi, Rohit. II. Título.

CDU 004(043)

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Lista de Códigos Fonte

# Capítulo 1

# Introdução

Atualmente, vários sistemas de software, como Linux[1] e Ghostscript,[2] são configuráveis [Siegmund et al., 2015]. Configurabilidade é uma forma de criar uma coleção de sistemas de software similares a partir de um conjunto de componentes de software usando meios de produção em comum [Donohoe, 2009]. As configurações desses sistemas podem ser combinadas, cada uma com diferentes funcionalidades e efeitos nos atributos de qualidade, gerando uma variabilidade de produtos similares.

Configurabilidade é normalmente implementada por pré-processadores C através da compilação condicional. Desenvolvedores preenchem o código utilizando derivativas de pré-processadores, como a macro `#ifdef`, para limitar escopos de configurações. Existem relatórios de sistemas industriais com milhares de configurações [Berger et al., 2013a] e extensivos exemplos de projetos de código aberto documentados em detalhes [Berger et al., 2013b], incluindo os sistemas BusyBox,[3] Bash[4] e Apache.[5]

Este capítulo inicia com a discussão do problema abordado neste trabalho (Seção 1.1). Em seguida, um exemplo motivante demonstra uma transformação que introduz um erro de compilação em determinadas configurações do sistema configurável (Seção 1.2). A solução proposta para melhorar o cenário atual é apresentada na Seção 1.3. Os estudos da avaliação da solução são apresentados (Seção 1.4). Além disso, as contribuições principais do trabalho

[1] https://www.linux.com/
[2] http://www.ghostscript.com/
[3] https://www.busybox.net/
[4] https://www.gnu.org/software/bash/
[5] http://www.apache.org/

são expostas (Seção 1.5). E, por fim, há uma visão geral da organização deste documento (Seção 1.6).

## 1.1 Problema

Alguns produtos de um sistemas configuráveis podem não compilar, uma vez que suas macros podem interagir de maneiras não triviais influenciando as funcionalidades das outras. Quando essas interações não são intencionais podem induzir à erros que se manifestam em determinadas configurações mas não em outras, ou que se manifestam de maneira diferente em diferentes configurações. Por exemplo, variáveis podem ser utilizadas fora do escopo em que foram definidos. Medeiros et al. [Medeiros et al., 2013] analisaram *releases* de 41 programas C com mais de 51 mil *commits* de oito sistemas configuráveis e encontrarem 24 erros de compilação diferentes. Eles detectaram arquivos que contém erros em todos os *commits*. Medeiros et al. [Medeiros et al., 2015a] afirmam que mais de 74% dos desenvolvedores acreditam que erros relacionados a configurações são mais difíceis de encontrar e mais críticos do que problemas que aparecem em todas as configurações.

Estudos abordam análises de sistemas configuráveis [Thum et al., 2014] através de extensões de técnicas clássicas [Apel et al., 2010; Bodden et al., 2013; Brabrand et al., 2012] e de checagem de modelos [Apel et al., 2011; Classen et al., 2010; Gruler et al., 2008]. A compilação de uma configuração individual pode ser realizada através de técnicas de análise de programas padrão [Abal et al., 2014]. Porém, compilar todas as configurações de um sistema configurável com `#ifdefs` talvez seja demorado e impraticável [Abal et al., 2014] em grandes sistemas, como o Linux.

A maior parte das ferramentas disponíveis, por exemplo GCC[6] e Clang,[7] não são conscientes de variabilidade e consideram apenas uma configuração por vez, requisitando mudanças importantes para lidar com mais de uma [Medeiros et al., 2016]. Enquanto que ferramentas conscientes de variabilidade, como TypeChef [Kästner et al., 2011], consideram todas as configurações do sistema para analisar código. Porém, essas ferramentas podem não escalar, uma vez que os sistemas configuráveis têm um possível espaço de configurações

[6] https://gcc.gnu.org/
[7] http://clang.llvm.org/

de tamanho exponencial.

Gazzillo and Grimm [Gazzillo and Grimm, 2012] afirmaram que o pré-processador do TypeChef não identifica várias interações entre as funcionalidades do pré-processador. Além disso, a ferramenta é um analisador limitado, pois desenvolvedores precisam não só reprojetar suas gramáticas com os combinadores da ferramenta, mas também tem que empregar corretamente as várias junções dos combinadores. Medeiros et al. [Medeiros et al., 2013] propuseram uma abordagem para melhorar este cenário através da criação de *stubs*. Porém, esta abordagem pode também não escalar devido ao tamanho do espaço de configurações.

Na prática, para lidar com o problema de escalabilidade, desenvolvedores usam estratégias de amostragem [Johansen et al., 2012; Oster et al., 2010; Perrouin et al., 2010] para compilar algumas configurações do sistema. Além disso, Medeiros et al. [Medeiros et al., 2015b] afirmam que a maioria dos problemas relacionados à configuração são detectados quando uma ou duas opções de configurações são habilitadas ou desabilitadas (por exemplo, ao usar os algoritmos maioria habilitada e maioria desabilitada), o que suporta a eficiência dessa abordagem. Porém, como a escolha de configurações não foca nas mudanças do código, essa abordagem pode deixar de detectar alguns erros localizados em configurações impactadas.

## 1.2 Exemplo Motivante

Essa seção mostra uma transformação aplicada ao BusyBox que introduz um erro de compilação em algumas configurações. BusyBox é um software com código livre que possui várias ferramentas *Unix* compactadas em um único arquivo executável. O Código 1.1 apresenta parte do arquivo `httpd.c` (commit d2277e2), do repositório do BusyBox que utiliza uma macro `#ifdef` para declarar a `struct pam_userinfo`. Essa `struct` só é declarada quando as macros `ENABLE_AUTH_MD5` e `ENABLE_PAM` são habilitadas. Mais tarde, no próximo *commit*, os desenvolvedores realizaram uma mudança no arquivo ao adicionar um trecho de código que utiliza essa `struct` quando a macro `ENALBLE_PAM` está habilitada. Porém, o código modificado não compila quando a macro `ENABLE_AUTH_MD5` é desabilitada e a macro `ENABLE_PAM` é habilitada. O compilador reporta a seguinte mensagem de erro: `variável tem tipo incompleto struct pam_userinfo`. O

Código 1.2 ilustra parte do código modificado (commit 7291755). Este tipo de erro também ocorre em outros sistemas configuráveis (Capítulo 4).

Foi realizada uma tentativa de executar o TypeChef no arquivo modificado real. Porém, executar essa ferramenta consome tempo uma vez que o arquivo `httpd.c` modificado contém 18 macros. Desta forma, analisadores sintáticos podem não escalar para avaliar este arquivo uma vez que eles podem ter configurações iniciais e processos de compilação custosos. Em adição, TypeChef foi executado em um exemplo simples apresentado no Código 1.2 e a ferramenta não detecta o erro. A abordagem proposta por Medeiros et al. [Medeiros et al., 2015b] não foca neste tipo de erro, portanto eles não detectam este erro de compilação.

Compilar todas as configurações de um arquivo como o `httpd.c`, que contém 18 macros, pode ser custoso mesmo que o *feature model* seja considerado e que proíba várias configurações. Um algoritmo de amostragem pode ser utilizado para verificar se uma algumas configurações compilam. Por exemplo, o algoritmo maioria-habilitada-desabilitada possui uma balança eficiente entre tamanho da amostra e capacidade de detecção de faltas sob diferentes hipóteses [Medeiros et al., 2016]. Porém, ao utilizar esta abordagem o erro de compilação não é detectado. O erro é exposto pela configuração em que apenas a macro `ENABLE_PAM` é habilitada. Em geral, os algoritmos de amostragem que não focam na mudança podem perder tempo em tentativas de compilar configurações que não são afetadas pela transformação. Para minimizar este problema, este trabalho propõe uma técnica para compilar sistemas configuráveis com `#ifdefs` baseada no impacto da mudança, que analisa apenas as configurações que podem ter sido impactadas pela transformação.

## 1.3 Solução

Este trabalho propõe uma técnica para compilar sistemas configuráveis com `#ifdefs` baseada no impacto da mudança. Ela analisa uma transformação realizada em um sistema configurável, identifica todas as configurações impactadas e compila apenas elas. A técnica executa essas atividades da seguinte forma: ela recebe dois arquivos de um sistema configuráveis C (original e modificado) como entrada. Depois, ela identifica o impacto da mudança entre o arquivo original e o modificado através da comparação textual dos arquivos. Em seguida, os resultados da comparação são analisados e as macros impactadas pela

```
#ifdef ENABLE_AUTH_MD5 &&
    ENABLE_PAM
   struct pam_userinfo {
      const char *name;
      const char *pw;
   };
#endif
```

Código Fonte 1.1: Trecho de código do arquivo `httpd.c` original.

```
#ifdef ENABLE_AUTH_MD5 &&
    ENABLE_PAM
   struct pam_userinfo {
      const char *name;
      const char *pw;
   };
#endif
#ifdef ENABLE_PAM
   struct pam_userinfo userinfo;
#endif
```

Código Fonte 1.2: Trecho de código do arquivo `httpd.c` modificado.

Figura 1.1: Trechos de código de commits consecutivos do arquivo `httpd.c`. Esta transformação introduz um erro de definição incompleta de tipo.

transformação são identificadas. Todas as configurações possíveis são geradas através das macros impactadas; e, os dois programas são compilados para cada uma delas. A técnica compara os erros de compilação encontrados durante as compilações e filtra os erros que foram detectados apenas no arquivo modificado. Em seguida, a técnica categoriza os erros de compilação de acordo com o tipo do erro e o elemento que o causou. Por fim, os erros de compilação introduzidos pela transformação e as configurações relacionadas são reportados para os desenvolvedores.

A técnica proposta foi implementada em uma ferramenta chamada CHECKCONFIGMX, que compila sistemas configuráveis com `#ifdefs`. CHECKCONFIGMX recebe dois arquivos de um sistema configurável com `#ifdefs`, analisa as mudanças entre eles, e indica as configurações que apresentam erros de compilação e quais são estes erros.

## 1.4 Avaliação

Um estudo empírico foi conduzido durante este trabalho para validar a técnica proposta e compilação de sistemas configuráveis com `#ifdefs`. O experimento deste estudo avaliou 163 transformações aplicadas aos seis maiores arquivos dos sitemas BusyBox, cinco mai-

ores arquivos do Apache HTTPD e três maiores arquivos do Expat. Estes arquivos foram escolhidos pois, potencialmente, têm a maior quantidade de construções de linguagem entre os arquivos dos sistemas.

CHECKCONFIGMX encontrou 595 erros de compilação em 214 transformações, que foram categorizados em 20 tipos diferentes de erros, como *uso de variável não declarada* e *definição incompleta de tipo*. Em adição, CHECKCONFIGMX reduz em pelo menos 50% (uma média de 99%) o número de configurações compiladas quando comparada à abordagem exaustiva, sem considerar *feature models* [Kang et al., 1990].

## 1.5 Contribuições

As contribuições principais deste trabalho são:

- Uma técnica para compilar sistemas configuráveis com `#ifdefs` baseada no impacto da mudança (Capítulo 3); e,
- Um estudo empírico considerando 3,913 pares do repositório Git do Busybox, Apache HTTPD e Expat, em relação ao impacto de mudanças de código no esforço para compilar (Capítulo 4).

## 1.6 Organização

Este documento está organizado da seguinte forma: o Capítulo 2 fundamenta os principais temas abordados durante este trabalho. O Capítulo 3 descreve a técnica proposta para compilar sistemas configuráveis com `#ifdefs`. O Capítulo 4 descreve a avaliação da técnica, realizada através de um estudo empírico. Em seguida, o Capítulo 5 apresenta os principais trabalhos relacionados. O Capítulo 6 apresenta as principais conclusões do trabalho. Por fim, o Apêndice A apresenta os passos seguidos na revisão sistemática realizada durante o trabalho.

# Capítulo 2

# Fundamentação Teórica

Este capítulo aborda os principais conceitos que foram abordados na execução deste trabalho. Inicialmente conceitos de sistemas configuráveis são apresentados (Seção 2.1). Em seguida, pré-processadores com foco na linguagem de programação C (Seção 2.2) e erros de compilação relacionados à configurações (Seção 2.3) são abordados. Mais adiante, os temas análise consciente de variabilidade (Seção 2.4) e análise por amostragem (Seção 2.5) são discutidos. E, por fim, breves contextualizações sobre análise de impacto da mudança (Seção 2.6) e testes de mutação (Seção 2.7) são abordados.

## 2.1 Sistemas Configuráveis

Linhas de Produto de *Software* ( *LPS*) são a métodos, ferramentas e técnicas da engenharia de *software* para criar uma coleção de sistemas de *software* similares a partir de um conjunto compartilhado de artefatos utilizando um meio de produção em comum [Clements and Northrop, 2009; Donohoe, 2009; Pohl et al., 2005]. As *LPS*s são sistemáticos e utilizam conceitos, teorias e artefatos, como *feature model* e conhecimentos de configurações [Cirilo et al., 2011; Teixeira et al., 2013]. Tem seus princípios baseados em fabricantes de automóveis, que permitem a produção em massa mais barata do que a criação de produtos individuais. Esses fabricantes utilizam uma plataforma em comum para obter produtos que podem ser customizados de acordo com a necessidade de consumidores específicos ou de segmentos de mercado [Clements and Northrop, 2009].

Uma *LPS* pode ser um conjunto de sistemas de *softwares* e similares que compartilham

uma coleção de funcionalidades em comum, satisfazendo as necessidades de consumidores específicos ou de segmentos do mercado. Os sistemas configuráveis são desenvolvidos a partir de um conjunto de recursos essenciais, que são documentos, especificações, componentes, e outros artefatos de *software* que naturalmente tornam-se altamente reusáveis durante o desenvolvimento de cada sistema específico na linha de produto [Clements, 2002; Gomaa, 2004; Pohl et al., 2005; van der Linden et al., 2007]. Dado o crescimento exponencial de produtos em função do número de opções de configuração, garantir que todos os produtos de uma linha satisfazem dadas propriedades não é uma questão trivial [Machado et al., 2012]. Em particular, isto é válido para a compilação completa de uma *LPS*.

Acerca da linguagem C, os desenvolvedores frequentemente usam o pré-processador C para lidar com variabilidade, resolver problemas de portabilidade, e implementar configurações de um sistema deste tipo [Spencer and Collyer, 1992]. Desta forma, os desenvolvedores envolvem o código fonte C com diretivas de pré-processamento, tais como `#ifdef`, `#else`, `#elif` e `#endif`. No entanto, sistemas C reais não usam os conceitos e artefatos de linha de produto de software e seu desenvolvimento não é sistemático. Assim, neste trabalho, utiliza-se o termo sistemas configuráveis para fazer referência a projetos, tais como Apache e BusyBox, que utilizam o pré-processador C para lidar com configurabilidade. A próxima seção apresenta uma visão geral sobre o pré-processador C.

## 2.2 Pré-processador C

Em algumas linguagens, por exemplo C e PL/I, há uma fase de tradução conhecida como pré-processamento. Um pré-processador é um sistema configurável que processa os dados de entrada para produzir uma saída que é usado como entrada para outro sistema configurável [Aho et al., 2007]. A saída é dita ser uma forma pré-processada dos dados de entrada, muitas vezes utilizadas por alguns programas subsequentes como compiladores [Aho et al., 2007]. A quantidade e o tipo de processamento feito dependem da natureza do pré-processador. Alguns pré-processadores são capazes apenas de executar substituições textuais relativamente simples e expansÃ$\mu$es de macro, enquanto outros têm o poder de linguagens de programação completas [Aho et al., 2007].

Pré-processadores são amplamente utilizados na prática. São essencialmente utilizados

em todos os projetos escritos em C, incluindo vários banco de dados e sistemas operacionais bem conhecidos. O pré-processador é executado durante o processo de compilação e realiza três atividades interativas [Medeiros, 2016]: lexicamente inclui arquivos (`#include`); expande as macros (`#define`); e, condicionalmente exclui partes do código de acordo com a definição das macros (`#ifdef` e `#if`). A inclusão de arquivos e expansÃ$\mu$es de macros `#ifdefs` são relativamente bem compreendidos e existem estratégias de mitigação disponíveis na literatura [Ernst et al., 2002; Kumar et al., 2012; Mennie and Clarke, 2004; Spinellis, 2003].

```
1  int main() {
2  #ifdef M1
3     int x;
4  #endif
5
6  #ifdef M2
7     int y;
8  #endif
9
10 #ifdef M3
11    int z;
12 #endif
13 }
```

Código Fonte 2.1: Código antes do pré-processamento.

```
1  int main() {
2
3     int x;
4
5
6
7     int y;
8
9
10
11    int z;
12
13 }
```

Código Fonte 2.2: Código pré-processado com todas as configurações habilitadas.

```
1  int main() {
2
3     int x;
4
5
6
7
8
9
10
11    int z;
12
13 }
```

Código Fonte 2.3: Código pré-processado com `M1` e `M3` habilitadas, e `M2` desabilitada.

Figura 2.1: Exemplo de um sistema configurável com `#ifdefs`.

Usando o pré-processador C, desenvolvedores lidam com um único código com diferentes configurações. Neste contexto, uma configuração representa o código a ser compilado de acordo com estado das macros (habilitadas e desabilitadas) presentes no código-fonte durante o seu pré-processamento. Desta forma, após o pré-processamento, apenas os códigos sob as macros habilitadas serão compilados. A Figura 2.1 demonstra o uso de `#ifdefs` para definir configurações um sistema configurável. O Código 2.1 mostra o código do sis-

tema antes do pré-processamento. Ao habilitar todas as macros, as variáveis `x`, `z` e `y` estarão declaradas dentro do escopo do código durante a compilação do sistema. Enquanto que ao habilitar apenas `M1` e `M3` (Código 2.3), apenas as váriaveis `x` e `z` são declaradas. Durante a compilação deste último código, a variável `y` não estará declarada no escopo do código.

## 2.3 Erros de Compilação Relacionados à Configurações

Um erro de compilação é relacionado à uma configuração é um erro causado pelo uso de diretivas de pré-processamento. No contexto deste trabalho, um erro de compilação é um resultado diferente do esperado, como uso de variável não declarada, durante a compilação do sistema configurável [IEEE, 1990].

Um erro de compilação relacionado à uma configuração é um erro que não ocorre em todas as configurações do código fonte. Desta forma, para definir se um erro é relacionado à uma configuração, é preciso identificar ao menos uma configuração válida onde o erro ocorre [Medeiros et al., 2016]. Observando o Código 2.4, é possível notar que há um erro de compilação na linha 3. O `array y` está sendo inicializado com uma variável do tipo `int` quando deveria ser um inicializador de lista ou uma `string` literal. Porém, este erro só ocorrerá quando o trecho de código for compilado, o que ocorre apenas quando `M1` está habilitada. Perceba que quando todas as macros forem desabilitadas, o erro não ocorrerá. Desta forma, este erro de compilação está relacionado à configurações com `M1` habilitada.

```
1 #ifdef M1
2     int x;
3     int y [1] = x;
4 #endif
```

Código Fonte 2.4: Exemplo de erro de compilação em um sistema configurável.

## 2.4 Análise Consciente de Variabilidade

Um analisador consciente de variabilidade leva em consideração as configurações de um sistema configurável. A análise gera árvores de sintaxe abstratas ampliadas com infor-

mações de variabilidade, garantindo a ausência de erros de sintaxe em todas as configurações [Medeiros, 2016]. Analisadores que implementam essa abordagem, como TypeChef e SuperC [Gazzillo and Grimm, 2012], lidam com interações entre macros, inclusão de arquivos (por exemplo através da diretiva `#include`), e compilação condicional. Em vez de considerar as definições e expansÃ$\mu$es de macro, e inclusÃ$\mu$es de arquivos entrelaçados, essa abordagem realiza pré-processamento parcial. Desta forma, ela pré-processa esses fatores mas mantém informações de variabilidade para análises futuras [68].

Considere o Código 2.5. A Figura 2.2 apresenta a árvore sintática abstrata do código gerada a partir de um *statement* condicional. Observe que existe um nó de escolha M1 que controla ambas as configurações: (1) A macro habilitada, e (2) a macro desabilitada. Desta forma, usando a árvore de sintaxe abstrata alinhada com informações variabilidade, pode-se procurar por erros relacionados à configuração em todas as configurações [Medeiros, 2016]. A Figura 2.3 apresenta uma captura de tela da execução da ferramenta TypeChef para análise do Código 2.5, observe que a ferramenta realiza análises léxica e sintática, e realiza checagem de tipo considerando todas as configurações do sistema.

```
1 if (c1
2 #ifdef M1
3     && c2
4 #endif
5 ) { }
```

Código Fonte 2.5: Trecho de código de um sistema configurável.

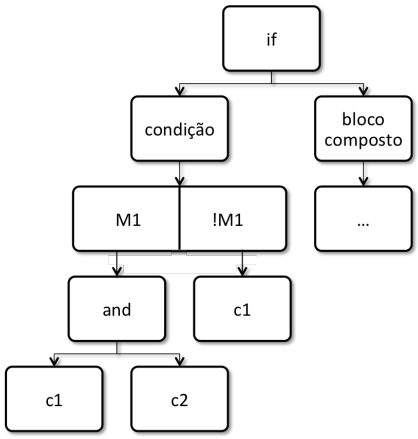


Figura 2.2: Árvore sintática abstrata do trecho de Código 2.5 alinhada com informações de variabilidade.

