

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

Uma Abordagem para Testar Implementações de
Refatoramentos Estruturais e Comportamentais de
Programas C

Gustavo Wagner Diniz Mendes

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

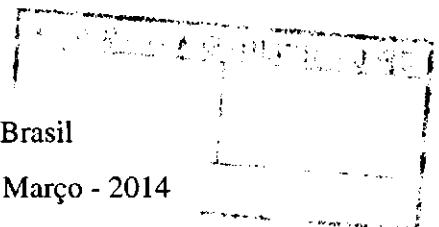
Linha de Pesquisa: Engenharia de Software

Rohit Gheyi

(Orientador)

Campina Grande, Paraíba, Brasil

©Gustavo Wagner Diniz Mendes, Março - 2014



**DIGITALIZAÇÃO:
SISTEMOTECA - UFCG**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M538a Mendes, Gustavo Wagner Diniz.
Uma abordagem para testar implementações de refatoramentos estruturais e comportamentais de programas C / Gustavo Wagner Diniz Mendes. – Campina Grande, 2014.
102 f.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2014.

"Orientação: Prof. Dr. Rohit Gheyi".

Referências.

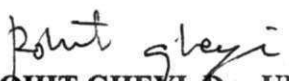
1. Engenharia de Software. 2. Testes de Programas. 3. Geração de Programas. 4. Refatoramentos. I. Gheyi, Rohit. II. Título.

CDU 004.41(043)

**"UMA ABORDAGEM PARA TESTAR IMPLEMENTAÇÕES DE REFATORAMENTOS
ESTRUTURAIS E COMPORTAMENTAIS DE PROGRAMAS C"**

GUSTAVO WAGNER DINIZ MENDES

DISSERTAÇÃO APROVADA EM 13/03/2014


ROHIT GHEYI, Dr., UFCG
Orientador(a)


MÁRCIO DE MEDEIROS RIBEIRO, Dr., UFAL
Examinador(a)


TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Refatorar um programa é o ato de mudar sua estrutura visando melhorar algum aspecto arquitetural, sem que se mude seu comportamento observável. Desenvolvedores utilizam ferramentas como Eclipse e NetBeans para auxiliá-los no refatoramento de seus programas. Essas ferramentas implementam um conjunto de condições para garantir que as transformações realizadas não mudem o comportamento observável do programa. Porém, não é trivial identificar todas as condições necessárias para que um refatoramento seja correto devido à complexidade da semântica das linguagens e por não haver uma definição formal das linguagens e conseqüentemente dos refatoramentos. Os desenvolvedores de ferramentas de refatoramento costumam, por não haver uma definição formal, implementar os refatoramentos de acordo com seus conhecimentos da linguagem de programação. Na prática, desenvolvedores utilizam coleções de testes para avaliar a corretude de suas implementações. Entretanto, não existem evidências que os desenvolvedores utilizem um processo sistemático na definição dessas coleções. Neste trabalho, propomos uma técnica para testar implementações de refatoramentos de programas C. Ela consiste em cinco fases. Primeiramente, geramos automaticamente diversos programas C para serem refatorados a partir do CDOLLY, que foi proposto baseado em uma especificação em Alloy. Depois, geramos automaticamente uma coleção de testes de unidade dos programas via o CATESTER. Em seguida, aplicamos o refatoramento através do CREXECUTOR nos programas gerados utilizando a ferramenta a ser testada. Após isso, executamos a coleção de testes nos programas refatorados e, por fim, classificamos os erros de compilação e mudanças comportamentais identificados em bugs. Avaliamos a nossa técnica na implementação de oito refatoramentos do Eclipse CDT. Estes refatoramentos não só modificam a estrutura do programa, como também o corpo das funções, como o refatoramento Extract Function. Detectamos 41 bugs, que introduzem erros de compilação no programa resultante, e seis bugs relacionados a mudanças comportamentais. Replicamos manualmente os bugs encontrados em implementações de refatoramentos do NetBeans CND, Xrefactory e OpenRefactory e detectamos 29 bugs nessas ferramentas.