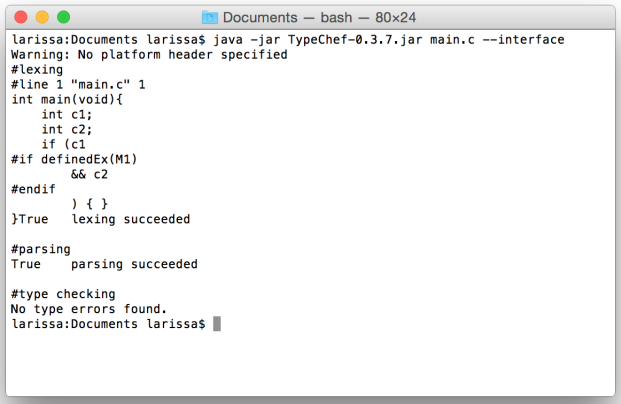


Figura 2.3: Captura de tela da execução da ferramenta TypeChef para análise do sistema configurável do Código 2.5

## 2.5 Análise por Amostragem

Desenvolvedores utilizam análise de amostragem (*sampling*) como abordagem viável para reutilizar ferramentas do estado da arte, como GCC [Johansen et al., 2012; Oster et al., 2010; Perrouin et al., 2010]. Assim, no lugar de analisar todas as configurações, a abordagem seleciona um subconjunto de configurações para analisar individualmente. A eficácia da amostragem para a detecção de erros relacionados à configurações depende significativamente de como as amostras são selecionadas [Medeiros, 2016].

O algoritmo de amostragem *cobertura de statement* [Tartler et al., 2011] seleciona um conjunto de configurações onde cada bloco de código opcional é ativado pelo menos uma vez [Tartler et al., 2014]. Observe a Figura 2.4, ao habilitar todas as macros, o algoritmo garante que os blocos `codigo1`, `codigo2` e `codigo4` são analisados pelo menos uma vez. Porém, para analisar o bloco `codigo3` é necessário uma outra configuração, como `M1` e `M3` habilitadas, e `M2` desabilitada B [Medeiros, 2016]. Nesta abordagem, a inclusão de cada bloco de código opcional pelo menos uma vez não garante que todas as combinações possíveis de blocos individuais de código opcional são considerados.

O algoritmo de amostragem *t-wise* abrange todas as $t$ combinações de macros de pré-processamento: a abordagem *pair-wise* verifica todos os pares de macros ($t = 2$) [Johansen et al., 2012; Oster et al., 2010; Perrouin et al., 2010] e seleciona quatro configurações, como

demonstrado na Figura 2.4. Considere as macros `M1` e `M2`, note que há uma configuração em que ambas estão desabilitadas (config-1), duas outras em que apenas uma está habilitada (config-2 e config-3), e uma outra configuração onde ambas as macros estão habilitadas (config-4). $t$ pode assumir outros valores inteiros para verificar diferentes combinações de macros, como *three-wise* ()$t = 3$) e *four-wise* ($t = 4$). À medida que $t$ aumenta, os tamanhos dos conjuntos de amostras também aumentam.

O algoritmo *maioria habilitada e desabilitada* verifica duas configurações, independente do número de macros de pré-processamento. Quando não há restrições entre macros de pré-processamento, a abordagem habilita todas as macros (config-1), e, em seguida, ele desabilita todas as elas (config-2). Enquanto que o algoritmo *uma desabilitada*. [Abal et al., 2014] desabilita uma macro de cada vez. Observe novamente a Figura 2.4, config-1 desabilita a macro `M1`, config-2 desabilita `M2` e config-3 desabilita `M3`. Em contraste, *uma habilitada* permite uma macro habilitada de cada vez.

```
1  #ifdef M1
2      //codigo1
3  #endif
4
5  #ifdef M2
6      //codigo2
7  #else
8      //codigo3
9  #endif
10
11 #ifdef M3
12     //codigo4
13 #endif
```

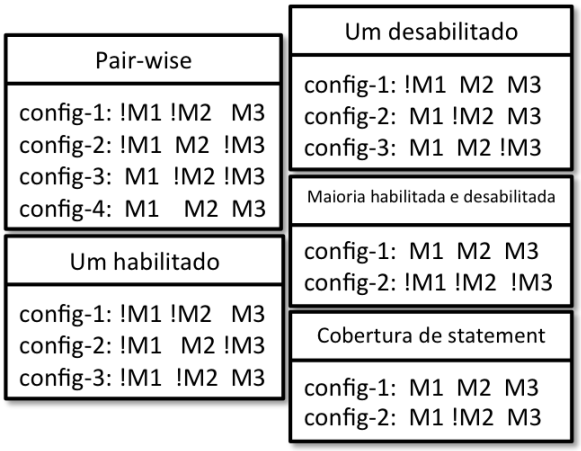


Figura 2.4: Comparação de algoritmos de amostragem através de um exemplo de um sistema configurável com `#ifdefs`.

## 2.6 Análise de Impacto da Mudança

Análise de impacto é a avaliação do efeito de uma mudança [Bohner and Arnold, 1996] no código fonte de um módulo, sobre os outros módulos do sistema [Turver and Munro, 1994]. Análise de impacto pode ser vista como uma técnica aproximada que deve focar na minimização do custo de detectar efeitos secundários indesejados [Queille et al., 1994] ou como a identificação do conjunto de entidades do software que precisam ser modificadas para implementar um novo requisito em um sistema já existente [Lindvall, 1997]. A análise do impacto de uma mudança depende do contexto em que está sendo empregada. Por exemplo, algumas abordagens analisam o código antes da mudança ter sido implementada. Outras avaliam o impacto da mudança após ela ter sido realizada. Além disso, a análise de impacto desejada pode ser à nível de entidades do código ou de módulos do sistema [Mongiovi, 2013]. Existem dois tipos de análise de impacto: estática e dinâmica. As seções a seguir descrevem estas abordagems.

### 2.6.1 Análise Estática

Na análise estática, o sistema configurável não é executado e a análise é feita no código do sistema configurável. Ela pode ser realizada antes da mudança ter sido feita para prever o impacto que a mudança pode causar no código. Para isso, é necessário informar detalhes da mudança proposta. Ela também pode ser feita após a mudança. Neste caso, é necessário avaliar o código antes e depois da mudança. A análise estática explora todos os caminhos de execução possíveis [Jonsson and Blekinge, 2005].

Observe o Código 2.6. Ele mostra um sistema configurável C que contém cinco funções. `funcao1` chama `funcao2` caso o parâmetro `x` recebido for igual a 10, caso contrário, chama `funcao3` . `funcao2` chama `funcao4` e `funcao3` chama `funcao5` . Uma função depende de quem ela chama. O grafo de dependência desse sistema configurável está ilustrado na Figura 2.5. A partir desse grafo, montamos a Tabela 2.1. Uma função modificada vai afetar todas as funções que dependem dela. Por exemplo, `funcao4` pode afetar `funcao1` e `funcao2` , pois essas funções dependem da `funcao4` (exercitam direta ou indiretamente). A tabela pode ser o resultado de uma análise estática realizada nesse sistema configurável. Veja que para qualquer execução desse sistema configurável, as possíveis

funções afetadas por uma função modificada estão na tabela.

```
1  funcao1 (int x) {
2     if (x == 10){
3        funcao2();
4     } else{
5        funcao3();
6     }
7  }
8  funcao2() {
9     funcao4 ( )
10 }
11 funcao3() {
12    funcao5 ( )
13 }
14 funcao4() {}
15 funcao5() {}
```

Código Fonte 2.6: Exemplo de funções de um sistema configurável C.

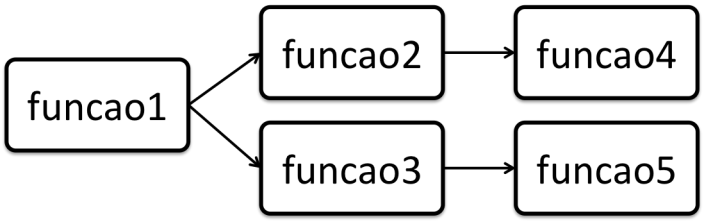


Figura 2.5: Grafo de dependência dos métodos do Código Fonte 2.6

### 2.6.2 Análise Dinâmica

A análise de impacto dinâmica é realizada a partir de execuções de um sistema configurável. Essa análise coleta os rastros das execuções para analisar o impacto. Seu resultado

| Função Modificada | Funções Afetadas |
|---|---|
| funcao1 | - |
| funcao2 | funcao1 |
| funcao3 | funcao1 |
| funcao4 | funcao1, funcao2 |
| funcao5 | funcao1, funcao2 |

Tabela 2.1: Métodos afetados por cada função do Código 2.6 considerando todas as possíveis execuções do sistema configurável.

é preciso, mas depende das entradas fornecidas, pois o resultado pode não ser generalizado para todas as execuções possíveis de um sistema configurável. Não existe garantia que a coleção de testes do sistema configurável, que está sendo executada, é característica de todas as possíveis execuções. Por isso, a análise de impacto dinâmica não promete identificar todos os possíveis impactos [Mongiovi, 2013].

Observe novamente o Código 2.6. A Tabela 2.1 foi construída a partir de uma análise estática no código. Note que ela não depende de valores de entrada, pois a análise estática generaliza os resultados para qualquer execução. Já na análise dinâmica, o sistema configurável precisa de um `main` para executar. Suponha que esta função foi adicionada (Código 2.7). Se a `funcao5` for modificada, a análise de impacto dinâmica consegue identificar as funções afetadas: `funcao1` e `funcao3`, pois as chamadas das linhas 4 a 6, saem do `main`, passam por `funcao3` e chegam na `funcao5`. No entanto, se a `funcao4` for modificada, não será possível identificar as funções afetadas por ela: `funcao1` e `funcao2`, pois não existe alguma execução partindo do `main` que chame `funcao4`. Neste último exemplo, a análise não foi bem sucedida porque o sistema configurável não possui casos de execução que cubram todos os possíveis caminhos do sistema configurável. Na prática, principalmente em programas grandes, é difícil obter execuções de todas as funções possíveis.

```
1  funcao1 (int x) {
2     if (x == 10){
3        funcao2();
4     } else{
5        funcao3();
6     }
7  }
8  funcao2() {
9     funcao4 ( )
10 }
11 funcao3() {
12    funcao5 ( )
13 }
14 funcao4() {}
15 funcao5() {}
16 int main() {
17    funcao1(20);
18 }
```

Código Fonte 2.7: Exemplo de funções de um sistema configurável C.

### 2.6.3 Estratégias de Análise de Impacto

Lehnert [Lehnert, 2011] realizou um estudo sobre as estratégias de análise de impacto existentes e fez um quadro comparativo entre várias abordagens. O estudo revelou que faltam validações empíricas das abordagens propostas e que também faltam abordagens que abrangem todos os estágios do processo de desenvolvimento do software. Por fim, concluiu que não existe uma classificação uniforme entre as abordagens, os tipos de mudanças e as dependências. Em programas orientados a objetos, características como o polimorfismo, a herança e o encapsulamento contribuem diretamente para tornar a tarefa da análise de impacto mais complexa [Li and Offutt, 1996]. Além disso, uma transformação local pode impactar diversas partes do sistema configurável, devido a dependências e inter-relacionamentos entre as classes [Harrold and Rothermel, 1994; Horwitz et al., 1990; Korel and Laski, 1990].

Em linguagens configuráveis, mudanças podem ter o mesmo efeito devido a dependências entre configurações. Uma vez que uma alteração de código que abrange uma opção de

configuração pode impactar diversas configurações com quem possui dependências. Considere a transformação demonstrada na Figura 2.6. Note que no Código 2.8, quando a macro `M1` está habilitada, `funcao1` retorna 1. Este valor foi modificado e, no Código 2.9, `funcao1` passa a retornar 2 quando a macro `M1` está habilitada. Todas as configurações deste sistema que tiverem a macro `M1` habilitada serão impactadas pela transformação.

```
1 int funcao1() {
2 #ifdef M1
3     return 1;
4 #else
5     return 0;
6 #endif
7 }
```

Código Fonte 2.8: Sistema configurável original.

```
1 int funcao1() {
2 #ifdef M1
3     return 2;
4 #else
5     return 0;
6 #endif
7 }
```

Código Fonte 2.9: Sistema configurável modificado.

Figura 2.6: Exemplo uma transformação realizada em um sistema configurável com `#ifdefs`.

## 2.7 Teste de Mutação

Teste de Mutação (ou análise de mutação ou mutação de sistema configurável) é usado para projetar novos testes de *software* e avaliar a qualidade dos testes já existentes. Teste de Mutação envolve modificar um sistema configurável de pequenas formas [DeMillo et al., 1978], Cada versão do sistema configurável modificado é chamada mutante, cujo comportamento é diferente do sistema configurável original. Testes podem detectar e rejeitar mutantes. Isto é chamado de matar ou capturar o mutante. Mutantes equivalentes são aqueles capazes de produzir resultados semelhantes ao sistema configurável original e, assim, não são mortos [DeMillo et al., 1978].

Teste de Mutação é uma técnica baseada na detecção de faltas, que fornece um critério de teste chamado de escore de mutação [Jia and Harman, 2011]. O escore de mutação podem ser utilizado para medir a eficácia de um conjunto de teste em termos da sua capacidade para detectar falhas. Escore de Mutação é a razão entre o número de mutantes mortos e o total de

mutantes. O objetivo é ter um escore igual a um ou 100%, o que significa que todas as faltas de todos os mutantes foram detectados. Quanto mais mutantes forem mortos, maior será o escore. O cálculo do escore de mutação é definido por:

$$\text{EscoreDeMutacao} = \frac{\#\text{mutantesMortos}}{\#\text{mutantes}}$$

O resultado de um teste de mutação pode ser usado como referência para novos casos de teste ou para a eficácia do conjunto de testes. Caso o escore se aproxime de 100%, então há um indicador de que as faltas em uma alta porcentagem de mutantes foram detectadas [Offutt, 1992]. Mutantes equivalentes produzem o mesmo comportamento que o sistema configurável original e, assim, não são mortos [Alexander et al., 2002; Hierons et al., 1999]. Por este motivo, eles não estão incluídos na fórmula que calcula o escore de mutação. Os mutantes vivos ou equivalentes são indicadores de que os casos de teste não são adequadas e, portanto, precisam de melhorias [Smith and Williams, 2009]. Provavelmente, o teste não analisou a parte do código onde a falta ocorre. Depois de melhorar os casos de teste, o testador pode repetir o procedimento até que um Escore de Mutação satisfatório seja alcançado.

Mutantes são baseados em operadores de mutação [DeMillo et al., 1978] bem definidos que tanto imitam erros de programação típicos (tais como o uso do operador errado ou uso de variável não declarada) ou forçam a criação de testes valiosos (como a divisão de cada expressão de zero) [Frankl et al., 1997]. As seções a seguir exemplificam mutantes derivados da aplicação de operadores.

### 2.7.1 MT1 - Remover ponto-e-vírgula

Considere a transformação demonstrada na Figura 2.7. Ao remover o ponto-e-vírgula da linha 3 do Código 2.10, o mutante (Código 2.11) gerado pela aplicação deste operador irá apresentar um erro de compilação quando `MT1` estiver habilitada.

```
1 int main() {
2 #ifdef MT1
3    int x;
4 #endif MT1
5 }
```

Código Fonte 2.10: MT1 - Trecho de código do sistema configurável original

```
1 int main() {
2 #ifdef MT1
3    int x
4 #endif MT1
5 }
```

Código Fonte 2.11: MT1 - Trecho de código do sistema configurável mutante.

Figura 2.7: Transformação realizada em um sistema configurável com `#ifdefs` após aplicação do operador de mutação MT1.

### 2.7.2 MT2 - Remover declaração

Considere a transformação demonstrada na Figura 2.8. Observe que ao remover a declaração da `funcao1` da linha 2 do Código 2.12, o código mutante (Código 2.13) apresentará um erro de compilação quando `MT2` estiver habilitada.

```
1 void funcao1() {}
2
3 void funcao2() {
4 #ifdef MT2
5    funcao1();
6 #endif
7 }
```

Código Fonte 2.12: MT2 - Trecho de código do sistema configurável original

```
1 void funcao2() {
2 #ifdef MT2
3    funcao1();
4 #endif
5 }
```

Código Fonte 2.13: MT2 - Trecho de código do sistema configurável mutante.

Figura 2.8: Transformação realizada em um sistema configurável com `#ifdefs` após aplicação do operador de mutação MT2.

### 2.7.3 MT3 - Alteração de um operador de ponteiro por outro

Considere a transformação demonstrada na Figura 2.9. Caso o operador de ponteiro da linha 4 do Código 2.14 seja substituído por outro operador de ponteiro, o código mutante

(Código 2.15) apresentará um erro de compilação, pois, neste escopo, `x` não é do tipo ponteiro.

```
1 #ifdef MT3
2    void funcao1() {
3       int  x;
4       int *p = &x;
5    }
6 #endif MT3
```

Código Fonte 2.14: MT3 - Trecho de código do sistema configurável original

```
1 #ifdef MT3
2    void funcao1() {
3       int  x;
4       int *p = *x;
5    }
6 #endif MT3
```

Código Fonte 2.15: MT3 - Trecho de código do sistema configurável mutante.

Figura 2.9: Transformação realizada em um sistema configurável com `#ifdefs` após aplicação do operador de mutação MT3.

### 2.7.4 MT4 - Duplicar declaração

Considere a transformação demonstrada na Figura 2.10. Observe que o Código 2.16 possui uma declaração da variável `x` na linha 3. Caso esta declaração seja duplicada, o código mutante (Código 2.17) apresentará um erro de compilação na linha 6 quando a macro `MT4` estiver habilitada.

```
1 int main() {
2 #ifdef MT4
3    int x;
4 #endif MT4
5 }
```

Código Fonte 2.16: MT4 - Trecho de código do sistema configurável original

```
1 int main() {
2    #ifdef MT4
3       int x;
4    #endif MT4
5
6    int x;
7 }
```

Código Fonte 2.17: MT4 - Trecho de código do sistema configurável mutante.

Figura 2.10: Transformação realizada em um sistema configurável com `#ifdefs` após aplicação do operador de mutação MT4.

### 2.7.5 MT5 - Alterar tipo de uma declaração

Considere a transformação demonstrada na Figura 2.11. Observe o Código 2.18, a variável `y`, declarada na linha 3, é do tipo `int` e recebe como valor a variável `x`, também do tipo `int`. Caso o tipo de `y` seja modificado de `int` para `array`, o código mutante 2.19 apresentará um erro de compilação quando a macro `MT5` estiver habilitada.

```
1 #ifdef MT5
2     int x;
3     int y = x;
4 #endif MT5
```

Código Fonte 2.18: MT5 - Trecho de código do sistema configurável original

```
1 #ifdef MT5
2     int x;
3     int y [1] = x;
4 #endif MT5
```

Código Fonte 2.19: MT5 - Trecho de código do sistema configurável mutante.

Figura 2.11: Transformação realizada em um sistema configurável com `#ifdefs` após aplicação do operador de mutação MT5.

### 2.7.6 MT6 - Remover parte de um *statement*

Considere a transformação demonstrada na Figura 2.12. Observe o *statement* na linha 2 do Código 2.20. Ao remover uma parte desse *statement*, o código mutante apresentará um erro de compilação quando a macro `MT6` estiver habilitada, pois o compilador esperava uma expressão e não `;` após o operador `+`.

```
1 #ifdef MT6
2     int x = 1 + 1;
3 #endif MT6
```

Código Fonte 2.20: MT6 - Trecho de código do sistema configurável original

```
1 #ifdef MT6
2     int x = 1 + ;
3 #endif MT6
```

Código Fonte 2.21: MT6 - Trecho de código do sistema configurável mutante.

Figura 2.12: Transformação realizada em um sistema configurável com `#ifdefs` após aplicação do operador de mutação MT6.

### 2.7.7 MT7 - Alterar tipo de retorno

Considere a transformação demonstrada na Figura 2.13. Note que o tipo de retorno de `funcao1` é `int` na linha 2 do Código 2.22. Porém, ao alterar este tipo de retorno para `void`, ocorrerá um erro de compilação ao habilitar a macro `MT7` do código mutante (Código 2.23)

```
1 #ifdef MT7
2     int funcao1() {
3         return 1;
4     }
5     void funcao2() {
6         int x = funcao1();
7     }
8 #endif MT7
```

Código Fonte 2.22: MT7 - Trecho de código do sistema configurável original

```
1 #ifdef MT7
2     void funcao1() {
3         return 1;
4     }
5     void funcao2() {
6         int x = funcao1();
7     }
8 #endif MT7
```

Código Fonte 2.23: MT7 - Trecho de código do sistema configurável mutante.

Figura 2.13: Transformação realizada em um sistema configurável com `#ifdefs` após aplicação do operador de mutação MT7.

# Capítulo 3

# Uma Técnica para Compilar Sistemas Configuráveis com `#ifdefs` Baseada no Impacto da Mudança

A compilação dos produtos de um sistema configurável é uma atividade que pode revelar erros em determinadas configurações, enquanto outras podem não apresentam problemas. Este capítulo descreve a técnica proposta para avaliar se uma mudança no código introduziu novos erros de compilação em um sistema configurável com `#ifdefs` através da análise das configurações impactados pela transformação. A Seção 3.1 descreve em detalhes cada passo da técnica. Em seguida, a Seção 3.2 descreve CHECKCONFIGMX, a ferramenta que implementa a técnica proposta.

## 3.1 Técnica

Nesta seção, a visão geral de nossa técnica é descrita. Atualmente, ela está implementada para C. Apesar disso, a técnica pode ser similarmente aplicada para outras linguagens configuráveis por diretivas de pré-processamento. A técnica proposta contém quatro passos principais (Figura 3.1). Ela recebe dois programas C (original e modificado). Depois realiza uma análise de impacto da mudança através de comparação textual (*diff*) entre o arquivo original e o modificado, identificando as macros impactadas pelas mudanças (Passo 1). A seguir, Todas as configurações possíveis são geradas considerando as macros impactadas, e

os dois programas são compilados para cada uma delas usando GCC (Passo 2). Os erros de compilação encontrados para as duas versões do programa são comparadas e apenas os novos erros que foram introduzidas pela transformação são filtrados (Passo 3). Adiante, a técnica categoriza os erros de compilação de acordo com o elemento que o causou e com o seu tipo, agrupando os erros que ocorrem em mais de uma configuração (Passo 4). Por fim, os erros de compilação e as configurações relacionadas são reportadas ao desenvolvedor.

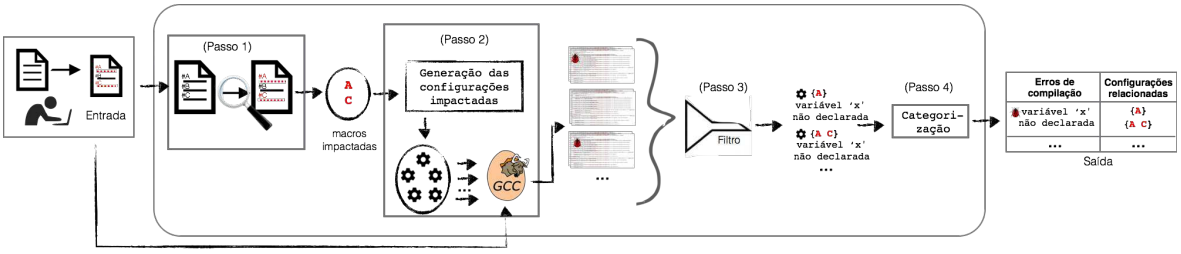


Figura 3.1: Uma técnica para compilar sistemas configuráveis com `#ifdefs` baseada no impacto da mudança.

### 3.1.1 Análise do impacto da mudança

O primeiro passo da técnica consiste em identificar o conjunto de macros impactadas pela mudança. A partir de duas versões de um arquivo, o analisador de impacto realiza um *diff* textual para identificar os trechos de código modificados pela transformação. Em seguida, ele pesquisa por macros nas diferenças textuais identificadas. Uma macro é considerada como diretamente impactada por uma mudança quando as diferenças textuais a contém. Uma macro é indiretamente impactada quando ela habilita a compilação de uma mudança de código.

A Figura 3.2 apresenta dois trechos de código de *commits* consecutivos do arquivo `httpd.c`, do repositório do BusyBox. Os nomes das macros foram renomeados para simplificar a explicação a seguir. O Código 3.1 ilustra parte do código original (commit d2277e2). Onde a função `check_user_passwd` é declarada sob a macro `M1`. O arquivo também contém outras macros, como `M4` e `M5`. Os desenvolvedores alteraram esse arquivo ao adicionar código sob a macro `M1`. O Código 3.2 ilustra parte do arquivo modificado (commit 7291755). A técnica proposta compara os arquivos original e modificado e identifica a diferença textual entre eles (transformação). Neste exemplo, a transformação adiciona o código das linhas 2 a 4 e 7 a 12 do código modificado (Código 3.2). Estes trechos de código

```
1  #ifdef M1
2     static int
          check_user_passwd() { }
3  #endif
4  #ifdef M4 && M5
5     static int miniHttpd(int
          server) { }
6  #endif
```

Código Fonte 3.1: Trecho de código do arquivo `httpd.c` original.

```
1  #ifdef M1
2  #ifdef M2 && M3
3     static int pam_talker() { }
4  #endif
5     static int
          check_user_passwd()
6     {
7  #ifdef M3
8        struct pam_conv
             conv_info =
9            { &pam_talker};
10 #endif
11    }
12 #endif
13 #ifdef M4 && M5
14    static int miniHttpd(int
          server) { }
15 #endif
```

Código Fonte 3.2: Trecho de código do arquivo `httpd.c` modificado.

Figura 3.2: Trechos de código de *commits* consecutivos do arquivo httpd.c. Esta transformação introduz um erro de compilação de uso de variável não declarada.

são considerados como impactados pela mudança. O conjunto de macros diretamente impactadas por este exemplo consiste apenas das macros `M2` e `M3`, uma vez elas fazem parta da mudança de código (diferença textual). Porém, elas estão aninhadas com a macro `M1`. Portanto, `M1` é indiretamente impactada pela mudança pois habilita ou desabilita o código impactado. O conjunto completo de macros impactadas é: `{M1, M2, M3}`.

### 3.1.2 Seleção das configurações impactadas

Neste passo, a técnica proposta produz todas as possíveis configurações impactadas a partir de um conjunto de macros impactadas por uma transformação identificada pelo passo anterior. Apenas as configurações impactadas são compiladas pois possivelmente apenas elas podem ter novos erros de compilação. Como resultado, é possível poupar tempo e esforço ao evitar a compilação de configurações não impactadas.

Inicialmente, a técnica utiliza um algoritmo combinatorial para encontrar todas as combinações de macros impactadas. Por exemplo, considere o exemplo anterior (Figure 3.2). O arquivo modificado contém as macros `M1`, `M2`, `M3`, `M4`, e `M5`. O algoritmo recebe como entrada as macros impactadas (`M1`, `M2` e `M3`) e retorna o conjunto de configurações que as habilitam. O algoritmo retorna o seguinte conjunto de configurações para o exemplo anterior: `{M1}, {M1 M2}, {M1 M3}, {M2}, {M2, M3}, {M3}, {M1, M2, M3}, {}`. Para simplificar, as configurações são representadas pelas macros habilitadas. Portanto, a configuração `{M1}` é igual a `{M1, !M2, !M3, !M4, !M5}`.

Em seguida, para cada configuração encontrada pelo algoritmo combinatorial, é feita uma cópia dele e todas as macros não impactadas são habilitadas. Por exemplo, para a configuração `{M1}`, é criada uma nova configuração `{M1, M4, M5}` que é adicionada ao conjunto de configurações impactadas. Por tanto, a técnica identifica o seguinte conjunto de configurações impactadas para o exemplo anterior: `{M1}, {M1, M2}, {M1, M3}, {M2}, {M2, M3}, {M3}, {M1, M2, M3}, {}, {M1, M4, M5}, {M1, M2, M4, M5}, {M1, M3, M4, M5}, {M2, M4, M5}, {M2, M3, M4, M5}, {M3, M4, M5}, {M1, M2, M3, M4, M5}, {M4, M5}`.

O número de configurações impactadas é diretamente proporcional ao número de macros impactadas. A técnica proposta analisa $O(2^{i+1})$ configurações, onde i é o número de macros impactadas. A técnica pode ter uma melhor performance quando analisando transformações

de granularidade fina aplicadas a sistemas altamente configuráveis com `#ifdefs`, uma vez que estas transformações tendem a ter poucas macros impactadas dentre um grande conjunto de macros.

Em seguida, ambas as versões do arquivo são compiladas para cada configuração impactada. Uma vez que as configurações impactadas são identificadas com base nas macros impactadas na versão modificada do arquivo, a versão original pode não conter as mesmas macros. Desta forma, para compilar as configurações no código original, as macros que existem apenas no código modificado são desabilitadas.

A Figura 3.2 apresenta parte do arquivo `httpd.c` com cinco macros. Porém, o *commit* real do arquivo contém 18 marcos. A técnica proposta gera apenas 16 configurações (99% de redução) para compilar cada versão do arquivo ao basear-se no impacto da mudança. Para este exemplo, um erro de compilação é encontrado quando `M1` e `M3` são habilitadas e `M2` é desabilitada.

### 3.1.3 Filtragem e categorização dos erros de compilação

No Passo 3 (filtro), a técnica seleciona automaticamente os erros de compilação que ocorrem apenas na versão modificada do arquivo. As mensagens são filtradas de acordo com seus modelos. Um modelo de uma mensagem de erro contém: o tipo do erro, que inclui o elemento do código que o causou; e, a linha do erro (número e conteúdo). Uma transformação pode adicionar ou remover algumas linhas do código antes do erro, em relação a linha do código. Portanto, as mensagens são filtradas através da remoção da localização do erro (linha e coluna), uma vez que o mesmo erro de compilação pode ocorrer em ambas as versões do arquivo de um sistema (original e modificado), porém em linhas diferentes. O objetivo da técnica consiste em identificar apenas os novos erros de compilação introduzidos pela transformação. Os erros pré-existentes não são identificados.

O último passo categoriza as mensagens de erro de compilação filtradas em mensagens distintas ao analisar se eles estão relacionados à mesma falta (Passo 4). Duas mensagens de erro são consideradas como relacionadas à mesma falta se elas contém o mesmo tipo de erro e o mesmo elemento. As mensagens de erro são analisadas e categorizadas com base nos seus modelos, ignorando a localização do erro (linha e coluna) e o conteúdo da linha do erro. Por exemplo, o GCC reporta a seguinte mensagem de erro quando tenta

compilar o programa apresentado no Código 1.2: `httpd.c:11:17: erro: “uso de variável não declarada ‘pam_talker’ {&pam_talker};`.” A técnica considera que a seguinte mensagem de erro é relacionada ao mesmo erro de compilação da anterior: `httpd.c:22:1: erro: “uso de variável não declarada ‘pam_talker’ pam_talker = 0;`.” As diferenças entre elas são a linha que o erro ocorreu e o conteúdo do erro. Por fim, se o mesmo erro de compilação ocorre em mais de uma configuração, ele é classificado como apenas um erro. Então, o categorizador reporta ao usuário o conjunto de erros de compilação e as configurações relacionadas identificados pela técnica.

## 3.2 CHECKCONFIGMX

A CHECKCONFIGMX é uma ferramenta desenvolvida que implementa a técnica proposta descrita anteriormente. A ferramenta analisa duas versões de um sistema configurável com `#ifdefs` implementado em C, e identifica as configurações que podem apresentar erros de compilação devido à mudanças de código entre as versões. A técnica da ferramenta implementa os quatro passos da técnica: identificar as diferenças textuais entre o arquivo original e o modificado usando o comando *diff* do Git; em seguida, identificar todas as macros impactadas pela transformação; gerar todas as possíveis configurações impactadas; e, para cada uma, compilar os dois programas com a configuração impactada utilizando GCC. Por fim, a ferramenta filtra e categoriza os novos erros de compilação introduzidos pela transformação e as configurações relacionadas e os reporta ao usuário. CHECKCONFIGMX utiliza a ferramenta GCC para realizar compilar e analisar os arquivos de entrada (original e modificado). O GCC considera inúmeros erros de compilação.

### 3.2.1 Git *diff*

CHECKCONFIGMX utiliza o comando *diff* do Git para realizar a análise de impacto da mudança. Git é um sistema de controle de versão distribuído e um sistema de gerenciamento de código fonte. O Cada diretório de trabalho do Git é um repositório com um histórico completo e habilidade total de acompanhamento das revisões, não dependente de acesso a uma rede ou a um servidor central.

O comando *diff* apresenta as mudanças textuais entre códigos de dois *commits* de um repositório. A Figura 3.3 apresenta uma captura de tela da execução do comando para identificar as diferenças textuais entre duas versões do arquivo `shell/ash.c` do BusyBox. CHECKCONFIGMX analisa toda a saída da ferramenta e identifica o código impactado pela mudança.

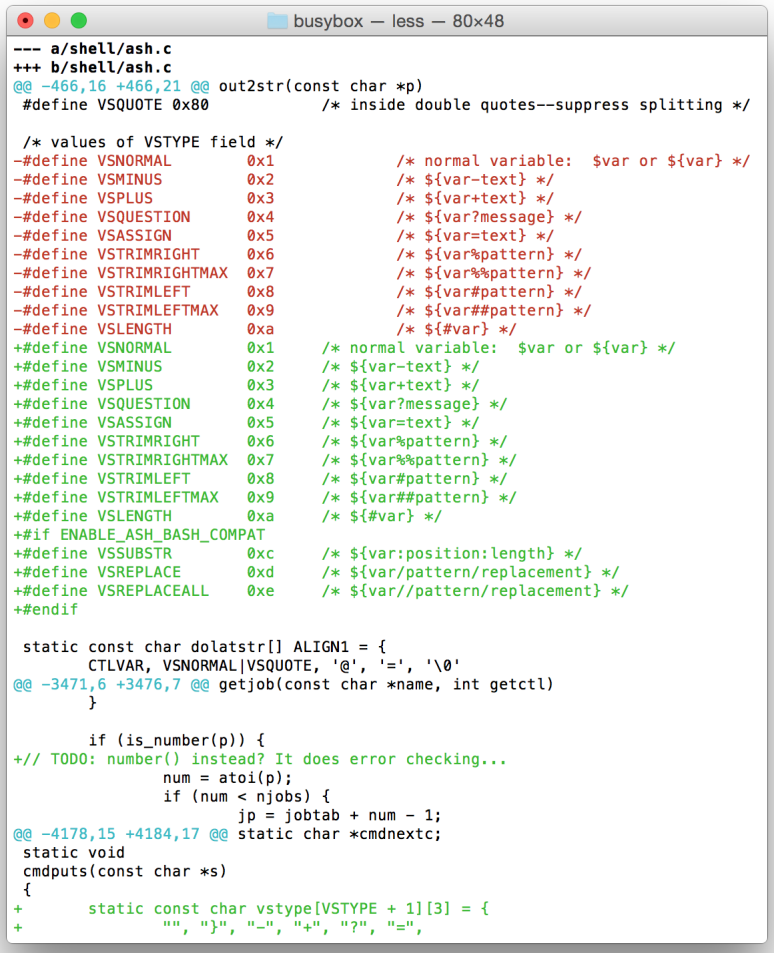


Figura 3.3: Captura de tela do comando *diff* do Git em versões do arquivo shell/ash.c do BusyBox.

### 3.2.2 GCC

CHECKCONFIGMX executa a ferramenta GCC para pré-processar e compilar os sistemas configuráveis. GCC é um conjunto de compiladores de linguagens de programação,

como C e C++. É considerada uma das ferramentas essênciais para manter o *software* independente de plataforma, pois permite compilar o código-fonte em binários executáveis para várias plataformas, como Mac OSX. GCC possui quatro passos: pré-processar; compilar; *assembler* e *linker*. CHECKCONFIGMX utiliza apenas os dois primeiros passos (Figura 3.4).

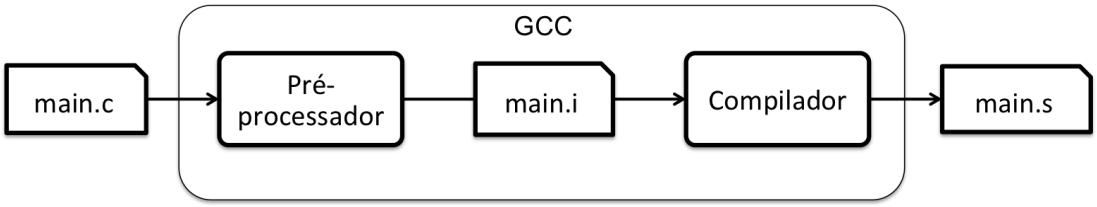


Figura 3.4: Passos do GCC utilizados pela ferramenta CHECKCONFIGMX.

Cada arquivo C recebido como entrada passa pelo pré-processador. Nesta passo, todos os arquivos de inclusão são expandidos, por exemplo as bibliotecas, e macros são expandidas, de acordo com a configuração determinada, para serem pré-processados. Em seguida, cada arquivo é compilado em um código *assembler*. Os erros de compilação, como o uso de uma variável não declarada, são detectados neste passo e apresentados ao usuário.

# Capítulo 4

# Avaliação

Este capítulo apresenta um estudo que avalia a técnica proposta em relação à detecção de erros de compilação baseada em análise de impacto da mudança. Primeiro, a Seção 4.1 apresenta a definição do experimento e em seguida a Seção 4.2 apresenta seu planejamento. A Seção 4.4 discute os principais resultados. Em seguida, a Seção 4.5 descreve as ameaças à validade do experimento. Por fim, a Seção 4.6 responde as perguntas de pesquisa.

## 4.1 Definição

O experimento foi definido seguindo a abordagem Objetivo, Questão, Métrica (GQM) [Basili et al., 1994] para coletar e analisar métricas significativas para medir o processo proposto. O objetivo deste experimento é analisar CHECKCONFIGMX com o intuito de avaliar mudanças de código com respeito a encontrar erros de compilação do ponto de vista do pesquisador no contexto de sistemas altamente configuráveis com `#ifdefs`. Em particular, o experimento aborda as seguintes questões de pesquisa:

- $\mathrm{RQ}_1$**:** Quais os tipos de erro de compilação encontrados por CHECKCONFIGMX?
  Os tipos dos erros de compilação detectados por CHECKCONFIGMX no estudo realizado são identificados.

- $\mathrm{RQ}_2$**:** Quanto esforço CHECKCONFIGMX reduz para encontrar erros de compilação em termos de configurações analisadas?
  Para cada transformação analisada, o total de configurações impactadas que CHECK-

CONFIGMX identifica é comparado com o total de possíveis configurações, sem considerar *feature models*.

- $\mathrm{RQ}_3$**:** Qual a taxa de transformações que introduzem ao menos um erro de compilação?
  A taxa de pares de *commits* (original, modificado) em que CHECKCONFIGMX detecta ao menos um erro de compilação introduzido pela transformação é medida.

- $\mathrm{RQ}_4$**:** Qual a taxa de erros de compilação que ocorrem em transformações que não impactaram macros?
  O total de macros impactadas e o total de erros de compilação introduzidos pelas transformações são medidos.

- $\mathrm{RQ}_5$**:** Qual a taxa de falsos positivos encontrados por CHECKCONFIGMX?
  Cada erro de compilação identificado pela ferramenta é analisado manualmente para avaliar se é real ou falso positivo.

## 4.2 Planejamento

Esta seção descreve os sistemas e a instrumentação utilizados no experimento.

### 4.2.1 Seleção dos Sistemas

Uma transformação é composta por um par de *commits* consecutivos de um repositório. Este estudo analisa as transformações do histórico dos repositórios Git dos seis maiores arquivos do BusyBox, cinco maiores arquivos do Apache HTTPD, e três maiores arquivos do Expat (com base no tamanho do último *commit*). Considera-se que os maiores arquivos podem ter os cenários mais interessantes para serem analisados e podem melhor avaliar a técnica proposta.

CHECKCONFIGMX pode avaliar transformações com qualquer número de macros impactadas. Porém, este estudo não foca em pares com mais do que quatro macros impactadas, uma vez que avaliá-los pode ser custoso. Apesar desta restrição, o estudo avalia 96.01% das transformações.

O estudo analisa 1.837 pares de *commits* (transformações) realizadas entre Abril de 2001 a Novembro de 2015 do histórico dos arquivos selecionados do repositório Git do BusyBox.

| SC | Arquivo | Pares | Des. | Média | |
|---|---|---|---|---|---|
| | | | | Macros | Diff |
| BusyBox | ash | 543 | 32 | 49,24 | 103,94 |
| | httpd | 247 | 15 | 15,91 | 44,35 |
| | hush | 696 | 18 | 29,19 | 49,35 |
| | modutils-24 | 24 | 4 | 24,75 | 21,50 |
| | ntpd | 107 | 18 | 7 | 48,64 |
| | vi | 223 | 22 | 17 | 47 |
| Apache HTTPD | core | 577 | 52 | 15 | 17 |
| | event | 147 | 19 | 26,27 | 24,89 |
| | mod_include | 353 | 39 | 5,62 | 40,53 |
| | mod_rewrite | 427 | 51 | 31,20 | 17 |
| | proxy_util | 524 | 33 | 3,68 | 28,18 |
| Expat | xmlparse | 166 | 6 | 9,81 | 52,88 |
| | xmltok | 36 | 5 | 4,22 | 19,89 |
| | xmltok_impl | 14 | 3 | 4,21 | 47,43 |
| | **Total** | **4.084** | **317** | **17,36** | **40,18** |

Tabela 4.1: Sistemas configuráveis com `#ifdefs` analisados durante o estudo realizado para avaliar a técnica proposta. SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Pares = Pares analisados; Des. = Desenvolvedores que realizaram os *commits*; Macros = Média de macros do histórico do arquivo analisado; Diff = Média de linhas da diferença textual entre o par analisado.

Os arquivos analisados possuem uma média de 29,94 macros diferentes por versão de arquivo. A maior quantidade de macros em um arquivo desse sistema é de 58, enquanto que a menor é de 1. São analisados 2.023 pares entre o primeiro *commit*, realizado em Agosto de 1999, e o último, realizado em Junho de 2016, do histórico dos arquivos selecionados repositório Git do Apache HTTPD. Os arquivos analisados possuem uma média de 11 macros diferentes por versão de arquivo. A maior quantidade de macros em um arquivo desse sistema é de 28, enquanto que a menor é de 1. São analisados 216 pares entre o primeiro *commit*, realizado em Setembro de 2000, e o último, realizado em Junho de Fevereiro de 2010, do histórico dos arquivos selecionados do repositório Git do Expat. Os arquivos analisados possuem uma média de 8,49 macros diferentes por versão de arquivo. A maior quantidade de macros em um arquivo desse sistema é de 12, enquanto que a menor é de 3. A Tabela 4.1 detalha os sistemas configuráveis com `#ifdefs` que são avaliados neste estudo.

### 4.2.2 Instrumentação

Os experimentos foram executados em um Mac OSX Yosemite com 2.6 GHz, core i5 e 8 GB de RAM. Em adição, foram utilizados: GCC 4.2.1 (Apple LLVM versão 6.0), recebendo como parâmetros o arquivo analisado e uma configuração, e considerando `lib` como a pasta padrão de bibliotecas; Java 1.8.0-45; GNU Bash 4.3.33 (x86 64-apple-darwin13.4.0); e, Git

2.5.4 (Apple Git-61).

### 4.2.3 Caracterização dos Sistemas

BusyBox é um *software* código aberto que oferece várias ferramentas Unix compactadas em um único arquivo executável. Ele é executado em uma variedade de ambientes como o Linux, Android e FreeBSD, no entanto muitas das ferramentas que ele oferece são projetados para trabalhar com interfaces fornecidas pelo núcleo do Linux. Foi criado especialmente para sistemas operacionais embarcados com recursos muito limitados. O sistema substitui funções básicas de mais de 300 comandos comuns, como `killall`. O código fonte do respositório Git do BusyBox[1] está ativo há quase 17 anos e conta com mais de 14.000 *commits*, caracterizando um alto nível de atividade. No último `commit` avaliado, o sistema continha 234.826 linhas de código e 651 macros. Ou seja, este sistema pode gerar um possível total de configurações de $2^{651}$, sem considerar *feature models*.