Abstract

Refactoring a program is the act of changing its structure in order to improve any architectural aspect, without changing its observable behavior. Developers use tools like Eclipse and NetBeans to help refactoring their programs. These tools implement a set of conditions to ensure that the transformations performed do not change the observable behavior of the program. However, it is not easy to identify all the necessary conditions for a refactoring to be correct due to the semantic complexity of the languages, mainly because the languages are not formally proved and neither the refactorings. In practice, developers often use collections of tests to assess the correctness of their implementations. However, there is no evidence of having systematic process in the definition of this collection. In this work, we propose a technique to test refactoring implementations of C programs. It consists of five phases. First, we generate automatically several C programs to be refactored from CDOLLY, which has been proposed based on a specification in Alloy. Then, we generate automatically a collection of unit tests of the programs via CATESTER. We then apply the refactoring through CREXECUTOR on the generated programs using the tool to be tested. After this, we execute the test collection in the refactored programs and finally we classify the compilation errors and behavioral changes in bugs. We evaluated our technique in the implementation of eight refactorings of Eclipse CDT. These refactorings not only modify the program structure, but also the body functions, such as the refactoring Extract Function. We detected 41 bugs, which introduce compilation errors in the resulting program, and six bugs related to behavioral changes. We analyzed manually these bugs in refactoring implementations of NetBeans, Xrefactory and OpenRefactory and identified 29 bugs in these tools.

Agradecimentos

Agradeço primeiramente a Deus, por ter me guiado até aqui.

À minha esposa Rachel Vinagre Martins Mendes, por todo o seu amor e carinho, e por ter suportado todos esses anos de estudos e trabalhos em paralelo, ao mesmo tempo que criamos nossos lindos filhos Júlia Vinagre Mendes e Eduardo Vinagre Mendes.

Aos meus pais Gilmar de Andrade Mendes e Telma Lúcia Diniz Mendes, por terem me criado com amor, e por terem me dado condições de poder estudar. Aos meus irmãos Filipe Augusto Diniz Mendes e Brunno Ashley Diniz Mendes, simplesmente por serem meus irmãos.

Ao meu orientador e amigo Rohit, que aceitou o desafio de me orientar mesmo sabendo que eu não conseguiria me dedicar integralmente ao mestrado devido ao meu trabalho. Por ter sido paciente, amigo e conselheiro na medida certa de minhas necessidades, e por me proporcionar um avanço considerável na forma de fazer pesquisa. Rohit, você não sabe o quanto essa caminhada foi importante para mim como profissional e como pessoa.

Agradeço aos meus colegas do SPG, em especial ao Gustavo Soares, ao Jeanderson Cândido, ao Laerte Xavier e Melina Mongiovi pelas discussões, ideias e ajudas para que esse trabalho fosse possível de ser realizado.

Aos professores e funcionários da COPIN e do DSC, que me ajudaram a crescer como profissional.

Ao Tribunal Regional do Trabalho da 13ª Região (PB), em especial aos colegas e amigos Max Pereira e Hildeberto Magalhães, por terem me apoiado.

Conteúdo

1	Introdução	1
1.1	Problema	2
1.2	Relevância	3
1.2.1	Exemplo 1	4
1.2.2	Exemplo 2	5
1.3	Solução	6
1.4	Avaliação	8
1.5	Contribuições	9
1.6	Organização	9
2	CDOLLY: Um Gerador de Programas C	10
2.1	Alloy	10
2.1.1	Exemplo	11
2.1.2	Assinaturas	12
2.1.3	Fatos e Predicados	13
2.1.4	Alloy Analyzer	14
2.2	Visão Geral do CDOLLY	15
2.3	Metamodelo de C	16
2.3.1	Sintaxe Abstrata de C	16
2.3.2	Regras de Boa Formação	18
2.4	Geração de Programas C	21
2.5	Geração de Programas Específicos	22
2.6	Discussão	23
2.7	Considerações Finais	23