O projeto Apache HTTP Server, conhecido como Apache HTTPD, é um esforço para desenvolver e manter um servidor HTTP de código aberto para os sistemas operativos modernos, incluindo UNIX e Windows. O objetivo do projeto é prover um servidor extensível, seguro e eficiente que fornece serviços HTTP em sincronia com os padrões HTTP atuais. Em maio de 2010,[2] o Apache serviu aproximadamente 54,68% de todos os sites e mais de 66% dos milhões de sites mais movimentados. O código fonte do repositório Git do Apache[3] está ativo há quase 20 anos e contém mais de 28.800 *commits*, caracterizando um alto nível de atividade.

Expat é um biblioteca de analisadores de XML orientados à *stream*, escrita em C. Como um dos primeiros analisadores XML de código aberto disponíveis, Expat encontrou um lugar em muitos projetos de código aberto, como Apache HTTPD e Mozilla. O código fonte do repositório do Expat[4] está ativo a quase 19 anos e contém mais de 1.300 *commits*.

[1] https://git.busybox.net/busybox/

[2] http://news.netcraft.com/archives/2010/05/14/may_2010_web_server_survey.html

[3] https://github.com/apache/httpd

[4] https://sourceforge.net/p/expat/code_git/ci/master/tree/

## 4.3 Resultados

CHECKCONFIGMX avaliou um total de 3,913 transformações aplicadas aos sistemas configuráveis BusyBox, Apache HTTPD e Expat. Os arquivos modificados das transformações somam um total de 77,364 macros, cuja a técnica proposta identificou 3,508 delas como sendo impactadas. Considerando a abordagem exaustiva de compilar todas as configurações possíveis, sem considerar *feature models*, CHECKCONFIGMX reduziu o esforço da análise, em termos de configurações analisadas, em pelo menos 50% (uma média de 99%). CHECKCONFIGMX identificou 776 configurações (6.02% do total de configurações impactadas) com pelo menos um erro de compilação. A ferramenta encontrou 5,639 mensagens de erros de compilação, que foram categorizados em 942 erros diferentes. Porém, 36.83% deles são falsos positivos. Por exemplo, foram detectados falsos positivos relacionados à bibliotecas não disponíveis em *commits* antigos. Então, a técnica proposta encontrou 595 erros de compilação reais.

Os 595 erros foram classificados em 20 tipos de erros de compilação. Os desenvolvedores dos sistemas configuráveis analisados introduziram estes erros de compilação em um total de 214 transformações. A Tabela 4.2 mostra o total de pares que introduziram erros de compilação e o total de desenvolvedores que realizaram os *commits* dessas transformações. O tipo de erro com maior ocorrência no estudo foi o *uso de variável não declarada* (49,7%), seguido por *não há membro na* `struct` (31,7%) e *definição incompleta de tipo* (3,1%). As tabelas Tabela 4.4 e 4.3 resumem os erros de compilação encontrados por CHECKCONFIGMX durante o estudo.

As transformações que introduziram pelo menos um erro de compilação contém uma média 12,92 LOC de diferença textual. A maior transformação, em termos de linhas de código modificadas, ocorreu em um par de *commits* do Apache HTTPD (*commits* f5858d9 e e56d601). Esta transformação possui uma diferença textual de 1.735 LOC e introduziu cinco erros de compilação em quatro configurações. CHECKCONFIGMX identificou 20 transformações que introduziram erros de compilação e possuíam apenas uma LOC de diferença textual. Os desenvolvedores introduziram em média 2,4 erros de compilação por transformação analisada no estudo. CHECKCONFIGMX também identificou quer 132 transformações não possuem macros impactadas mas introduziram 307 erros de compilação no total.

| SC | Arquivo | Erros | |
|---|---|---|---|
| | | Pares | Des. |
| BusyBox | ash | 27 | 11 |
| | httpd | 18 | |
| | hush | 16 | |
| | modutils-24 | 8 | |
| | ntpd | 6 | |
| | vi | 14 | |
| Apache HTTPD | core | 79 | 29 |
| | event | 6 | |
| | mod_include | 26 | |
| | mod_rewrite | 3 | |
| | proxy_util | 8 | |
| Expat | xmlparse | 3 | 1 |
| | **Total** | **214** | **41** |

Tabela 4.2: Erros de compilação detectados por CHECKCONFIGMX; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Pares = Pares de *commits* que introduziram erros de compilação; Des. = Desenvolvedores que realizaram os *commits* que introduziram erros de compilação.

Por exemplo, uma transformação aplicada no arquivo `proxy_util.c` do Apache HTTPD (*commits* 935de30 e e63bcb2), não impactou macros, mas introduziu 31 erros de compilação diferentes. As Tabelas 4.5 a 4.16 detalham o total de erros encontrados nas transformações analisadas.

## 4.4 Discussão

Essa seção discute questões relacionadas: aos erros de compilação encontrados; aos passos de análise de impacto, filtro, e categorizador; aos falsos positivos, aos falsos negativos; e, ao tempo de avaliação das transformações.

### 4.4.1 Erros de compilação

CHECKCONFIGMX identificou que a maior parte dos erros de compilação que os desenvolvedores introduziram nas transformações analisadas foram causados por *uso de variável não declarada* (49,7%), seguido por *não há membro na* `struct` (31,7%) e *definição incompleta de tipo* (3,1%). O primeiro e o terceiro erro que mais foram introduzidos são relacionados à `struct`. Este resultado pode indicar que os desenvolvedores podem ter dificuldade em lidar com este tipo de estrutura enquanto usam diretivas `#ifdef`. A Tabela 4.3 mostra os tipos e o total de erros de compilação encontrados pela técnica proposta neste trabalho.

Uma transformação aplicada ao arquivo `xmlparse.c` do Expat (*commits* 3370a61

| Tipos de Erros de Compilação | BusyBox | Apache HTTPD | Expat | Total |
|---|---|---|---|---|
| uso de vairável não declarada | 171 | 123 | 2 | 296 |
| nenhum membro na *struct* | 24 | 165 | - | 189 |
| membro do tipo base de referência não é uma estrutura ou união | - | 19 | - | 19 |
| definição incompleta de tipo | 16 | 2 | | 18 |
| operandores inválidos para expressão binária | 12 | - | - | 12 |
| muito poucos argumentos para chamada de função | - | 12 | - | 12 |
| nome de tipo desconhecido | - | 11 | - | 11 |
| uso de *label* não declarado | 9 | - | - | 9 |
| expressão não é atribuível | 6 | - | - | 6 |
| tipos conflitantes | - | 4 | - | 4 |
| valor subscrito não é uma matriz, ponteiro, ou vector | 4 | - | - | 4 |
| ponteiros para tipos incompatíveis | 2 | - | 1 | 3 |
| referência à variável local declarada em função *enclosing* | - | 3 | - | 3 |
| endereço do *bit-field* solicitado | - | 2 | - | 2 |
| tipo de objeto chamado não é um ponteiro de função ou função | 2 | - | - | 2 |
| *statement continue* não está em um *statement* de laço | - | 1 | - | 1 |
| função não pode retornar tipo array | - | 1 | - | 1 |
| função não-*void* deve retornar um valor | - | 1 | - | 1 |
| não é um ponteiro de função ou função | - | 1 | - | 1 |
| nome do tipo requer um especificador ou qualificador | - | 1 | - | 1 |
| **Total** | **246** | **346** | **3** | **595** |

Tabela 4.3: Tipos de erros de compilação encontrados por CHECKCONFIGMX durante o estudo.

| SC | Arquivo | Filtro | | Categorização | |
|---|---|---|---|---|---|
| | | Configs. | Total | Total | Real |
| BusyBox | ash | 284 | 1.689 | 145 | 89 |
| | httpd | 51 | 302 | 110 | 63 |
| | hush | 93 | 255 | 36 | 31 |
| | modutils-24 | 33 | 344 | 35 | 33 |
| | ntpd | 14 | 22 | 9 | 5 |
| | vi | 48 | 207 | 41 | 26 |
| Apache HTTPD | core | 165 | 943 | 199 | 181 |
| | event | 7 | 165 | 17 | 17 |
| | mod_include | 85 | 735 | 105 | 85 |
| | mod_rewrite | 12 | 302 | 17 | 12 |
| | proxy_util | 42 | 350 | 64 | 50 |
| Expat | xmlparse | 6 | 13 | 5 | 3 |
| | xmltok | - | - | - | - |
| | xmltok_impl | 12 | 312 | 159 | 0 |
| | **Total** | **852** | **5.639** | **942** | **595** |

Tabela 4.4: Total de erros de compilação encontrados nos sistemas configuráveis pelos Passos 3 e 4 da técnica proposta; SC = Sistema COnfigurável; Arquivo = Nome do arquivo analisado; Filtro/Configs. = Configurações impactadas com erros de compilação; Filtro/Total = Mensagens de erros de compilação introduzidos por transformações; Categorização/Total = Erros de compilação que ocorrem em diferentes elementos e de diferentes tipos; Categorização/Real = Erros de compilação reais após remoção de falsos positivos através da análise manual dos erros detectados.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| BusyBox | ash | 8de331d | 3f11b1b | 12 | 2 | 40 | 8 | 36 | 6 | 3 |
| | | 3f11b1b | 0b62158 | 23 | 1 | 42 | 4 | 20 | 4 | 4 |
| | | 19008b8 | 6ab0378 | 33 | 2 | 43 | 8 | 24 | 3 | 4 |
| | | 6ab0378 | a34b48a | 1 | 1 | 44 | 2 | 2 | 1 | 1 |
| | | 5c3d2b3 | 991a1da | 75 | 1 | 45 | 4 | 26 | 11 | 7 |
| | | 94e87bc | 0e6f661 | 99 | 0 | 46 | 2 | 2 | 1 | 1 |
| | | 59f351c | 92e13c2 | 332 | 4 | 43 | 32 | 720 | 10 | 9 |
| | | f7d5665 | 80591b0 | 46 | 2 | 45 | 6 | 21 | 6 | 4 |
| | | 80591b0 | 468aea2 | 10 | 0 | 47 | 1 | 5 | 3 | 1 |
| | | 468aea2 | 4a9ca13 | 27 | 1 | 46 | 4 | 74 | 9 | 5 |
| | | 5e25ddb | cd2663f | 11 | 3 | 46 | 10 | 18 | 4 | 1 |
| | | 9cd4c76 | b07a496 | 130 | 4 | 45 | 32 | 112 | 4 | 3 |
| | | b07a496 | ef527f5 | 102 | 4 | 45 | 32 | 128 | 3 | 1 |
| | | 3177ba0 | a53de7f | 3 | 1 | 47 | 2 | 2 | 1 | 1 |
| | | 34c73c4 | d6855d1 | 2 | 1 | 47 | 2 | 2 | 1 | 1 |
| | | 834dee7 | b730474 | 6 | 1 | 47 | 2 | 5 | 2 | 2 |
| | | be54d6b | 340299a | 38 | 0 | 48 | 1 | 1 | 1 | 1 |
| | | 4e12b1a | 8ad78e1 | 10 | 0 | 47 | 1 | 2 | 2 | 1 |
| | | 653d8e7 | b21f379 | 11 | 0 | 47 | 2 | 3 | 2 | 2 |
| | | b21f379 | f173395 | 2 | 0 | 47 | 1 | 1 | 1 | 1 |
| | | eb85849 | 5e34ff2 | 24 | 2 | 44 | 8 | 128 | 15 | 4 |
| | | 82a6fb3 | 883cea4 | 9 | 2 | 44 | 4 | 4 | 1 | 1 |
| | | c8334a4 | ecc2a2e | 72 | 3 | 43 | 8 | 24 | 2 | 2 |
| | | 1166d7b | 844f990 | 57 | 1 | 45 | 4 | 16 | 1 | 1 |
| | | e7670ff | 76ace25 | 11 | 1 | 45 | 2 | 2 | 1 | 1 |
| | | da75f44 | 1fcbff2 | 1 | 0 | 32 | 2 | 145 | 40 | 25 |
| | | 1fcbff2 | fd33e17 | 36 | 1 | 31 | 4 | 14 | 3 | 2 |

Tabela 4.5: Principais resultados na análise do arquivo `ash.c` do BusyBox; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| BusyBox | httpd | 0cb6f35 | 4333a09 | 48 | 3 | 14 | 2 | 10 | 1 | 1 |
| | | 4333a09 | 9a51540 | 55 | 0 | 17 | 2 | 65 | 26 | 14 |
| | | 0d8766a | e7f8a32 | 12 | 2 | 15 | 6 | 6 | 1 | 1 |
| | | d086b50 | da0dcd1 | 9 | 1 | 13 | 1 | 1 | 1 | 1 |
| | | 8c69afd | 7dbf1b4 | 13 | 2 | 17 | 8 | 120 | 27 | 12 |
| | | 9230582 | c4523c2 | 7 | 2 | 17 | 3 | 6 | 3 | 3 |
| | | 0eb406c | 25b4630 | 70 | 1 | 18 | 1 | 2 | 2 | 2 |
| | | a53de7f | d73cbd3 | 1 | 1 | 18 | 1 | 2 | 2 | 2 |
| | | 5415c85 | b424930 | 16 | 1 | 18 | 1 | 6 | 2 | 2 |
| | | b424930 | 9d1d4c0 | 6 | 1 | 18 | 1 | 2 | 1 | 1 |
| | | cbb4e61 | 5e34ff2 | 40 | 1 | 15 | 2 | 43 | 22 | 7 |
| | | a4bcbd0 | 108b8c5 | 33 | 2 | 14 | 1 | 4 | 4 | 4 |
| | | a3aa3e3 | dbc6a7a | 7 | 1 | 15 | 1 | 1 | 1 | 1 |
| | | 535ce1d | 7a2ba32 | 58 | 2 | 15 | 4 | 10 | 4 | 2 |
| | | 8030a14 | 9575518 | 1 | 0 | 17 | 1 | 1 | 1 | 1 |
| | | d2277e2 | 7291755 | 158 | 4 | 14 | 6 | 9 | 3 | 3 |
| | | f282c6b | 03419aa | 20 | 2 | 17 | 2 | 6 | 5 | 5 |
| | | 8cce1b3 | 7a42693 | 1 | 1 | 18 | 1 | 1 | 1 | 1 |

Tabela 4.6: Principais resultados na análise do arquivo `httpd.c` do BusyBox; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| BusyBox | hush | eaecbf3 | 4c9b68f | 341 | 0 | 7 | 1 | 1 | 1 | 1 |
| | | 08126f6 | 8317799 | 9 | 2 | 21 | 4 | 4 | 1 | 1 |
| | | 30c9cc5 | 6eaf8de | 16 | 1 | 26 | 2 | 2 | 1 | 1 |
| | | 6eaf8de | 16c2fea | 6 | 1 | 26 | 2 | 4 | 1 | 1 |
| | | 211b59b | cc3f20b | 13 | 1 | 26 | 4 | 4 | 1 | 1 |
| | | ff182a3 | 733e3fb | 44 | 2 | 26 | 4 | 18 | 3 | 3 |
| | | 5415c85 | be709c2 | 34 | 3 | 26 | 6 | 22 | 4 | 4 |
| | | be709c2 | 5e052ca | 147 | 4 | 25 | 32 | 88 | 4 | 2 |
| | | cf22c89 | bcb2553 | 128 | 3 | 26 | 16 | 68 | 6 | 5 |
| | | 45cb9f9 | 4554b72 | 7 | 2 | 27 | 2 | 4 | 2 | 2 |
| | | 4554b72 | ff29b4f | 3 | 1 | 28 | 2 | 3 | 3 | 1 |
| | | c373527 | afd7a8d | 110 | 2 | 27 | 8 | 20 | 3 | 3 |
| | | 39456a1 | ec2c655 | 27 | 1 | 28 | 2 | 5 | 2 | 2 |
| | | 40b8dc4 | 9f8128f | 142 | 1 | 28 | 2 | 2 | 1 | 1 |
| | | a2b11e3 | cd418a2 | 44 | 3 | 31 | 4 | 8 | 2 | 2 |
| | | cd418a2 | 0e15138 | 13 | 2 | 32 | 2 | 2 | 1 | 1 |

Tabela 4.7: Principais resultados na análise do arquivo `hush.c` do BusyBox; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| BusyBox | modutils-24 | ba1315d | 1ad4db1 | 80 | 3 | 21 | 8 | 188 | 12 | 12 |
| | | 36309cf | 3bc3f08 | 30 | 1 | 23 | 2 | 22 | 4 | 4 |
| | | 73c571a | e1de3af | 46 | 4 | 21 | 8 | 100 | 9 | 9 |
| | | e1de3af | f439304 | 41 | 3 | 22 | 4 | 22 | 5 | 3 |
| | | 1f63229 | d5f1b1b | 70 | 4 | 21 | 8 | 8 | 1 | 1 |
| | | 083e172 | 98a4c7c | 2 | 0 | 25 | 1 | 2 | 2 | 2 |
| | | ea8b252 | 58662f2 | 22 | 0 | 25 | 1 | 1 | 1 | 1 |
| | | d48fdde | 8dff01d | 1 | 0 | 25 | 1 | 1 | 1 | 1 |

Tabela 4.8: Principais resultados na análise do arquivo `modutils-24.c` do BusyBox; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| BusyBox | ntpd | ca6c7e4 | 363e89b | 130 | 2 | 1 | 4 | 4 | 1 | 1 |
| | | 2d3253d | 4168fdd | 34 | 0 | 7 | 2 | 3 | 2 | 1 |
| | | 074e8dc | ede737b | 61 | 0 | 7 | 2 | 3 | 2 | 1 |
| | | b124c34 | e8ce285 | 37 | 0 | 8 | 1 | 1 | 1 | 1 |
| | | 5a21c85 | 278842d | 27 | 1 | 7 | 4 | 10 | 2 | 1 |

Tabela 4.9: Principais resultados na análise do arquivo `ntpd.c` do BusyBox; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| BusyBox | vi | 3f98040 | d402edf | 18 | 3 | 11 | 4 | 88 | 8 | 6 |
| | | 61e45db | 6f347ef | 3 | 0 | 16 | 1 | 25 | 1 | 1 |
| | | cd473dd | a68ea1c | 13 | 2 | 14 | 4 | 4 | 1 | 1 |
| | | a68ea1c | dfba741 | 1 | 0 | 16 | 1 | 2 | 1 | 1 |
| | | dfba741 | bc68cd1 | 15 | 0 | 16 | 1 | 1 | 1 | 1 |
| | | bc68cd1 | dbf935d | 8 | 1 | 15 | 1 | 1 | 1 | 1 |
| | | 2d6af16 | e5e1a10 | 16 | 3 | 10 | 16 | 32 | 1 | 1 |
| | | d3c042f | a985d30 | 4 | 0 | 18 | 1 | 7 | 5 | 2 |
| | | 5f6aaf3 | c3a9dc8 | 5 | 1 | 17 | 1 | 1 | 1 | 1 |
| | | c0dab37 | 8131eea | 4 | 1 | 17 | 2 | 12 | 5 | 5 |
| | | 243d175 | c5fb0ad | 15 | 2 | 17 | 4 | 10 | 4 | 2 |
| | | 9e7c002 | 264f373 | 47 | 2 | 16 | 1 | 3 | 3 | 3 |
| | | a8d6f9b | 2c51202 | 57 | 2 | 18 | 1 | 2 | 1 | 1 |

Tabela 4.10: Principais resultados na análise do arquivo `vi.c` do BusyBox; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arq. | Commit | | Diff | Macros | | Filtro | | Categ. | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Mod. | | Imp. | N. Imp. | Configs | Erros | Total | Reais |
| Apache HTTPD | core.c | 39175ff | aafa7af | 13 | 1 | 7 | 4 | 8 | 1 | 1 |
| | | f72b47b | 13b1008 | 4 | 0 | 9 | 2 | 10 | 1 | 1 |
| | | 9d2dcd6 | 8425de7 | 38 | 0 | 15 | 2 | 6 | 1 | 1 |
| | | 3d76ab9 | 1571a1a | 18 | 0 | 15 | 2 | 4 | 1 | 1 |
| | | 665e489 | e3d405c | 4 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | da750ee | b044606 | 34 | 0 | 15 | 2 | 106 | 4 | 1 |
| | | b044606 | f2353e1 | 13 | 0 | 15 | 2 | 91 | 21 | 19 |
| | | 4f4c4c2 | 6635960 | 3 | 0 | 15 | 1 | 1 | 1 | 1 |
| | | bea7dd2 | 27677a4 | 136 | 0 | 15 | 2 | 4 | 2 | 1 |
| | | a8200af | 32d63f8 | 2 | 0 | 15 | 2 | 6 | 1 | 1 |
| | | 7a1b149 | c75705c | 8 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | e513a86 | 4bcc975 | 69 | 1 | 14 | 4 | 12 | 2 | 2 |
| | | 552bcb8 | 7a0b2a3 | 39 | 0 | 16 | 2 | 6 | 2 | 2 |
| | | 2150a93 | 3551f5e | 1 | 0 | 16 | 1 | 1 | 1 | 1 |
| | | 4bf14d2 | d5cf06c | 40 | 0 | 16 | 2 | 8 | 3 | 3 |
| | | 0cfe95c | 8e3fce3 | 4 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | 74ac1a9 | dc013bf | 39 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | f66eb26 | d939b45 | 1 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | 3282361 | 27e4fd5 | 31 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | 27e4fd5 | d5c229c | 20 | 1 | 13 | 4 | 28 | 1 | 1 |
| | | 9ce109b | 0eb04f0 | 48 | 0 | 15 | 2 | 6 | 2 | 2 |
| | | 14a52ff | 353e96e | 20 | 0 | 15 | 2 | 8 | 1 | 1 |
| | | 353e96e | 171d55e | 132 | 0 | 15 | 2 | 12 | 1 | 1 |
| | | fe218c2 | 636074a | 5 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | 636074a | 7ade13a | 1 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | 6af7dd2 | 4b60b8f | 4 | 0 | 15 | 2 | 5 | 3 | 3 |
| | | 1568189 | 6be139b | 1 | 0 | 15 | 2 | 2 | 1 | 1 |
| | | 8374e44 | 7840912 | 63 | 1 | 15 | 4 | 8 | 1 | 1 |
| | | dc51940 | aa8ed93 | 4 | 0 | 16 | 2 | 2 | 1 | 1 |
| | | 1e3c2b1 | f95a876 | 16 | 0 | 16 | 2 | 2 | 1 | 1 |
| | | 8fd5758 | c915a16 | 24 | 0 | 19 | 2 | 16 | 1 | 1 |
| | | 9d1e8c0 | eedb633 | 24 | 0 | 19 | 2 | 12 | 1 | 1 |
| | | 6d58f49 | c0c5249 | 92 | 0 | 20 | 2 | 4 | 1 | 1 |
| | | 26388a4 | 35fb6d5 | 6 | 0 | 19 | 2 | 9 | 2 | 2 |
| | | 4886b9b | b35b9cf | 6 | 0 | 18 | 2 | 2 | 2 | 2 |
| | | f270ca7 | f941713 | 4 | 0 | 22 | 2 | 2 | 1 | 1 |
| | | f941713 | 921a9e8 | 9 | 0 | 22 | 2 | 8 | 5 | 5 |
| | | 2b1553b | 4ca81d0 | 69 | 0 | 22 | 1 | 1 | 1 | 1 |
| | | b666a93 | f7b70da | 4 | 0 | 12 | 2 | 2 | 1 | 1 |
| | | f7b70da | fb453f4 | 1 | 0 | 12 | 2 | 2 | 1 | 1 |

| SC | Arq. | Commit | | Diff | Macros | | Filtro | | Categ. | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Mod. | | Imp. | N. Imp. | Configs | Erros | Total | Reais |
| Apache HTTPD | core.c | 8c6c690 | b35e909 | 31 | 0 | 12 | 2 | 8 | 4 | 4 |
| | | e75eebc | fbd058e | 50 | 0 | 12 | 2 | 2 | 1 | 1 |
| | | 11da826 | 327bdf3 | 5 | 0 | 14 | 2 | 11 | 2 | 2 |
| | | 0952a6c | feeb6f6 | 27 | 0 | 14 | 2 | 15 | 4 | 4 |
| | | 0b78761 | 998e2d3 | 48 | 0 | 14 | 2 | 15 | 3 | 3 |
| | | 7d26e8b | e2b76ba | 114 | 0 | 14 | 2 | 23 | 2 | 2 |
| | | e2b76ba | 3187d26 | 21 | 0 | 14 | 2 | 16 | 4 | 3 |
| | | 3187d26 | 741c7f3 | 30 | 0 | 14 | 2 | 30 | 5 | 5 |
| | | 04dd154 | e7ca31b | 48 | 0 | 14 | 2 | 22 | 5 | 5 |
| | | e7ca31b | 8ac0643 | 7 | 0 | 14 | 2 | 2 | 2 | 2 |
| | | f67f82e | 1a95045 | 39 | 0 | 14 | 2 | 40 | 10 | 10 |
| | | 41238ff | 8d9cdb3 | 41 | 0 | 14 | 2 | 18 | 4 | 4 |
| | | 8d9cdb3 | 7f85bc4 | 2 | 0 | 14 | 2 | 11 | 6 | 2 |
| | | a6367a8 | 5dc1806 | 1 | 0 | 14 | 2 | 2 | 1 | 1 |
| | | 1899578 | baebd18 | 12 | 0 | 14 | 2 | 12 | 1 | 1 |
| | | baebd18 | 6e071cc | 2 | 0 | 14 | 2 | 4 | 1 | 1 |
| | | 9163aac | 7d19004 | 1 | 0 | 14 | 2 | 2 | 1 | 1 |
| | | dd4c9b0 | bc1b918 | 103 | 0 | 14 | 2 | 7 | 5 | 5 |
| | | b9d93c4 | 2e211ad | 2 | 0 | 14 | 1 | 1 | 1 | 1 |
| | | 2e211ad | ac3c414 | 40 | 0 | 14 | 2 | 14 | 1 | 1 |
| | | b16ce45 | a7b98d4 | 3 | 0 | 14 | 2 | 2 | 1 | 1 |
| | | 38e47cd | 5d91812 | 8 | 0 | 14 | 2 | 5 | 3 | 3 |
| | | 5d91812 | ff92bb8 | 45 | 0 | 14 | 2 | 36 | 6 | 6 |
| | | ac9b04a | e13061c | 19 | 0 | 14 | 2 | 13 | 6 | 6 |
| | | e13061c | 89f2cad | 16 | 0 | 14 | 2 | 2 | 1 | 1 |
| | | 5527404 | dd647cd | 19 | 0 | 14 | 2 | 4 | 1 | 1 |
| | | e18119d | 62296b5 | 3 | 0 | 14 | 2 | 8 | 1 | 1 |
| | | 9043789 | 2f9049f | 33 | 0 | 14 | 2 | 14 | 2 | 2 |
| | | 41a8762 | 8a1e837 | 18 | 0 | 14 | 2 | 2 | 1 | 1 |
| | | dfd16b1 | a677072 | 131 | 0 | 14 | 2 | 36 | 7 | 5 |
| | | a677072 | b3d7500 | 49 | 0 | 14 | 1 | 1 | 1 | 1 |
| | | b3d7500 | 5e6c543 | 94 | 0 | 14 | 2 | 30 | 4 | 4 |
| | | 5e6c543 | fcb9b20 | 5 | 0 | 14 | 2 | 6 | 2 | 2 |
| | | 53b773b | eb41cbb | 101 | 0 | 14 | 2 | 34 | 2 | 2 |
| | | eb41cbb | 6ec77db | 4 | 0 | 14 | 2 | 2 | 1 | 1 |
| | | d4f1d82 | 380cf6c | 3 | 0 | 14 | 2 | 2 | 1 | 1 |
| | | c355fae | 146b53c | 39 | 0 | 14 | 2 | 21 | 2 | 2 |
| | | 62e4306 | f396577 | 1 | 0 | 13 | 2 | 19 | 6 | 6 |
| | | 707b7ba | 724290a | 3 | 0 | 13 | 2 | 19 | 6 | 6 |

Tabela 4.11: Principais resultados na análise do arquivo `core.c` do Apache HTTPD; SC = Sistema Configurável; Arq. = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Mod. = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Configs = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| Apache HTTPD | event | 7b62f42 | 47c0028 | 13 | 1 | 23 | 1 | 27 | 5 | 5 |
| | | c2a783f | 3f6e54f | 501 | 2 | 23 | 2 | 114 | 5 | 5 |
| | | 3d54c34 | 67c29fe | 7 | 0 | 26 | 1 | 1 | 1 | 1 |
| | | 6768dcb | 863723b | 107 | 0 | 26 | 1 | 9 | 3 | 3 |
| | | b33c28d | 61abd95 | 175 | 1 | 25 | 1 | 13 | 2 | 2 |
| | | 3acbc53 | fee8441 | 6 | 0 | 26 | 1 | 1 | 1 | 1 |

Tabela 4.12: Principais resultados na análise do arquivo `event.c` do Apache HTTPD; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

e 6ce9922) introduziu um erro de compilação relacionado ao *uso de variável não declarada*. O Código 4.1 mostra um trecho de código do arquivo original que: declara a variável `isParamEntity` sob a macro `XML_DTD`; e, implementa a função `XML_StopParser`. O Código 4.2 mostra um trecho de código do arquivo modificado. Nesta versão do arquivo, a função `XML_StopParser` chama a variável `isParamEntity`. Porém, o arquivo modificado não compila quando é compilado com a macro `XML_DTD` desabilitada, pois a variável `isParamEntity` não é declarada nesta configuração. Os desenvolvedores resolveram este erro no *commit* 79aa349 (44 dias depois). Eles incluíram o seguinte comentário na mensagem do *commit*: `Erros de compilação resolvidos quando XML_DTD e XML_CONTEXT_BYTES estão desabilitadas`. O comentário também indica outro erro de compilação que ocorreu em um arquivo que não foi analisado durante o estudo e foi causado por outra macro desabilitada. Medeiros et al. [Medeiros et al., 2015b] também avaliou o histórico do Expat. Porém, apesar de detectarem este tipo de erro, eles não detectaram o erro exibido no Código 4.2. Eles manualmente classificaram uma série de falhas, identificadas pela sua implementação consciente de variabilidade, em diferentes faltas. Este processo pode consumir tempo e tender ao erro.

CHECKCONFIGMX detectou uma transformação aplicada ao arquivo `httpd.c` do BusyBox (*commits* d2277e2 e 7291755) que introduz dois erros de compilação. As Figu-

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| Apache HTTPD | mod_include | 8acdf92 | bb97f86 | 9 | 0 | 10 | 1 | 4 | 4 | 4 |
| | | 1860c91 | ff9eb4a | 135 | 3 | 5 | 8 | 180 | 13 | 12 |
| | | d9753cf | a5f3330 | 4 | 0 | 8 | 1 | 1 | 1 | 1 |
| | | 13046c8 | 64ce6ec | 24 | 0 | 8 | 1 | 3 | 1 | 1 |
| | | d670a5e | 04455a0 | 63 | 0 | 9 | 1 | 5 | 4 | 3 |
| | | 7edd765 | 3e644f7 | 3 | 1 | 8 | 1 | 5 | 4 | 4 |
| | | a4f510b | b2ef303 | 3 | 0 | 9 | 1 | 1 | 1 | 1 |
| | | a0702ed | b8d2eeb | 6 | 0 | 13 | 1 | 1 | 1 | 1 |
| | | b8d2eeb | a83b217 | 106 | 4 | 9 | 16 | 16 | 1 | 1 |
| | | af14f6f | e857a5a | 1 | 1 | 12 | 1 | 1 | 1 | 1 |
| | | e857a5a | 3ab6187 | 9 | 0 | 13 | 1 | 1 | 1 | 1 |
| | | 3ab6187 | 42c582c | 327 | 1 | 12 | 2 | 42 | 6 | 5 |
| | | 2c5e015 | e67da29 | 24 | 0 | 13 | 1 | 2 | 2 | 2 |
| | | f5858d9 | e56d601 | 1735 | 2 | 10 | 4 | 42 | 8 | 5 |
| | | 1cedebf | 3a17c4a | 98 | 0 | 11 | 2 | 2 | 1 | 1 |
| | | aa1db80 | d480499 | 23 | 0 | 4 | 2 | 2 | 1 | 1 |
| | | d714b4c | 5f2b471 | 35 | 1 | 3 | 4 | 102 | 9 | 9 |
| | | d24a117 | c004982 | 131 | 1 | 3 | 4 | 52 | 4 | 4 |
| | | 8cf30db | 81f3651 | 39 | 0 | 4 | 2 | 4 | 2 | 2 |
| | | fa863b3 | 3565103 | 558 | 2 | 3 | 8 | 88 | 2 | 2 |
| | | 3565103 | 9cf4b52 | 180 | 1 | 4 | 2 | 10 | 1 | 1 |
| | | 16fa6c1 | 9149eca | 65 | 0 | 5 | 2 | 12 | 2 | 2 |
| | | 90261c3 | 49a9edf | 6 | 0 | 3 | 2 | 2 | 1 | 1 |
| | | c7fad31 | a2e664a | 86 | 1 | 2 | 4 | 128 | 24 | 18 |
| | | 701b528 | cda2222 | 19 | 0 | 3 | 2 | 12 | 1 | 1 |
| | | cda2222 | 956e5a4 | 1 | 0 | 3 | 2 | 4 | 1 | 1 |

Tabela 4.13: Principais resultados na análise do arquivo `mod_include.c` do Apache HTTPD; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| Apache HTTPD | mod_rewriter | a81e4fa | 5f9700c | 8 | 0 | 12 | 2 | 34 | 8 | 6 |
| | | bbc1387 | 6afd185 | 24 | 2 | 5 | 8 | 264 | 7 | 5 |
| | | f81ff58 | c2bdbf1 | 14 | 0 | 7 | 2 | 4 | 2 | 1 |

Tabela 4.14: Principais resultados na análise do arquivo `mod_rewriter.c` do Apache HTTPD; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| Apache HTTPD | proxy_util | d3fafc6 | fa881d5 | 53 | 1 | 3 | 4 | 24 | 3 | 2 |
| | | 39a24ae | 8c0c69c | 57 | 2 | 3 | 8 | 48 | 3 | 3 |
| | | dafdf2d | a366db6 | 26 | 1 | 4 | 4 | 4 | 1 | 1 |
| | | 388597b | 258a413 | 16 | 1 | 4 | 4 | 4 | 1 | 1 |
| | | 9b318e9 | 4837b01 | 59 | 1 | 4 | 2 | 2 | 1 | 1 |
| | | 669d56c | 7c3cc35 | 2 | 0 | 4 | 2 | 2 | 1 | 1 |
| | | 935de30 | e63bcb2 | 3 | 0 | 4 | 2 | 158 | 31 | 31 |
| | | e63bcb2 | 95688ec | 474 | 1 | 3 | 2 | 26 | 11 | 7 |
| | | 4407935 | c6f24e3 | 3 | 0 | 3 | 2 | 6 | 3 | 3 |

Tabela 4.15: Principais resultados na análise do arquivo `proxy_util.c` do Apache HTTPD; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

| SC | Arquivo | Commit | | Diff | Macros | | Filtro | | Categorização | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | Modificado | | Imp. | N. Imp. | Configs. | Erros | Total | Reais |
| Expat | xmlparser | 16884a7 | e4bde9b | 557 | 2 | 7 | 4 | 8 | 3 | 1 |
| | | 3370a61 | 6ce9922 | 13 | 0 | 9 | 1 | 1 | 1 | 1 |
| | | 6d06f9e | f79358c | 1 | 1 | 9 | 1 | 4 | 1 | 1 |

Tabela 4.16: Principais resultados na análise do arquivo `xmlparser.c` do Expat; SC = Sistema Configurável; Arquivo = Nome do arquivo analisado; Commit = identificador dos commits do par no repositório; Original = Sistema configurável original; Modificado = Sistema configurável modificado; Data = data que os commits foram realizados; Diff = Diferença de linhas de código entre o par; Imp. = Total macros impactadas pela transformação; N. Imp. = Total de macros não impactadas pela transformação; Config. = Configurações que apresentaram falhas na compilação; Erros = Total de mensagens de erros introduzidas pela transformação; Total = Total de erros de compilação categorizados; Reais = Total de erros de compilação categorizados após a remoção de falso positivos.

```
1 #ifdef XML_DTD
2 #define isParamEntity (...)
3 #endif
```

Código Fonte 4.1: Trecho de código do arquivo `xmlparse.c` original.

```
1 #ifdef XML_DTD
2 #define isParamEntity (...)
3 #endif
4 XML_StopParser(...){
5     if (isParamEntity) {...}
6 }
```

Código Fonte 4.2: Trecho de código do arquivo `xmlparse.c` modificado.

Figura 4.1: Trechos de código de commits consecutivos do arquivo `xmlparse.c`. Esta transformação introduz um erro de compilação do uso de variável não declarada.

ras 1.1 e 3.2 mostram trechos de código das versões originais e modificadas desse arquivo. Medeiros et al. [Medeiros et al., 2015b] estendeu a ferramenta TypeChef e avaliou o arquivo modificado. Eles encontraram apenas um dos erros que esta transformação introduz, uma vez que eles não detectam erros de compilação relacionados à *definição incompleta de tipo*. Em adição, o TypeChef foi executado em pequenos e simples exemplos de cada tipo de erro encontrado no estudo. Como resultado, foi possível concluir que a ferramenta não detecta os seguintes tipos de erro: *definição incompleta de tipo*, *nome de tipo desconhecido*, *label não declarada*, *função não pode retornar o tipo* `array`, *statement* `continue` *não está em um laço*, e *endereço de* `bit-field` *requisitado*. As abordagens de amostragem uma desabilitada e uma habilitada não detectam o erro introduzido pela transformação mostrada na Figura 3.2. Elas compilam apenas as configurações em que uma macro está desabilitada ou desabilitada. Neste exemplo, duas macros devem ser habilitadas para que o erro seja exposto.

CHECKCONFIGMX detectou uma transformação aplicada ao arquivo `core.c` do Apache HTTPD (*commits* dfd16b1 e a677072) que introduz um erro de compilação relacionado à requisição do endereço de um `bit-field`. O Código 4.5 mostra um trecho do código original do arquivo, onde a `struct conn_rec` é declarada contendo o `signed int double_reverse`. O Código 4.6 mostra um trecho do código modificado do arquivo, que tenta acessar o endereço de `double_reverse`. Esta tentativa de acesso causa um erro de compilação pois a memória pode ser acessada apenas byte a byte (8 bits). A versão modificada do arquivo contém 5.596 LOC e 14 macros. A transformação contém 131 LOC de diferença textual e não impacta macros. Os desenvolvedores resolveram este erro mais de 40 dias depois, após 3 *commits* realizados neste arquivo. O total de macros no arquivo pode aumentar a complexidade do código, levando a possível introdução de novos erros de compilação. A ferramenta TypeChef não detecta este tipo de erro.

### 4.4.2 Análise de impacto da mudança

CHECKCONFIGMX encontrou 163 pares (3.99%) com mais de quatro macros impactadas. Entre eles, 66 pares têm cinco macros impactadas, 22 pares têm seis macros impactadas, e 20 pares têm sete macros impactadas. Uma transformação do arquivo `ash.c` do BusyBox impactou 54 macros (commits cb81e64 e c470f44). A maior parte das transformações que introduziram erros de compilação impactaram ao menos uma macro. CHECKCONFIGMX

```
1  struct conn_rec {
2     signed int double_reverse:2;
3  };
4  void do_double_reverse (conn_rec
       *conn){...}
5  char * ap_get_remote_host(
       conn_rec *conn, ...)
6  {
7     if (...) {
8        do_double_reverse(conn)
9      }
10 }
11 // Este arquivo possui 31 blocos
       de #ifdefs e #ifs
12 // com 14 macros diferentes
```

Código Fonte 4.3: Trecho de código do arquivo `core.c` modificado.

```
1  struct conn_rec {
2     signed int double_reverse:2;
3  };
4  void do_double_reverse (int *
       double_reverse) {...}
5  char * ap_get_remote_host(
       conn_rec *conn){
6     if (...) {
7        do_double_reverse(&conn->
             double_reverse)
8     }
9  }
10 // Este arquivo possui 31 blocos
       de #ifdefs e #ifs
11 // com 14 macros diferentes
```

Código Fonte 4.4: Trecho de código do arquivo `core.c` modificado.

Figura 4.2: Trechos de código de commits consecutivos do arquivo `core.c`. Esta transformação introduz um erro de compilação de endereço de `bit-fiedl` requisitado.

identificou que 132 dessas transformações não impactaram macros, e 52 delas impactam apenas uma macro. Juntas, elas representam 77,7% das transformações que introduziram ao menos um erro de compilação. Esses erros podem ser detectados ao combinar os algoritmos de amostragem um habilitado e todos desabilitados e habilitados. Porém, para compilar o histórico dos arquivos analisados, o algoritmo um habilitado precisaria compilar uma média de 20 configurações por transformação com pelo menos uma macro impactada, enquanto que CHECKCONFIGMX compilou uma média de 3,42 configurações por cada uma.

Os repositórios dos sistemas configuráveis analisados contém 4,076 *commits*, CHECKCONFIGMX analisou 3,913 (96.01%) pares formados por eles. O objetivo da técnica proposta consiste em reduzir o esforço de compilar todas as possíveis configurações através da compilação apenas das impactadas. Portanto, ela é útil em analisar transformações de granularidade fina, que impactam poucas macros e, consequentemente, poucas configurações. Apesar disto, a técnica pode ajudar desenvolvedores a avaliar arquivos grandes com uma grande quantidade de macros impactadas.

O estudo focou em analisar os pares com no máximo quatro macros impactadas pois compilar mais de 32 configurações pode ser custoso. Por exemplo, com o propósito de testar a técnica proposta em transformações que envolvam mais do que quatro macros impactadas, foi analisada uma transformação aplicada ao arquivo `event.c` do Apache HTTPD (*commits* 11da82 e 273b7aa). O arquivo modificado possui 27 macros, destas apenas 7 foram impactadas pela transformação. A técnica gerou e compilou 256 configurações e não detectou erro de compilações nelas. CHECKCONFIGMX levou 65,62s para compilar as configurações, tempo quase 30 vezes maior do que a média de tempo deste passo, considerando todas as transformações avaliadas no estudo. CHECKCONFIGMX pode avaliar outras transformações com mais de quatro macros impactadas de forma similar. O desenvolvedor deve indicar o número máximo de macros impactadas que a ferramenta deve considerar.

### 4.4.3 Filtro

Após compilar todas as configurações impactadas de cada transformação, CHECKCONFIGMX realizou o passo de filtro e encontrou 5,639 mensagens de erros de compilação em 776 configurações de 214 transformações. Foram encontradas 720 novas mensagens em uma transformação aplicada ao arquivo `ash.c` do BusyBox (*commits* 59f351c and 92e13c2).

```
1  struct conn_rec {
2     signed int double_reverse:2;
3  };
4  void do_double_reverse (conn_rec
       *conn){...}
5  char * ap_get_remote_host(
       conn_rec *conn, ...)
6  {
7     if (...) {
8        do_double_reverse(conn)
9      }
10 }
11 // Este arquivo possui 31 blocos
       de #ifdefs e #ifs
12 // com 14 macros diferentes
```

Código Fonte 4.5: Trecho de código do arquivo `core.c` original.

```
1  struct conn_rec {
2     signed int double_reverse:2;
3  };
4  void do_double_reverse (int *
       double_reverse) {...}
5  char * ap_get_remote_host(
       conn_rec *conn){
6     if (...) {
7        do_double_reverse(&conn->
             double_reverse)
8     }
9  }
10 // Este arquivo possui 31 blocos
       de #ifdefs e #ifs
11 // com 14 macros diferentes
```

Código Fonte 4.6: Trecho de código do arquivo `core.c` modificado.

Figura 4.3: Trechos de código de commits consecutivos do arquivo `core.c`. Esta transformação introduz um erro de compilação de endereço de`bit-field` requisitado.