3	Uma Técnica para Testar Implementações de Refatoramentos de Programas C	25
3.1	Visão Geral	25
3.2	Fase 1: Geração de Programas	28
3.3	Fase 2: Geração de Testes de Unidade	28
3.3.1	Geração de Dados	28
3.3.2	Geração de Sequência	30
3.3.3	Oráculo	32
3.3.4	Expressividade de C no CATester	33
3.3.5	Exemplo	34
3.4	Fase 3: Aplicação de Refatoramento	35
3.4.1	Entradas para o Refatoramento	36
3.5	Fase 4: Execução dos Testes de Regressão	37
3.6	Fase 5: Categorização dos Bugs	37
3.6.1	Erros de Compilação	37
3.6.2	Mudança Comportamental	38
3.7	Exemplo de Aplicação da Técnica	39
3.8	Considerações Finais	43
4	Avaliação	44
4.1	Definição	44
4.2	Planejamento	45
4.2.1	Fatores e Variáveis de Resposta	45
4.2.2	Seleção das Unidades Experimentais	46
4.2.3	Instrumentação	51
4.3	Operação	52
4.4	Discussão	67
4.4.1	Técnica	67
4.4.2	Outros Tipos de Bugs	71
4.4.3	Classificador de Bugs	74
4.4.4	Status dos Bugs Reportados	75
4.5	Respostas aos Questionamentos	75

<i>CONTEÚDO</i>	vi
4.6 Ameaças à Validade	76
4.6.1 Validade Interna	76
4.6.2 Validade Externa	77
4.6.3 Validade de Construção	77
4.7 Considerações Finais	78
5 Trabalhos Relacionados	79
5.1 Geração de Testes e Programas	79
5.2 Refatoramentos	83
6 Conclusões	88
6.1 Trabalhos Futuros	91
A Teoria em Alloy do CDOLLY	99

Lista de Figuras

2.1	Parte do modelo de objetos da definição de variáveis usado no CDOLLY. . .	12
2.2	Exemplo de uma instância em Alloy.	14
2.3	Passos para geração de um programa C com CDOLLY.	15
2.4	Modelo de objetos usado no CDOLLY.	17
2.5	Tradução de Alloy para C.	22
3.1	Técnica para testar implementações de refatoramento C.	26
3.2	Componentes do CATESTER.	29

Lista de Tabelas

2.1	Regras de boa formação de programas C implementadas no CDOLLY.	20
2.2	Escopo 2 e 3 do metamodelo C.	24
3.1	Valores inicialmente utilizados como argumentos de funções no CATESTER.	30
3.2	Tipos e valores passados aos refatoramentos.	36
4.1	Tipos de refatoramentos suportados pelo Eclipse CDT.	47
4.2	Tipos de refatoramentos avaliados no experimento.	47
4.3	Resultado da avaliação de refatoramentos do Eclipse CDT pelo CREXECUTOR.	52
4.4	Bugs detectados nas ferramentas NetBeans CND, Xrefactory e OpenRefactory. Maiores detalhes sobre cada um dos bugs podem ser encontrados em http://www.dsc.ufcg.edu.br/~spg/cautomatictester	65
4.5	Tipos de refatoramentos suportados pelo NetBeans CND.	66
4.6	Tipos de refatoramentos suportados pelo Xrefactory.	66
4.7	Tipos de refatoramentos suportados pelo OpenRefactory.	66
4.8	Bugs detectados nos refatoramentos por tipo de entrada do nome do elemento a ser refatorado.	68