Este número é quase 100 vezes maior que a média de mensagens de erro de compilação (7,29) por transformação analisada que introduz erro de compilação. Estas 720 mensagens foram caracterizaras em 10 erros de compilação, dos quais 1 é falso positivo.

### 4.4.4 Categorização

CHECKCONFIGMX agrupou, com base em padrões, as mensagens de erro de compilação identificadas em diferentes erros de compilação por meio da análise do tipo de erro e do elemento que o causou. O objetivo deste passo é não reportar um erro duplicado, uma vez que um elemento pode causar o mesmo erro mais de uma vez na mesma configuração ou em diferentes configurações de um mesmo arquivo. CHECKCONFIGMX reduziu o número de mensagens de erros de compilação para 942 (16.7%) erros de compilação distintos. Por exemplo, o filtro encontrou as seguintes mensagens de erro para uma transformação aplicada ao arquivo `mod_include.c` do Apache HTTPD (*commits* fa863b3 e 3565103):

1. `não há membro chamado 're_result' na 'struct ssi_internal_ctx'`
   `if (!ctx->intern->re_result || !ctx->intern->re_string)`
2. `não há membro chamado 're_result' na 'struct ssi_internal_ctx'`
   `apr_size_t len = (*ctx->intern->re_result)[idx]`
3. `não há membro chamado 're_result' na 'struct ssi_internal_ctx'`
   `(*ctx->intern->re_result)[idx]`

Estas mensagens de erro de compilação são consideradas pela ferramenta como relacionados a mesma falta e portanto são categorizados em um erro de compilação do tipo *não há membro na* `struct`, relacionado ao elemento `structssi_internal_ctx`.

### 4.4.5 Falsos positivos

Os resultados do categorizador automático da CHECKCONFIGMX foram analisados manualmente. Como resultado, foram identificados que 347 (36.83%) erros de compilação

encontrados pelo Passo 4 (categorização) são falsos positivos. Alguns problemas, como bibliotecas indisponíveis e incompatibilidade de compiladores podem ter causado estes problemas. Os sistemas analisados possuem mais de 10 anos, e algumas das bibliotecas requisitadas e os compiladores utilizados na época em que foram implementados não estão mais disponíveis. Desta forma, utilizar a técnica proposta em um ambiente controlado, com todas as bibliotecas necessárias e compiladores disponíveis, iria minimizar estes problemas.

A técnica proposta implementa uma análise por arquivo. Desta forma, apenas os arquivos analisados e suas dependências foram considerados no estudo, ao invés do sistema completo. Por exemplo, todos os 159 erros de compilação reportados por CHECKCONFIGMX como introduzidos por duas transformações do arquivo `xmltok_impl.c` do Expat são falsos positivos. Os erros são relacionados ao *uso de variável não declarada*. A princípio, CHECKCONFIGMX os considerou como erros de compilação reais pois os arquivos não possuem cabeçalhos de bibliotecas (`include headers`). Porém, após uma investigação mais profunda, foi identificado que outo arquivo (`ascii.h`) define as variáveis utilizadas em `xmltok_impl.c`, e que causaram os erros de compilação. Em adição, há um terceiro arquivo (`xmlparse.c`) que inclui os dois primeiros arquivos (`ascii.h` e `xmltok_impl.c`). Portanto, estes erros não aparecem em uma análise glocal uma vez que o escopo da compilação iria incluir o arquivo que define as variáveis.

### 4.4.6 Falsos negativos

A técnica proposta foi investigada com respeito a detecção de falsos negativos através de teste de mutação. Foram aplicados manualmente 70 mutações de sete tipos nas últimas 10 versões dos seis maiores arquivos do repositório do BusyBox. A Tabela 4.17 apresenta a distribuição da aplicação dos mutantes no arquivos do sistema configurável. Os mutantes foram manualmente revisados para certificar que são são mutantes equivalentes. O escore de mutação foi utilizado para mediar a taxa de falsos positivos da CHECKCONFIGMX, uma vez que o resultado deste escore pode ser usado como referência para novos casos de teste ou para medir a qualidade de testes já existentes [Just et al., 2014]. A Tabela 4.18 apresenta os operadores de mutação utilizados para aplicar as mutações. Alguns defeitos escolhidos foram detectados por abordagens anteriores Abal et al. [2014]; Medeiros et al. [2016] mas este estudo não detectou nos sistemas analisados, por exemplo o MT1. Abordagens anteri-

ores Abal et al. [2014]; Medeiros et al. [2016, 2015b] encontraram problemas em sistemas configuráveis reais causados por mudanças similares as introduzidas por estes operadores de mutação

| | shell/ash.c | shell/hush.c | editors/vi.c | modutils/ modutils-24.c | networking/ httpd.c | networking/ ntpd.c |
|---|---|---|---|---|---|---|
| MT1 | 2 | 2 | 1 | 1 | 2 | 2 |
| MT2 | 2 | 2 | 2 | 2 | 1 | 1 |
| MT3 | 1 | 1 | 2 | 2 | 2 | 2 |
| MT4 | 1 | 2 | 2 | 2 | 2 | 1 |
| MT5 | 2 | 2 | 1 | 2 | 1 | 2 |
| MT6 | 2 | 2 | 2 | 1 | 2 | 1 |
| MT7 | 2 | 1 | 1 | 2 | 2 | 2 |

Tabela 4.17: distribuição da aplicação dos mutantes nos seis maiores arquivos do sistema configurável BusyBox.

| id | Operador de Mutação | Tipo de *field* |
|---|---|---|
| MT1 | Remover ";" | *statement* |
| MT2 | Remover declaração | função, identificador |
| MT3 | Alteração de um operador de ponteiro por outro | identificador |
| MT4 | Duplicar declaração | função, identificador |
| MT5 | Alterar tipo de uma declaração | identificador |
| MT6 | Remover *field* | *statement* |
| MT7 | Alterar tipo de retorno | função |

Tabela 4.18: Operadores de mutação aplicados à 10 versões dos seis maiores arquivos do repositório do BusyBox.

CHECKCONFIGMX matou todos os mutantes. Este resultado aumenta a confiança de que a técnica proposta pode não reportar falsos negativos. A redução do esforço para avaliar as transformações utilizando a técnica proposta em vez de uma abordagem exaustiva, como TypeChef, também foi medida. CHECKCONFIGMX identificou que 70 macros foram impactadas pela transformação, dentre um conjunto de 1.287 macros. A ferramenta avaliou uma mediana de duas configurações impactadas por transformação. A mediana de todas as configurações possíveis, sem considerar *feature models*, é 8.388.608. Portanto, CHECKCONFIGMX reduziu pelo menos 50% (uma média de 99%) o esforço para avaliar as transformações em termos de configurações compiladas.

O conjunto de operadores de mutação pode não ser representativo para todos os erros que acontecem na prática. Porém, eles foram úteis para demonstrar que a técnica proposta detecta todos os erros introduzidos pelas mutações.

### 4.4.7 Tempo

O tempo total de casa passo realizado por CHECKCONFIGMX durante este estudo foi medido. A ferramenta levou em média 6s para avaliar cada transformação aplicada ao sistema. Desenvolvedores podem executar CHECKCONFIGMX quando modificarem um arquivo de um sistema configurável com `#ifdefs`. Em alguns segundos, CHECKCONFIGMX pode reportar para eles o conjunto de possíveis erros de compilação introduzidos pelas mudanças no código.

Em algumas transformações, o tempo médio de execução foi um pouco mais alto que a média de tempo considerando todas as transformações. Por exemplo, o tempo médio para avaliar as transformações aplicada aos arquivos `modutils-24.c` do BusyBox e `mod_include.c` do Apache HTTPD foi 9s. O total de configurações impactadas afeta o tempo total para compilar o sistema configurável. O arquivo `modutils-24.c` possui quatro transformações com 16 ou 32 configurações impactadas. Em adição, foram encontrados erros de compilação em algumas dessas configurações. Portanto, estes resultados aumentam o tempo da filtragem dos erros. O tempo do filtro foi reduzido em outras transformações com o mesmo número de configurações impactadas mas sem erros de compilação detectados por CHECKCONFIGMX. A análise mais demorada ocorreu em uma transformação aplicada ao arquivo `hush.c` do BusyBox (*commits* 68d5cb5 e 3eab24e). CHECKCONFIGMX levou 20,28s para avaliar essa transformação, que impactou 32 configurações. A ferramenta levou 14s para compilar todas as configurações impactadas. A média de tempo para avaliar todas as transformações aplicadas ao arquivo `hush.c` não foi afetada por este par, uma vez que foram aplicadas 696 transformações.

Em geral, o Passo 2 (compilar as configurações) consumiu mais tempo que os demais. CHECKCONFIGMX utiliza o GCC para compilar os arquivos com as configurações impactadas e levou em média 2,5s nesse passo. Por outro lado, o passo mais rápido foi a análise do impacto da mudança (Passo 1). A ferramenta levou em média 0,07s para identificar as macros impactadas pela mudança. Por fim, ela levou em média 1,5s e 1,9s para realizar

| SC | Arquivo | Tempo (s) | | | | |
|---|---|---|---|---|---|---|
| | | AI | GCC | Filtro | Categ. | Total |
| BusyBox | ash | 0,02 | 2,02 | 0,69 | 2,49 | 5,23 |
| | httpd | 0,05 | 2,13 | 1,21 | 1,35 | 4,75 |
| | hush | 0,05 | 2,62 | 1,54 | 1,66 | 5,88 |
| | modutils-24 | 0,05 | 3,72 | 2,03 | 3,29 | 9,10 |
| | ntpd | 0,04 | 1,91 | 1,27 | 2,51 | 5,74 |
| | vi | 0,05 | 1,77 | 1,13 | 2,68 | 5,63 |
| Apache HTTPD | core | 0,05 | 2,50 | 1,68 | 1,26 | 5,49 |
| | event | 0,11 | 2,81 | 2,13 | 1,63 | 6,69 |
| | mod_include | 0,30 | 4,30 | 2,44 | 1,77 | 9,82 |
| | mod_rewrite | 0,07 | 1,77 | 1,19 | 1,30 | 4,34 |
| | proxy_util | 0,06 | 3,13 | 2,10 | 2,53 | 7,84 |
| Expat | xmlparse | 0,06 | 1,56 | 1,03 | 1,30 | 3,96 |
| | xmltok | 0,04 | 1,76 | 1,16 | 1,74 | 4,71 |
| | xmltok_impl | 0,06 | 1,44 | 1,14 | 1,29 | 3,94 |
| | **Average** | **0,07** | **2,52** | **1,54** | **1,91** | **6,06** |

Tabela 4.19: Média de tempo da execução da CHECKCONFIGMX durante a avaliação das transformações analisadas neste estudo; SS = Sistema Configurável; Arquivo = Nome do arquivo analisado; Tempo (s) = Média dos tempos; AI = Análise de Impacto (Passo 1); GCC = Seleção das configurações impactadas e execução do GCC (Passo 2); Filtro = Filtro das mensagens de erro encontradas pelo GCC (Passo 3); Categ. = Categorização das mensagens filtradas em erros de compilação (Passo 4); Total = Soma de todos os passos.

os passos de filtro (Passo 3) e de categorização (Passo 4), respectivamente. A Figura 4.19 mostra as médias de tempo que CHECKCONFIGMX levou para avaliar as transformações analisadas neste estudo.

## 4.5 Ameaças à Validade

A técnica proposta neste trabalho possui algumas ameaças à validade. Ela recebe como entrada duas versões de um mesmo arquivo. Um *feature model* Kang et al. [1990] possui um papel significativo na compilação de um sistema configurável Clements and Northrop [2009]; Pohl et al. [2005], uma vez que cada *feature* pode requisitar configurações específicas. Porém, este artefato não foi considerado durante este estudo. Ao considerar *feature models*, CHECKCONFIGMX pode encontrar menos erros de compilação nas transformações analisadas. Entretanto, outros sistemas configuráveis podem ter cenários similares na prática e CHECKCONFIGMX pode ser útil para detectá-los.

A técnica considera apenas um arquivo por vez e pode levar à falsos positivos causados por dependências (por exemplo, bibliotecas incluídas pela diretiva `#include`) com arquivos que não estão localizados na pasta padrão de bibliotecas do sistema. Em um ambiente

de desenvolvimento, CHECKCONFIGMX pode ser configurada para incluir outras pastas de bibliotecas durante a execução do GCC, diminuindo a taxa de falsos positivos. Além disto, CHECKCONFIGMX pode ser configurada considerar o sistema completo.

Durante a fase de definição da instrumentação do experimento, Ubuntu 14.04 foi utilizado para executar GCC em alguns dos sistemas, e em seguida o mesmo foi executado nos mesmos arquivos no Mac OSX Yosemite. Exatamente a mesma versão da ferramenta foi utilizada em ambos sistemas operacionais. Os resultados das compilações nos dois sistemas operacionais foram comparados e alguns resultados conflitantes surgiram. Haviam falhas do compilador no Ubuntu que não foram detectadas no Mac, e vice-versa. Além disso, Medeiros et al. [Medeiros et al., 2015b] detectou um membro não declarado em uma struct usando a sua abordagem com TypeChef no arquivo `networking/http.c` (*commit* 7291755). Este erro de compilação foi detectado no par 66 (Figura 3.2) desse estudo usando Mac OSX. No entanto, usando o Ubuntu não foi possível detectá-lo. Os diferentes resultados podem ser causados por dependências de plataformas internas do GCC, que podem conter *bugs*.

A análise de impacto da mudança realizada por CHECKCONFIGMX não considera as estruturas de C. Isto pode causar falsos positivos ou negativos. Por exemplo, se o desenvolvedor remove uma variável que foi utilizada sob uma macro diferente, CHECKCONFIGMX não irá tentar compilar este código e, consequentemente, detectar o erro se esta segunda macro não for impactada pela mudança. Apesar da análise de impacto de mudanças ser simples, CHECKCONFIGMX detectou erros de compilação. A ferramenta pode usar outra abordagem da análise de impacto de mudanças para melhorar esta etapa.

## 4.6 Respostas às Questões de Pesquisa

Baseado nos resultados do estudo, as seguintes observações foram realizadas:

- $\mathrm{RQ}_1$**:** Quais os tipos de erro de compilação encontrados por CHECKCONFIGMX? CHECKCONFIGMX detectou 595 erros de compilação introduzidos pelas transformações analisadas. Foram encontrados 20 tipos de erros de compilação em 214 (5.46%) transformações aplicadas aos 14 maiores arquivos dos repositórios do BusyBox, Apache HTTPD, e Expat. A Tabela 4.3 apresenta todos os erros de compilação encontrados pela técnica proposta.

- $\mathrm{RQ}_2$**:** Quanto esforço CHECKCONFIGMX reduz para encontrar erros de compilação em termos de configurações analisadas?

  Ao ser comparada com a abordagem exaustiva que compila todas as configurações possíveis, sem considerar *feature models*, CHECKCONFIGMX reduziu o esforço da análise em pelo menos 50% (uma média de 99%) em termos de configurações compiladas por transformação.

- $\mathrm{RQ}_3$**:** Qual a taxa de transformações que introduzem ao menos um erro de compilação?

  CHECKCONFIGMX identificou que 5.46% das transformações analisadas introduziram pelo menos um erro de compilação. A ferramenta encontrou 113 transformações que introduziram um erro de compilação; 34 transformações que introduziram dois erros de compilação; 17 transformações que introduziram três erros de compilação; 14 transformações que introduziram quatro erros de compilação; e, 35 transformações que introduziram pelo menos cinco erros de compilação.

- $\mathrm{RQ}_4$**:** Qual a taxa de erros de compilação que ocorrem em transformações que não impactaram macros?

  CHECKCONFIGMX detectou 307 erros de compilação em 132 transformações que não impactaram macros. Este resultado indica que 55,86 dos erros de compilação poderiam ter sido identificados pela abordagem de amostragem todos habilitados-desabilitados. CHECKCONFIGMX identificou que as seguintes transformações que introduziram erros de compilação: 48 impactaram uma macro; 27 impactaram duas macros; 11 impactaram três macros; e, 8 impactaram quatro macros.

- $\mathrm{RQ}_5$**:** Qual a taxa de falsos positivos encontrados por CHECKCONFIGMX?

  Manualmente foram identificados que 347 (36.83%) dos erros de compilação detectados por CHECKCONFIGMX (Passo 4) neste estudo são falsos positivos. Bibliotecas indisponíveis e versões diferentes de compiladores são as prováveis causas destes erros. O uso da ferramenta em um ambiente controlado, com todas as bibliotecas e compiladores requisitados disponíveis, deve solucionar estes problemas.

# Capítulo 5

# Trabalhos relacionados

Este capítulo apresenta os principais trabalhos relacionados. A Seção 5.1 discute a análise de sistemas configuráveis. Em seguida, a Seção 5.2 discute abordagens que realizam testes combinatoriais para detectar erros relacionados à configurações. Por fim, a Seção 5.3 apresenta trabalhos relacionados na área de análise de impacto.

## 5.1 Análise de Sistemas Configuráveis

Medeiros et al. [Medeiros et al., 2013] analisaram erros relacionados à configuração em sistemas configuráveis [Garvin and Cohen, 2011; Kuhn et al., 2004; Medeiros et al., 2015b] analisando repositórios de software e considerando erros já resolvidos pelos desenvolvedores. Abal et al. [Abal et al., 2014] analisou manualmente o repositório do núcleo do Linux para estudar problemas relacionados à configuração e encontraram 42 erros relacionados à configurações. Um banco de dados contendo o informações sobre cada erro encontrado está disponível online.[1] Além disso, eles afirmaram que ferramentas de análise automática para sistemas configuráveis que escalam para o *núcleo* do Linux não existiam. A técnica proposta neste trabalho pode ser utilizada em transformações de granularidade fina aplicadas nos arquivos do Linux.

Tartler et al. [Tartler et al., 2014] encontrou erros relacionados à configurações no *núcleo* do Linux usando a ferramenta Undertaker [Tartler et al., 2011]. Undertaker analisa todos os blocos de diretivas do pré-processador dentro de um arquivo e verifica se eles podem ser

[1] http://vbdb.itu.dk/

desabilitados ou não. Dessa forma, a ferramenta encontra um conjunto de configurações que maximiza a cobertura do sistema completo e identifica código morto. Essa abordagem não analisa uma transformação e pode analisar configurações não impactadas por mudanças de código. A técnica proposta neste trabalho foca em analisar apenas arquivos transformados e compilar apenas as suas configurações impactadas.

Existem estratégias que analisam código C na presença de diretivas de pré-processamento. Somé e Lethbridge [Somé and Lethbridge, 1998], e Garrido e Johnson [Garrido and Johnson, 2005] propuseram abordagens para pré-processar ou modificar o código-fonte antes de analisar sintaticamente sua estrutura. No entanto, esta estratégia não é interessante para analisar variabilidade, uma vez que pode perder informações sobre as diretivas de pré-processamento [Medeiros, 2016]. A técnica proposta utiliza um pré-processador C para gerar o código a ser compilado a partir das diretivas `#ifdef` presentes no código.

Kästner et al. [Kästner et al., 2011] propuseram um analisador consciente de variabilidade (TypeChef) que analisa todas as configurações de um sistema configurável C de uma só vez. Além disso, ele realiza análise de verificação de tipo [Kästner et al., 2012a; Kenner et al., 2010]. Porém, TypeChef não escala e pode não detectar alguns defeitos usados no Estudo II. Medeiros et al. [Medeiros et al., 2013] propuseram uma abordagem para lidar com o problema de escalabilidade do TypeChef através da exclusão de diretivas `#include` e utilizando *stubs* para analisar sintaticamente o código. Gazzillo e Grimm [Gazzillo and Grimm, 2012] propuseram um analisador chamado SuperC. Apesar de ser mais rápido do que TypeChef, ele não executa verificação de tipo durante a análise de código. Além disso, as ferramentas podem não escalar e deixar de detectar erros de compilação. A técnica proposta neste trabalho considera esse tipo de diretivas e compila apenas as configurações impactadas.

Embora alguns investigadores propuseram abordagens para analisar espaços de configurações completos de forma sólida para algumas classes de defeitos [Gazzillo and Grimm, 2012; Kästner et al., 2012b, 2009], a maioria das ferramentas de análise, como GCC, Clang, e Eclipse CDT,[2] consideraram apenas uma única configuração de cada vez [Medeiros, 2016]. Alguns evitam a necessidade de sintezar todo o sistema configurável para verificar a boa formação [Apel et al., 2008; Teixeira et al., 2013; Thaker et al., 2007]. Oster et al. [Oster et al., 2010], Perrouin et al. [Perrouin et al., 2010], e Johansen [Johansen et al., 2012] afirmam

[2] `https://eclipse.org/cdt/`

que a amostragem é uma alternativa viável para reutilizar ferramentas existentes na literatura, como o GCC, para detectar erros relacionados à configuração. Essas abordagens não focam nas configurações impactadas pela transformação, dessa forma pode ser que não encontrem erros de compilação relacionados à elas. A técnica proposta neste trabalho analisa as configurações impactadas pela transformação, focando onde os erros podem estar localizados. CHECKCONFIGMX estende a ferramenta GCC em sua implementação para compilar o sistema configurável com `#ifdefs` analisado.

Evans e Larochelle [Evans, 1996; Larochelle and Evans, 2001] propuseram Splint para detectar erros semânticos em C. A ferramenta verifica estaticamente programas para encontrar vulnerabilidades de segurança e codificação erros. Enquanto Novark et al. [Bond and McKinley, 2008] apresentaram Plug para detectar vazamentos de memória em programas em C e C++. Essas ferramentas podem substituir o GCC na técnica proposta.