Lista de Códigos Fonte

1.1	Erro de compilação. Programa original.	4
1.2	Erro de compilação. Programa refatorado via <i>Extract Function</i>	4
1.3	Mudança comportamental. Programa original.	5
1.4	Mudança comportamental. Programa refatorado via <i>Rename</i>	5
2.1	Assinatura que especifica o tipo variável e suas subassinaturas.	12
2.2	Fato definindo que se uma função retornar <code>void</code> , não há uma instrução <code>return</code> na função.	13
2.3	Definição de um predicado em Alloy e seu escopo.	14
2.4	BNF incompleta da linguagem C reconhecida pelo CDOLLY.	16
2.5	Exemplo de código C gerado pelo CDOLLY.	18
2.6	Regra de boa formação. Se função não for <code>void</code> , há retorno ao final.	19
2.7	Regra de boa formação. Variável de uma atribuição numa função ou é um parâmetro ou foi declarada na função.	19
2.8	Exemplo de código gerado pelo CDOLLY que não compila, pois falta regra de boa formação.	20
2.9	Otimizações especificadas no CDOLLY.	21
2.10	Execução do predicado <i>show</i> com escopo 2.	21
2.11	Predicado para restringir as soluções do modelo.	23
2.12	Exemplo de programa gerado pelas restrições de Alloy.	23
3.1	Predicado para restringir as instâncias para refatoramento <i>Rename Function</i>	27
3.2	Programa gerado pelo CDOLLY para refatoramento <i>Rename Function</i>	27
3.3	Teste de unidade do Código Fonte 3.2 gerado pelo CATESTER.	27
3.4	Exemplos de funções.	31
3.5	Sequência da função <code>f1</code>	31

3.6	Sequência da função f_2 (estendendo sequência do Código Fonte 3.5).	31
3.7	Oráculo: exemplo da Regra 1.	32
3.8	Oráculo: exemplo da Regra 2.	33
3.9	BNF incompleta da linguagem C reconhecida pelo CATESTER.	33
3.10	Programa em C.	35
3.11	Sequência gerada pelo CATESTER.	35
3.12	Teste de unidade definido pelo CATESTER.	35
3.13	Programa gerado pelo CDOLLY para refatoramento <i>Rename Function</i>	38
3.14	Código refatorado pelo CREXECUTOR, porém não compila.	38
3.15	Programa gerado pelo CDOLLY para refatoramento <i>Rename Global Variable</i>	39
3.16	Código refatorado pelo CREXECUTOR. Ao executar, lança sinal <i>segmentation fault</i>	39
3.17	Especificando programas com características específicas para o refatoramento <i>Extract Local Variable</i>	40
3.18	Programa gerado pelo CDOLLY para refatoramento <i>Extract Local Variable</i>	41
3.19	Teste gerado pelo CATester do Código Fonte 3.18.	41
3.20	Programa refatorado por <i>Extract Local Variable</i>	41
3.21	Programa gerado pelo CDOLLY para refatoramento <i>Extract Local Variable</i>	43
3.22	Código Fonte 3.21 após refatoramento.	43
3.23	Teste de unidade do Código Fonte 3.21.	43
4.1	Exemplo de aplicação do refatoramento <i>Toggle Function</i> . Será selecionada a função <code>pow_2</code>	48
4.2	Resultado após aplicação do refatoramento <i>Toggle Function</i> na função <code>pow_2</code>	48
4.3	Exemplo de aplicação do refatoramento <i>Rename Function</i> . Será refatorada a função <code>pow_2</code>	49
4.4	Resultado após aplicação do refatoramento <i>Rename Function</i> na função <code>pow_2</code>	49
4.5	Antes da aplicação do refatoramento <i>Extract Function</i> em <code>var * var</code>	49
4.6	Depois da aplicação do refatoramento <i>Extract Function</i>	49
4.7	Antes da aplicação do refatoramento <i>Extract Constant</i> no literal 100.	50
4.8	Depois da aplicação do refatoramento <i>Extract Constant</i>	50