## 5.2 Análise por Amostragem

Pesquisadores propuseram várias estratégias para lidar com espaço de configurações. Cohen et al. [Cohen et al., 1997] e Kuhn et al. [Kuhn et al., 2004] propuseram testes combinatoriais para verificar diferentes combinações de opções de configuração e priorizar casos de teste [Cohen et al., 2003; Qu et al., 2008; Sampath et al., 2008]. Marijan et al. [Marijan et al., 2013], e Bryce e Colbourn. [Bryce and Colbourn, 2006] usaram algoritmos de amostragem t-wise para cobrir todas as t combinações de opções configuração. Tartler et al. [Tartler et al., 2011] propuseram o algoritmo de amostragem cobertura de *statement* e Abal et al. [Abal et al., 2014] propuseram o algoritmo um desabilitado. Medeiros et al. [Medeiros et al., 2016] comparou 10 algoritmos de amostragem e recomendaram utilizar maioria habilitada-desabilitada e uma habilitada uma desabilitada. Diferente da técnica proposta neste trabalho, essas abordagens não analisam configurações impactadas. Apesar dos benefícios da amostragem, como a escolha das configurações amostrados não foca no código transformado, essas abordagens podem avaliar configurações que não são afetados pela transformação. Desta forma, essas abordagens podem não detectar alguns erros de compilação.

Medeiros et al. [Medeiros et al., 2015b] instanciou o TypeChef para investigar variáveis não declaradas e não utilizadas nos arquivos de cabeçalho da plataforma Linux. Os desen-

volvedores levaram vários anos para corrigir alguns questões relacionadas à configuração. A maioria dos erros relacionados à configuração envolvem apenas algumas macros, um resultado semelhante aos do estudo realizado neste trabalho (Capítulo 4), uma vez que a 96% das versões de sistemas analisadas apresentavam não mais que quatro macros impactadas por transformações. Medeiros et al. [Medeiros et al., 2016] identificaram que é difícil encontrar todos os bugs usando qualquer algoritmo de amostragem separadamente e identificou várias combinações de algoritmos que proporcionam um equilíbrio útil entre o tamanho da amostra e capacidades de detecção de falhas. Para melhorar essa atividade, Medeiros et al. [Medeiros et al., 2016] propuseram o algoritmo LSA, uma combinação dos algoritmos de amostragem todo habilitados-desabilitados, um habilidade e um desabilidade. Assim como as abordagens anteriores, essas abordagens podem não detectar erros de compilação pois não focam nas configurações impactadas e podem não analisá-las.

## 5.3 Análise de Impacto da Mudança

Law e Rothermel [Law and Rothermel, 2003] propuseram uma abordagem baseada no particionamento estático e dinâmico, e em algoritmos recursivos de chamadas gráficos para identificar métodos impactados por uma mudança. Hattori [Hattori, 2008] propôs a ferramenta Impala que implementa uma técnica de análise de impacto probabilística. A ferramenta identifica os impactos de uma transformação antes da implementação das mudanças no código e atribui probabilidades de ocorrência aos impactos através do uso de informações do histórico de mudanças do sistema analisado. O Impala recebe um conjunto de parâmetros que indicam características das transformações que desejam ser aplicadas, e estima o impacto das mudanças. A técnica proposta neste trabalho realiza uma análise estática e identifica configurações impactadas após a mudança ser realizada no código.

Ren et al. [Ren et al., 2005] propuseram Chianti, uma ferramenta de análise de impacto da mudança para Java. Zhang et al. [Zhang et al., 2012] propuseram FaultTracer, uma ferramenta de análise de impacto da mudança, que melhora Chianti ao refinar as dependências entre as mudanças atômicas, e adicionando mais regras para calcular o impacto da mudança. Com base em um conjunto de testes e as mudanças aplicadas a um programa, ambas as ferramentas recebem duas versões do programa como parâmetros, e decompõe a mudança em

mudanças atômicas e indicam os casos de teste que são impactados pela mudança. Apenas estes casos de testes precisam ser executados novamente. Chianti e FautTracer dependem de uma coleção de testes para avaliar o impacto da mudança. Grafos de chamadas são construídos a partir da execução dos testes no programa e são analisados para identificar os testes impactados. A técnica proposta neste trabalho não depende de coleção de testes. Ela recebe duas versões de um programa e identifica configurações que foram impactadas pela transformação.

Wloka et al. [Wloka et al., 2009] propuseram o JUnitMX, uma ferramenta que indica se uma coleção de testes está exercitando todas as entidades impactadas por uma transformação. Eles propuseram a métrica cobertura da mudança que avalia o quanto uma coleção de testes exercitou determinada mudança. O JUnitMX utiliza o Chianti para realizar a análise de impacto. A cobertura da mudança é medida a nível de mudanças atômicas identificadas na análise de Chianti. Mongiovi et al. [Mongiovi et al., 2014] propuseram uma ferramenta, Safira, que calcula a cobertura da mudança a nível de métodos e *statements* impactados de transformações realizadas em códigos Java. Assim como essas abordagens, a técnica proposta neste trabalho analisa transformações. Porém a análise de impacto da técnica não considera as estruturas de C, realizando a análise através de comparação textual.

# Capítulo 6

# Conclusões

Este trabalho propõe uma técnica para compilar sistemas configuráveis com `#ifdefs` baseada na análise de impacto. Para dar suporte a mudanças de código defeituosas, foi apresentada uma estratégia para detectar erros de compilação relacionados à configurações introduzidos por mudanças de código, através de uma análise consciente de variabilidade.

Para dar suporte aos desenvolvedores para implementar sistemas configuráveis escritos em C, foi apresentada CHECKCONFIGMX, uma ferramenta capaz de analisar apenas as configurações impactadas pela transformação e detectar diferente tipos de erro de compilação, incluindo erros de tipo. Ao utilizar CHECKCONFIGMX, os desenvolvedores reduzem os esforços de compilação e ganham os benefícios de um ambiente consciente de variabilidade e guiado pelo impacto da mudança para realizar mudanças nos sistemas.

Um desenvolvedor pode realizar mudanças no código e utilizar CHECKCONFIGMX para avaliar se foram introduzidos erros de compilação com pouco esforço. Porém, o desenvolvedor deve ter cautela ao tentar corrigir um erro e executar a ferramenta novamente para avaliar a nova transformação. Uma vez que CHECKCONFIGMX foca em reportar os erros introduzidos por uma transformação.

CHECKCONFIGMX considera apenas um arquivo por vez e pode levar à falsos positivos causados por dependências com arquivos que não estão localizados na pasta padrão de bibliotecas do sistema. Em um ambiente de desenvolvimento, CHECKCONFIGMX pode ser configurado para incluir outras pastas de bibliotecas durante a execução do GCC, diminuindo a taxa de falsos positivos. Além disso, a análise de impacto da CHECKCONFIGMX é realizada através da comparação textual, não considerando a estrutura de C. Isto pode causar

falso positivos ou falsos negativos. Outros analisadores de impacto podem ser utilizados para aperfeiçoar esse passo. A avaliação de falhas falso positivas e a categorização de falha para erros de compilação, ambos manuais, podem ser limitadas pela experiência dos pesquisadores. Essas atividades podem levar ao erro. A categorização pode ser automatizada através do agrupamento de mensagens similares. Apesar de dessas limitações, CHECKCONFIGMX detectou alguns erros de compilação durante a avaliação realizada neste trabalho.

CHECKCONFIGMX detectou 595 erros de compilação de 20 tipos introduzidos por 214 transformações. A maioria dos erros de compilação são relacionados a *variáveis não declaradas* (49,74%) *não há membro na* `struct` (31,76%). Cada erro de compilação detectado foi analisado manualmente e concluiu-se que 36.83% deles são falso positivos. Em geral, as bibliotecas indisponíveis e incompatibilidade de compiladores provocaram esses problemas. Utilizar a técnica proposta em um ambiente controlado, com todas as bibliotecas e compiladores necessários disponíveis, minimizaria os problemas. CHECKCONFIGMX reduziu, pelo menos, 50% (uma média de 99%) do esforço de compilar configurações comparando com a abordagem exaustiva, sem considerar *feature models*.

Para um melhor resultado na redução de esforço para analisar sistemas configuráveis com `#ifdefs` usando a CHECKCONFIGMX, o desenvolvedor deve executá-la em versões com poucas mudanças de código (granularidade fina), pois estas tendem a impactar menos macros em relação à muitas mudanças. O esforço da análise realizada pelo CHECKCONFIGMX é diretamente proporcional ao número de macros impactadas pela transformação. Por exemplo, um cenário favorável a ferramenta ocorre quando *commits* são realizados em um curto espaço de tempo.

Por não considerar o impacto das transformações aplicadas a um sistema configurável, abordagens exaustivas pode ter que analisar todas as configurações possíveis. Os resultados da avaliação mostram evidências de que CHECKCONFIGMX pode ser útil na análise de alterações de código com até a quatro macros impactadas aplicado a arquivos com um grande número de macros. Para arquivos pequenos com poucas macros CHECKCONFIGMX pode ter pouco ganho através da comparação com as abordagens exaustivas.

## 6.1 Trabalhos Futuros

Este trabalho propões uma técnica para compilar sistemas configuráveis com `#ifdefs` através da análise de impacto. Porém, esta análise de impacto é realizada por meio de comparação textual. Pretende-se substituir este passo por um algoritmo de análise de impacto mais robusto que considere a estrutura de C, e eliminar falsos positivos e falsos negativos que a implementação atual desse passo possa vir a causar. Pretende-se também automatizar e melhorar a categorização de falhas para erros.

Além disso, pretende-se criar um *plugin* do Eclipse e tornar CHECKCONFIGMX uma ferramenta de análise contínua. Desta forma, a ferramenta realizará análises em tempo de desenvolvimento. A medida que o desenvolvedor implementa o código de um sistema configurável, o *plugin* analisa se mudanças de código introduziram erros de compilação. O intervalo entre as análises pode ser configurado como a cada salvar de um arquivo, por exemplo.

A avaliação realizada neste trabalho mostrou indícios de que CHECKCONFIGMX pode ser útil em encontrar erros de compilação e que reduz o esforço de análise quando comparada à abordagem tradicional de compilar todas as configurações. Em adicional, pretende-se realizar um estudo complementar para avaliar o quão útil a técnica da ferramenta é em relação à abordagens de amostragem existentes na literatura [Abal et al., 2014; Medeiros et al., 2016; Oster et al., 2010], considerando esforço de análise e detecção de erros de compilação.

# Bibliografia

Abal, I., Brabrand, C., and Wasowski, A. (2014). 42 variability bugs in the linux kernel: A qualitative analysis. In *Proceedings of the 29th International Conference on Automated Software Engineering*, pages 421–432. ACM.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compiladores - Princípios, técnicas e ferramentas*. Addison-Wesley.

Alexander, R. T., Bieman, J. M., Ghosh, S., and Ji, B. (2002). Mutation of Java objects. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering*, pages 341–351. IEEE Computer Society.

Apel, S., Kästner, C., Groblinger, A., and Lengauer, C. (2010). Type safety for feature-oriented product lines. *Proceedings of the 25th International Conference on Automated Software Engineering*, pages 251–300.

Apel, S., Kästner, C., and Lengauer, C. (2008). Feature featherweight Java: a calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 101–112. ACM.

Apel, S., Speidel, H., Wendler, P., von Rhein, A., and Beyer, D. (2011). Detection of feature interactions using feature-aware verification. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 372–375. IEEE Computer Society.

Babar, M. A. and Zhang, H. (2009). Systematic literature reviews in software engineering: Preliminary results from interviews with researchers. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 346–355.

Basili, V. R., Caldiera, G., and Rombach, D. H. (1994). *The Goal Question Metric Approach*. John Wiley & Sons.

Berger, T., Rublack, R., Nair, D., Atlee, J. M., Becker, M., Czarnecki, K., and Wasowski, A. (2013a). A survey of variability modeling in industrial practice. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems*, pages 1–7. ACM.

Berger, T., She, S., Lotufo, R., Wasowski, A., and Czarnecki, K. (2013b). A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, pages 1611–1640.

Bodden, E., Toledo, T., Ribeiro, M., Brabrand, C., Borba, P., and Mezini, M. (2013). SPLLIFT: Statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 355–364. ACM.

Bohner, S. A. and Arnold, R. S. (1996). *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press.

Bond, M. D. and McKinley, K. S. (2008). Tolerating memory leaks. In Harris, G. E., editor, *Proceedings of the 14th International Conference on Object Oriented Programming Systems Languages and Applications*, pages 109–126. ACM.

Brabrand, C., Ribeiro, M., Toledo, T., and Borba, P. (2012). Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, pages 13–24. ACM.

Bryce, R. C. and Colbourn, C. J. (2006). Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, pages 960–970.

Cirilo, E., Nunes, I., Garcia, A., and de Lucena, C. J. (2011). Configuration knowledge of software product lines: A comprehensibility study. In *Proceedings of the 2nd International Workshop on Variability & Composition*, pages 1–5. ACM.

Classen, A., Heymans, P., Schobbens, P., Legay, A., and Raskin, J. (2010). Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 335–344. ACM.

Clements, P. (2002). Being proactive pays off. *IEEE Software*, pages 28–30.

Clements, P. and Northrop, L. (2009). *Software product lines: Practices and patterns*. Addison-Wesley.

Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. (1997). The aetg system: An approach to testing based on combinatiorial design. *IEEE Transactions on Software Engineering*, pages 437–444.

Cohen, M. B., Gibbons, P. B., Mugridge, W. B., and Colbourn, C. J. (2003). Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 38–48. IEEE Computer Society.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, pages 34–41.

Donohoe, P. (2009). Introduction to software product lines. In *Proceedings of the 13th International Software Product Line Conference*, page 305. ACM.

Ernst, M. D., Badros, G. J., and Notkin, D. (2002). An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, pages 1146–1170.

Evans, D. (1996). Static detection of dynamic memory errors. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53. ACM.

Frankl, P. G., Weiss, S. N., and Hu, C. (1997). All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, pages 235–253.

Garrido, A. and Johnson, R. E. (2005). Analyzing multiple configurations of a C program. In *Proceedings of the 21st International Conference on Software Maintenance*, pages 379–388. IEEE Computer Society.

Garvin, B. J. and Cohen, M. B. (2011). Feature interaction faults revisited: An exploratory study. In *Proceedings of the 33rd International Symposium on Software Reliability Engineering*, pages 90–99. IEEE Computer Society.

Gazzillo, P. and Grimm, R. (2012). SuperC: Parsing all of C by taming the preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 323–334. ACM.

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc.

Gruler, A., Leucker, M., and Scheidemann, K. (2008). Modeling and model checking software product lines. In *Proceedings of the 10th International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 113–131. Springer-Verlag.

Harrold, M. J. and Rothermel, G. (1994). Performing data flow testing on classes. In *SIGSOFT FSE*, pages 154–163.

Hattori, L. P. (2008). Anáise probabilística de impacto de mudanças baseada em históricos de mudanças do software. Master's thesis, Universidade Federal de Campina Grande.

Hierons, R. M., Harman, M., and Danicic, S. (1999). Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, pages 233–262.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, pages 26–60.

IEEE (1990). Ieee glossary of software engineering terminology, standard 610.12. Technical report.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions of Software Engineer*, pages 649–678.

Johansen, M. F., Haugen, O., and Fleurey, F. (2012). An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 17th International Software Product Line Conference*, pages 46–55. ACM.

Jonsson, P. and Blekinge, H. (2005). *Impact Analysis: Organisational Views and Support Techniques*. Blekinge Institute of Technology.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M., Holmes, R., and Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute.

Kästner, C., Apel, S., Thüm, T., and Saake, G. (2012a). Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol*, page 14.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 17th International Conference on Object Oriented Programming Systems Languages and Applications*, pages 805–824. ACM.

Kästner, C., Ostermann, K., and Erdweg, S. (2012b). A variability-aware module system. In *Proceedings of the 18th International Conference on Object Oriented Programming Systems Languages and Applications*, pages 773–792. ACM.

Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S. (2009). Featureide: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*, pages 611–614. IEEE.

Kenner, A., Kästner, C., Haase, S., and Leich, T. (2010). Typechef: toward type checking `#ifdef` variability in c. In *FOSD*, pages 25–32. ACM.

Korel, B. and Laski, J. W. (1990). Dynamic slicing of computer programs. *Journal of Systems and Software*, pages 187–195.

Kuhn, D. R., Wallace, D. R., and Gallo, A. M. (2004). Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, pages 418–421.

Kumar, A., Sutton, A., and Stroustrup, B. (2012). Rejuvenating C++ programs through demacrofication. In *Proceedings of the 29th International Conference on Software Maintenance*, pages 98–107. IEEE Computer Society.

Larochelle, D. and Evans, D. (2001). Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th Conference on USENIX Security Symposium*, pages xx–xx. USENIX Association.

Law, J. and Rothermel, G. (2003). Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society.

Lehnert, S. (2011). A review of software change impact analysis. Technical report, Department of Software Systems.

Li, L. and Offutt, A. (1996). Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of the International Conference on Software Maintenance*, pages 171–184. IEEE Press.

Lindvall, M. (1997). *An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Systems Evolution*. PhD thesis, Linköping University.

Machado, G., Alves, V., and Gheyi, R. (2012). Formal specification and verification of well-formedness in business process product lines. In *6th Latin American Workshop no Aspect-Oriented Software Development: Advanced Modularization Techniques, LAWASP'12*. IEEE Computer Society.

Marijan, D., Gotlieb, A., Sen, S., and Hervieu, A. (2013). Practical pairwise testing for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 227–235. ACM.

Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., and Apel, S. (2016). A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages xx–xx.

Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., and Gheyi, R. (2015a). The love/hate relationship with the C preprocessor: An interview study. In *Proceedings of the 29th European Conference on Object-Oriented Programming*, pages 495–518.

Medeiros, F., Ribeiro, M., and Gheyi, R. (2013). Investigating preprocessor-based syntax errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, pages 75–84. ACM.

Medeiros, F., Rodrigues, I., Ribeiro, M., Teixeira, L., and Gheyi, R. (2015b). An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In *Proceedings of the 15th International Conference on Generative Programming: Concepts and Experiences*, pages 35–44.

Medeiros, F. M. (2016). *An Approach to Safely Evolve Preprocessor-Based C Program Families*. PhD thesis, Universidade Federal de Campina Grande.

Mennie, C. A. and Clarke, C. L. A. (2004). Giving meaning to macros. In *Proceedings of the 12th International Workshop on Program Comprehension*, pages 79–88. IEEE Computer Society.

Mongiovi, M. (2013). Uma abordagem para avaliar refatoramentos baseada no impacto da mudança. Master's thesis, Universidade Federal de Campina Grande.

Mongiovi, M., Gheyi, R., Soares, G., Teixeira, L., and Borba, P. (2014). Making refactoring safer through impact analysis. *Science of Computer Programming*, pages 39–64.

Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20.

Oster, S., Markert, F., and Ritter, P. (2010). Automated incremental pairwise testing of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, pages 196–210. Springer-Verlag.

Perrouin, G., Sen, S., Klein, J., Baudry, B., and Traon, Y. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 459–468. IEEE Computer Society.

Pohl, K., Bockle, G., and Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag.

Qu, X., Cohen, M. B., and Rothermel, G. (2008). Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 17th International Conference on Software Testing and Analysis*, pages 75–86. ACM.

Queille, J., Voidrot, J., Wilde, N., and Munro, M. (1994). The impact analysis task in software maintenance: A model and a case study. In Müller, H. A. and Georges, M., editors, *Proceedings of the 11th International Conference on Software Maintenance*, pages 234–242. IEEE Computer Society.

Ren, X., Ryder, B. G., Stoerzer, M., and Tip, F. (2005). Chianti: A change impact analysis tool for Java programs. In *Proceedings of the 27th International Conference on Software Engineering*, pages 664–665. ACM.

Sampath, S., Bryce, R. C., Viswanath, G., Kandimalla, V., and Koru, A. (2008). Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 1st International Conference on Software Testing, Verification and Validation*, pages 141–150. IEEE Computer Society.

Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. (2015). Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 284–294. ACM.

Smith, B. H. and Williams, L. (2009). Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, pages 1819–1832.

Somé, S. S. and Lethbridge, T. (1998). Parsing minimization when extracting information from code in the presence of conditional compilation. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 118–125. IEEE Computer Society.

Spencer, H. and Collyer, G. (1992). `#ifdef` considered harmful, or portability experience with C news. In *USENIX Summer*, pages xx–xx. USENIX Association.

Spinellis, D. (2003). Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, pages 1019–1030.

Tartler, R., Dietrich, C., Sincero, J., Schröder-Preikschat, W., and Lohmann, D. (2014). Static analysis of variability in system software: The 90,000 `#ifdefs` issue. In *USENIX Annual Technical Conference*, pages 421–432. USENIX Association.

Tartler, R., Lohmann, D., Dietrich, C., Egger, C., and Sincero, J. (2011). Configuration coverage in the analysis of large-scale system software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, pages 1–5. ACM.

Teixeira, L., Borba, P., and Gheyi, R. (2013). Safe composition of configuration knowledge-based software product lines. *Journal of Systems and Software*, pages 1038–1053.

Thaker, S., Batory, D., Kitchin, D., and Cook, W. (2007). Safe composition of product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 95–104, New York, NY, USA. ACM Press.

Thum, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014). A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, pages 1–45.

Turver, R. J. and Munro, M. (1994). An early impact analysis technique for software maintenance. *Journal of Software Maintenance*, pages 35–52.

van der Linden, F., Schmid, K., and Rommes, E. (2007). *Software product lines in action - the best industrial practice in product line engineering*. Springer.

Wloka, J., Ryder, B. G., and Tip, F. (2009). Junitmx - a change-aware unit testing tool. In *Proceedings of the 31st International Conference on Software Engineering*, pages 567–570. IEEE Computer Society.

Zhang, L., Kim, M., and Khurshid, S. (2012). FaultTracer: a change impact and regression fault analysis tool for evolving Java programs. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, pages 40–40. ACM.

# Apêndice A

# Revisão Sistemática

Este apêndice descreve o processo seguido neste trabalho para realizar uma revisão sistemática sobre o tema estudado. Revisão sistemática é uma metodologia de estudo secundário que visa estabelecer um levantamento formal do estado da arte, de forma de forma robusta e consistente, a partir de um planejamento e execução criteriosos [Babar and Zhang, 2009]. Para tanto, o processo de pesquisa é conduzido segundo uma sequência metodologicamente bem definida de etapas, de acordo com um protocolo de estudo previamente planejado

## A.1 Questão de Pesquisa

Atualmente a maior parte dos sistemas complexos são configuráveis. Apesar de configurabilidade ter benefícios, mudanças de código que impactam uma opção de configuração podem ter consequências desconhecidas. Ainda mais quando as opções interagem entre si.

É comum que desenvolvedores desejem testar esses sistemas ao realizarem evoluções para evitar a adição de novos bugs. Porém, as ferramentas de testes convencionais, como GCC, não levam em consideração múltiplas configurações sendo necessário que o desenvolvedor identifique todas as configurações e execute a ferramenta escolhida para cada uma delas, podendo chegar a um número exponencial. O objetivo desta revisão é investigar as abordagens existentes na literatura. O resultado almejado é uma descrição de como o teste do código de sistemas altamente configuráveis é realizado para identificar novos bugs. Deseja-se entender por exemplo:

- Sistemas altamente configuráveis (Linhas de produto de software);

- Análise de repositórios;
- Análise de impacto; e,
- Teste de mutação.

### A.1.1 Planejamento da Revisão

Esta seção consiste na construção de definições que permitem ao pesquisador estabelecer uma distinção entre estudos relevantes e irrelevantes para o propósito específico da revisão. As seguintes estratégias de pesquisa foram consideradas: derivar a maior parte dos termos usando as características população, intervenção e saída das questões de pesquisa; identificar sinônimos para os termos encontrados; checar palavras chaves de artigos disponíveis; quando a fonte de dados permitir usar conectores boleanos and or; e, usar a pesquisa padrão das ferramentas de busca.

A Tabela A.1 sumariza os resultados destas estratégias quando colocadas em prática. Enquanto que a Tabela A.2 resume a formularização da questão de pesquisa levando em consideração estes resultados e os objetivos da revisão.

| **Característica** | **Valor** | **Lista de sinônimos (em inglês)** |
|---|---|---|
| **Intervenção** | Técnicas de análise de sistemas altamente configuráveis baseadas no impacto da mudança | Highly configurable systems; Software product line; `#ifdef`; Conditional Compilation; Repositories Analysis; Impact Analysis. |
| **População** | Desenvolvedores de software | Developer; Programmer. |
| **Saída** | Detecção de novos bugs; Corretude na detecção de novos bugs. | Mutation Test; Compilation Errors; |

Tabela A.1: Identificação da intervenção, população, saída e conjunto de sinônimos.

### A.1.2 Seleção de Trabalhos e Estudos

A tabela A.3 apresenta os critérios para seleção de *trabalhos* e de estudos.

| | | | |
|---|---|---|---|
| **Formularização da questão** | Motivação | Realizar uma revisão sistemática da literatura para um maior conhecimento no estado da arte. | |
| | Qualidade e Amplitude da Questão | Problema | Não há na literatura um estudo que explore as técnicas de análise de sistemas altamente configuráveis para identificação de novos bugs. |
| | | Questões | RQ1. Quais são as técnicas de detecção de novos bugs em sistemas altamente configuráveis?<br>RQ2. Quais são as técnicas de análise de impacto?<br>RQ3. Como testes de mutação são utilizados para avaliar a corretude de técnicas de detecção de bugs?<br>RQ4. Como repositórios de sistemas altamente configuráveis são analisados? |
| | | Palavras-chaves e Sinônimos | Software Product Line; Program Family; Conditional Compilation; `#ifdef`; Compilator Errors; Repositories; Mutants; Impact Analysis; C programming language; C; Software Evolution. |
| | | Intervenção | Artigos de conferências e periódicos. |
| | | Efeito | Maior conhecimento sobre o estado da arte na pesquisa. |
| | | Medida de resultado | Número de artigos revisados. Espera-se que a revisão contenha 50 artigos gerais e 30 bem relacionados, totalizando 80 artigos. |
| | | População | Trabalhos disponíveis nas bibliotecas digitais, como IEEE e ACM. |
| | | Aplicação | Estudo do estado da arte. |
| | | Design Experimental | Bloco randômico. |

Tabela A.2: Formularização da Questão

| | | |
|---|---|---|
| **Seleção de Trabalhos** | Definição do Critério de Seleção de Trabalhos | Em qual conferência e/ou períodico em que o paper foi publicado. |
| | Linguagem dos Estudos | Inglês e Português. |
| | Identificação de Trabalhos | Manual; Através de motores de busca da web. |
| | Referências de Verificação | Orientador |
| **Seleção de Estudos** | Definição de Critérios de Inclusão e Exclusão de Estudos | Conferência e/ou periódico em que o paper foi publicado (ICSE, ICSEM, GPCE, SPLC, SBES, SBCARS, MSR); Quantidade de citações; Resumo abordando o tema pesquisado; |
| | Definição de Tipos de Estudos | Quantitativo; Observativos; Caracterizadores. |
| | Procedimentos para Seleção de Estudos | Checar o local da publicação; Ler o abstract. |

Tabela A.3: Plano de Revisão

O procedimento para seleção de trabalhos e estudos da revisão é descrito a seguir:

1. Crie um arquivo " Primmary_Selection.docx"

2. Crie três seções:

   "Estudos aceitos";

   "Estudos rejeitados"; e,

   "Esperando por uma revisão mais profunda".

3. Para cada fonte de pesquisa:

   Dirija-se para pesquisa avançada;

   Modele a string de pesquisa usando as propriedades da fonte; e,

   Realize a pesquisa.

4. Para cada artigo retornado:

   Leia o título. Se o artigo se enquadra dentro de um critério de exclusão coloque o artigo na seção "artigos rejeitados" do arquivo "Primmary_Selection.docx", caso contrário leia o abstract;

   Leia o Abstract;

   Se o artigo se enquadra dentro de um critério de exclusão coloque o artigo na seção "Estudos rejeitados" do arquivo "Primmary_Selection.docx";

   Se não for possível responder aos critérios de exclusão coloque na seção "Esperando por uma revisão mais profunda" do arquivo "Primmary_Selection.docx";

   Caso contrário coloque na seção "Estudos aceitos" do arquivo "Primmary_Selection.docx".

5. Para todos os artigos na seção "Esperando por uma revisão mais profunda" do arquivo "Primmary_Selection.docx", leia a introdução:

   Se o artigo se enquadra dentro de um critério de exclusão coloque o artigo na seção "Estudos rejeitados" do arquivo "Primmary_Selection.docx";

   Caso contrário coloque na seção "Estudos aceitos" do arquivo "Primmary_Selection.docx";

6. Para todos os artigos na seção “Estudos aceitos” do arquivo “Primmary_Selection.docx”:

   Leia o artigo completo;

   Se o artigo se enquadra dentro de um critério de inclusão coloque o artigo na seção “Estudos aceitos” do arquivo “Primmary_Selection.docx”;

   Caso contrário coloque na seção “Estudos rejeitados” do arquivo “Primmary_Selection.docx”

7. Leia a seção trabalhos relacionados dos artigos. Se o artigo se enquadra dentro de um critério de inclusão coloque o artigo na seção “Estudos aceitos” do arquivo “Primmary_Selection.docx” Escreva o relatório utilizando os estudos da seção “Estudos aceitos” do arquivo “Primmary_Selection.docx”

## A.2 Execução da Revisão

Após o planejamento ser realizado e avaliado, a revisão sistemática é iniciada. Durante essa fase, a pesquisa no source definido deve é executada e os estudos obtidos devem ser avaliados de acordo com critérios estabelecidos. Finalmente, as informações relevantes a questão da pesquisa deve ser extraído dos estudos selecionados.

### A.2.1 Seleção dos Estudos

A seguir a lista dos trabalhos aceitos após a execução do procedimento descrito na Seção A.1.2.

- MSR

  1. Juristo, N.; Vegas, S., Using differences among replications of software engineering experiments to gain knowledge, in Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on , pp.1-1, 2-3 May 2010
  doi: 10.1109/MSR.2010.5463362

  2. Rastkar, S.; Murphy, G.C., On what basis to recommend: Changesets or interactions?, in Mining Software Repositories (MSR), 2009. International Working

Conference on , vol., no., pp.155-158, 16-17 May 2009
doi: 10.1109/MSR.2009.5069494

3. Thongtanunam, P.; McIntosh, S.; Hassan, A.E.; Iida, H., Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System, in Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on , pp.168-179, 16-17 May 2015
doi: 10.1109/MSR.2015.23

4. Khomh, F.; Dhaliwal, T.; Ying Zou; Adams, B., Do faster releases improve software quality? An empirical case study of Mozilla Firefox, in Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on , pp.179-188, 2-3 June 2012
doi: 10.1109/MSR.2012.6224279

5. Wang, S.; Khomh, F.; Ying Zou, Improving bug localization using correlations in crash reports, in Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on , pp.247-256, 18-19 May 2013
doi: 10.1109/MSR.2013.6624036

6. Shin, Y.; Bell, R.; Ostrand, T.; Weyuker, E., Does calling structure information improve the accuracy of fault prediction?, in Mining Software Repositories, 2009 MSR '09. 6th IEEE International Working Conference on, pp.61-70, 16-17 May 2009
doi: 10.1109/MSR.2009.5069481

7. Hemmati, H.; Nadi, S.; Baysal, O.; Kononenko, O.; Wei Wang; Holmes, R.; Godfrey, M.W., The MSR Cookbook: Mining a decade of research, in Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on , pp.343-352, 18-19 May 2013
doi: 10.1109/MSR.2013.6624048

- ICSE

8. Ren, X.; Ryder, B.; Stoerzer, M.; Tip, F., Chianti: a change impact analysis tool for Java programs, in Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on , pp.664-665, 15-21 May 2005

doi: 10.1109/ICSE.2005.1553643

9. Andrews, J.H.; Briand, L.C.; Labiche, Y., Is mutation an appropriate tool for testing experiments? [software testing], in Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on , pp.402-411, 15-21 May 2005
doi: 10.1109/ICSE.2005.1553583

10. Gethers, M.; Dit, B.; Kagdi, H.; Poshyvanyk, D., Integrated impact analysis for managing software changes, in Software Engineering (ICSE), 2012 34th International Conference on , vol., no., pp.430-440, 2-9 June 2012
doi: 10.1109/ICSE.2012.6227172

11. Apel, S.; Von Rhein, A.; Wendler, P.; Groslinger, A.; Beyer, D., Strategies for product-line verification: Case studies and experiments, in Software Engineering (ICSE), 2013 35th International Conference on , pp.482-491, 18-26 May 2013
doi: 10.1109/ICSE.2013.6606594

12. Holmes, R.; Notkin, D., Identifying program, test, and environmental changes that affect behaviour, in Software Engineering (ICSE), 2011 33rd International Conference on , pp.371-380, 21-28 May 2011
doi: 10.1145/1985793.1985844

13. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D., When and Why Your Code Starts to Smell Bad, in Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on , vol.1, no., pp.403-414, 16-24 May 2015
doi: 10.1109/ICSE.2015.59

14. Bhattacharya, P., Using software evolution history to facilitate development and maintenance, in Software Engineering (ICSE), 2011 33rd International Conferenceon , pp.1122-1123, 21-28 May 2011
doi: 10.1145/1985793.1986012

15. Goeritzer, R., Using impact analysis in industry, in Software Engineering (ICSE), 2011 33rd International Conference on , pp.1155-1157, 21-28 May 2011
doi: 10.1145/1985793.1986027

16. Nadi, S., A study of variability spaces in open source software, in Software Engineering (ICSE), 2013 35th International Conference on , pp.1353-1356, 18-26 May 2013
doi: 10.1109/ICSE.2013.6606715

17. Thao, C.; Managing evolution of software product line, in Software Engineering (ICSE), 2012 34th International Conference on , pp.1619-1621, 2-9 June 2012
doi: 10.1109/ICSE.2012.6227224

18. Jie Zhang, Scalability Studies on Selective Mutation Testing, in Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on , vol.2, pp.851-854, 16-24 May 2015
doi: 10.1109/ICSE.2015.276

19. Zhang; L.; Hou, S.; Hu; J. Xie, T.; Mei, H. Is operator-based mutant selection superior to random mutant selection?, in Software Engineering, 2010 ACM/IEEE 32nd International Conference on , vol.1, no., pp.435-444, 2-8 May 2010
doi: 10.1145/1806799.1806863

20. Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in software product lines. In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 311-320.
doi: 10.1145/1368088.1368131

21. Liebig, J.; Apel, S.; Lengauer, C.; Kästner, C.; Schulze, M., An analysis of the variability in forty preprocessor-based software product lines. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10), Vol. 1. ACM, New York, NY, USA, 105-114.
doi: 10.1145/1806799.1806819

22. Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 746-755.
doi: 10.1145/1985793.1985898

23. Ma, Y.; Offutt, J.; Kwon, Y., MuJava: a mutation system for java. In Procee-

dings of the 28th international conference on Software engineering (ICSE '06). ACM, New York, NY, USA, 827-830.
doi: 10.1145/1134285.1134425

24. Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 435-445.
doi: 10.1145/2568225.2568271

- ICSM

25. Ceccarelli, M.; Cerulo, L.; Canfora, G.; Di Penta, M., An eclectic approach for change impact analysis, in Software Engineering, 2010 ACM/IEEE 32nd International Conference on , vol.2, pp.163-166, 2-8 May 2010
doi: 10.1145/1810295.1810320

26. Thung, F.; Lo, D.; Lingxiao Jiang; Lucia; Rahman, F.; Devanbu, P.T., When would this bug get reported?, in Software Maintenance (ICSM), 2012 28th IEEE International Conference on , pp.420-429, 23-28 Sept. 2012
doi: 10.1109/ICSM.2012.6405302

27. Wilkerson, J.W., A software change impact analysis taxonomy, in Software Maintenance (ICSM), 2012 28th IEEE International Conference on , pp.625-628, 23-28 Sept. 2012
doi: 10.1109/ICSM.2012.6405338

28. D'Ambros, M.; Robbes, R., Effective mining of software repositories, in Software Maintenance (ICSM), 2011 27th IEEE International Conference on , pp.598-598, 25-30 Sept. 2011
doi: 10.1109/ICSM.2011.6080839

29. A. Garrido and R. Johnson, Analyzing multiple configurations of a C program, Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, 2005, pp. 379-388.
doi: 10.1109/ICSM.2005.23

- SPLC

30. White, J.; Schmidt, D.C.; Benavides, D.; Trinidad, P.; Ruiz-Cortes, A., Automated Diagnosis of Product-Line Configuration Errors in Feature Models, in Software Product Line Conference, 2008. SPLC '08. 12th International , pp.225-234, 8-12 Sept. 2008
doi: 10.1109/SPLC.2008.16

31. Ganesan, D.; Knodel, Jens; Kolb, R.; Haury, U.; Meier, G., Comparing Costs and Benefits of Different Test Strategies for a Software Product Line: A Study from Testo AG, in Software Product Line Conference, 2007. SPLC 2007. 11th International , pp.74-83, 10-14 Sept. 2007
doi: 10.1109/SPLINE.2007.21

32. Loesch, F.; Ploedereder, E., Optimization of Variability in Software Product Lines, in Software Product Line Conference, 2007. SPLC 2007. 11th International , pp.151-162, 10-14 Sept. 2007
doi: 10.1109/SPLINE.2007.31

33. Teixeira, L.; Borba, P; Gheyi, R., 2015. Safe evolution of product populations and multi product lines. In Proceedings of the 19th International Conference on Software Product Line (SPLC '15). ACM, New York, NY, USA, 171-175.
doi: 10.1145/2791060.2791084

34. Leopoldo Teixeira, Vander Alves, Paulo Borba, and Rohit Gheyi. A product line of theories for reasoning about safe evolution of product lines. In Proceedings of the 19th International Conference on Software Product Line (SPLC '15). ACM, New York, NY, USA, 161-170.
doi: 10.1145/2791060.2791105

35. Passos, L.; Guo, J.; Teixeira, L., Czarnecki, K., Wąsowski, A.,; Borba. 2013. Coevolution of variability models and related artifacts: a case study from the Linux kernel. In Proceedings of the 17th International Software Product Line Conference (SPLC '13). ACM, New York, NY, USA, 91-100.
doi: 10.1145/2491627.2491628

36. Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the linux kernel variability model. In Proceedings of

the 14th international conference on Software product lines: going beyond (SPLC'10), Jan Bosch and Jaejoon Lee (Eds.). Springer-Verlag, Berlin, Heidelberg, 136-150.

- SBES

  37. Teixeira, L.; Borba, P.; Gheyi, R., Safe Composition of Configuration Knowledge-Based Software Product Lines, in Software Engineering (SBES), 2011 25th Brazilian Symposium on , vol., no., pp.263-272, 28-30 Sept. 2011
  doi: 10.1109/SBES.2011.15

  38. Delamaro, M.; Chaim, M.; Vincenzi, A.; Jino, M.; Maldonado, J. 2011. Twenty-Five Years of Research in Structural and Mutation Testing. In Proceedings of the 2011 25th Brazilian Symposium on Software Engineering (SBES '11). IEEE Computer Society, Washington, DC, USA, 40-49.
  doi: 10.1109/SBES.2011.16

- SBCARS

  39. Ferreira, F.; Borba, P.; Soares, G.; Gheyi, R., 2012. Making Software Product Line Evolution Safer. In Proceedings of the 2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS '12). IEEE Computer Society, Washington, DC, USA, 21-30.
  doi: http://dx.doi.org/10.1109/SBCARS.2012.18

- GPCE

  40. Medeiros, F.; Ribeiro, M.; Gheyi, R., 2013. Investigating preprocessor-based syntax errors. In Proceedings of the 12th international conference on Generative programming: concepts & experiences (GPCE '13). ACM, New York, NY, USA, 75-84.
  doi: http://dx.doi.org/10.1145/2517208.2517221

  41. Medeiros, F.; Rodrigues, I.; Ribeiro, M.; Teixeira, L.; Gheyi, R,. 2015. An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015). ACM, New York, NY, USA, 35-44.
  doi: http://dx.doi.org/10.1145/2814204.2814206

42. Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. 2013. Does the discipline of preprocessor annotations matter?: a controlled experiment. In Proceedings of the 12th international conference on Generative programming: concepts & experiences (GPCE '13). ACM, New York, NY, USA, 65-74.
doi: http://dx.doi.org/10.1145/2517208.2517215

43. Sincero, J.; Tartler, R.; Lohmann, D.; Schröder-Preikschat, W., Efficient extraction and analysis of preprocessor-based variability. In Proceedings of the ninth international conference on Generative programming and component engineering (GPCE '10). ACM, New York, NY, USA, 33-42.
doi: http://dx.doi.org/10.1145/1868294.1868300

44. Neves, L.; Teixeira, L.; Sena, D.; Alves, V.; Kulezsa, U.; Borba, P., Investigating the safe evolution of software product lines. In Proceedings of the 10th ACM international conference on Generative programming and component engineering (GPCE '11). ACM, New York, NY, USA, 33-42.
doi: http://dx.doi.org/10.1145/2047862.2047869

- SPLASH

45. Medeiros, F., Safely evolving configurable systems. In Companion Proceedings of the International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2015). ACM, New York, NY, USA, 85-86.
doi: http://dx.doi.org/10.1145/2814189.2815365

46. Medeiros, F., An approach to safely evolve program families in C. In Proceedings of the companion publication of the 2014 ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '14). ACM, New York, NY, USA, 25-27.
doi: http://dx.doi.org/10.1145/2660252.2660254

47. Kästner, C.; Ostermann, K; and Erdweg, S., A variability-aware module system. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12). ACM, New York, NY,

USA, 773-792.
doi: http://dx.doi.org/10.1145/2384616.2384673

48. Kästner, C.; Giarrusso, P.; Rendel, T.; Erdweg, S.; Ostermann, K.; Berger, T.; Variability-aware parsing in the presence of lexical macros and conditional compilation. In Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11). ACM, New York, NY, USA, 805-824.
doi: http://dx.doi.org/10.1145/2048066.2048128

- PLDI

49. Bodden, E.; Tolêdo, T.; Ribeiro, M.; Brabrand, C.; Borba, P.; Mezini, M., SPLLIFT: statically analyzing software product lines in minutes instead of years. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13). ACM, New York, NY, USA, 355-364.
doi: http://dx.doi.org/10.1145/2491956.2491976

- ICTAC

50. Borba, P.; Teixeira, P.; Gheyi. R.; A theory of software product line refinement. In Proceedings of the 7th International colloquium conference on Theoretical aspects of computing (ICTAC'10), Ana Cavalcanti, David Deharbe, Marie-Claude Gaudel, and Jim Woodcock (Eds.). Springer-Verlag, Berlin, Heidelberg, 15-43.

- FSE

51. Siegmund, N.; Grebhahn, A.; Apel, S.; and Kästner, C., Performance-influence models for highly configurable systems. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 284-294.
doi: http://dx.doi.org/10.1145/2786805.2786845

- Journals

52. Borba, P.; Teixeira, P.; Gheyi, R., A theory of software product line refinement. Theor. Comput. Sci. 455 (October 2012), 2-30.
doi: http://dx.doi.org/10.1016/j.tcs.2012.01.031

53. Neves, L.; Borba, P.; Alves, V.; Turnes, L.; Teixeira, L.; Sena, D.; Kulesza, U., Safe evolution templates for software product lines. Journal of System Software. 106, C (August 2015), 42-58.
doi: 10.1016/j.jss.2015.04.024

54. Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. IEEE Transaction Software Engineer. 28, 12 (December 2002), 1146-1170.
doi: http://dx.doi.org/10.1109/TSE.2002.1158288

55. Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A theory of,software product line refinement. Theor. Comput. Sci. 455 (October,2012), 2-30.
doi: http://dx.doi.org/10.1016/j.tcs.2012.01.031

- Outros

56. Abal, I.; Brabrand, C.; Wasowski, A., 40 variability bugs in the Linux Kernel. Technical report, IT Univ. Copenhagen,Denmark, 2014.

57. Apel, S.; Batory, D.; Kaestner, C.; Saake, G., Feature-Oriented Software Product Lines: Conceptsand Implementation. Springer,2013.