4.9	Antes da aplicação do refatoramento <i>Extract Local Variable</i> na expressão 100+i.	51
4.10	Depois da aplicação do refatoramento <i>Extract Local Variable</i>	51
4.11	Restrições Alloy para geração de programas pelo CDOLLY para refatora- mento <i>Rename Parameter</i>	53
4.12	Programa gerado pelo CDOLLY para refatoramento <i>Rename Parameter</i> . Eclipse permitiu refatorar param para int.	54
4.13	Bug devido a refatoramento para nome igual à palavra reservada int de C.	54
4.14	Programa gerado pelo CDOLLY para refatoramento <i>Rename Parameter</i> . Eclipse permitiu refatorar param para *param	54
4.15	Bug devido a refatoramento para nome com operador *.	54
4.16	Restrições Alloy para geração de programas pelo CDOLLY para refatora- mento <i>Rename Function</i>	55
4.17	Programa gerado pelo CDOLLY para refatoramento <i>Rename Function</i> . Eclipse permitiu refatorar func para *func.	55
4.18	Bug de <i>Rename Function</i> devido a refatoramento para nome com operador *.	55
4.19	Restrições Alloy para geração de programas pelo CDOLLY para refatora- mento <i>Rename Global Variable</i>	56
4.20	Programa gerado pelo CDOLLY para refatoramento <i>Rename Global Varia- ble</i> . Eclipse permitiu refatorar global para o literal 0.	56
4.21	Bug de <i>Rename Global Variable</i> devido a refatoramento para nome igual à literal 0.	56
4.22	Programa gerado pelo CDOLLY para refatoramento <i>Rename Global Varia- ble</i> . Eclipse permitiu refatorar global para *global.	57
4.23	Bug devido a refatoramento para nome com operador *.	57
4.24	Restrições Alloy para geração de programas pelo CDOLLY para refatora- mento <i>Rename Local Variable</i>	57
4.25	Programa gerado pelo CDOLLY para refatoramento <i>Rename Local Variable</i> . Eclipse permitiu refatorar varDecl para nome com espaço (a v).	58
4.26	Bug do <i>Rename Local Variable</i> devido a refatoramento para nome com espaço.	58

4.27 Programa gerado pelo CDOLLY para refatoramento <i>Rename Local Variable</i> . Eclipse permitiu refatorar <code>varDecl</code> para <code>*varDecl</code>	58
4.28 Bug do <i>Rename Local Variable</i> devido a refatoramento para nome com operador <code>*</code>	58
4.29 Restrições Alloy para geração de programas pelo CDOLLY para refatoramento <i>Rename #define</i>	59
4.30 Programa gerado pelo CDOLLY para refatoramento <i>Rename #define</i> . Eclipse permitiu refatorar <code>id</code> para <code>&int</code>	59
4.31 Bug do <i>Rename #define</i> devido a refatoramento para nome com operador <code>&</code>	59
4.32 Restrições Alloy para geração de programas pelo CDOLLY para refatoramento <i>Extract Function</i>	60
4.33 Programa gerado pelo CDOLLY para refatoramento <i>Extract Function</i> . Eclipse permitiu extrair função para nome de outra já existente.	61
4.34 Bug do <i>Extract Function</i> devido a refatoramento para nome de função já existente.	61
4.35 Programa gerado pelo CDOLLY para refatoramento <i>Extract Function</i> . Eclipse permitiu refatorar <code>func</code> para <code>*old</code>	61
4.36 Bug devido a refatoramento para nome com operador <code>*</code>	61
4.37 Restrições Alloy para geração de programas pelo CDOLLY para refatoramento <i>Extract Constant</i>	62
4.38 Programa gerado pelo CDOLLY para refatoramento <i>Extract Constant</i> . Eclipse permitiu extrair constante <code>10</code> para nome de variável global já existente.	63
4.39 Bug do <i>Extract Constant</i> devido a refatoramento para nome com espaço.	63
4.40 Programa gerado pelo CDOLLY para refatoramento <i>Extract Constant</i>	63
4.41 Bug devido a refatoramento para nome com operador <code>*</code>	63
4.42 Restrições Alloy para geração de programas pelo CDOLLY para refatoramento <i>Extract Local Variable</i>	64
4.43 Programa gerado pelo CDOLLY para refatoramento <i>Extract Local Variable</i> . Eclipse permitiu extrair variável local para nome <code>&old</code>	64
4.44 Bug do <i>Extract Local Variable</i> devido a refatoramento para nome com operador <code>&</code>	64

4.45 Programa gerado pelo CDOLLY para refatoramento <i>Extract Local Variable</i> . Eclipse permitiu extrair variável local para nome <code>*old</code>	65
4.46 Bug devido a refatoramento para nome com operador <code>*</code>	65
4.47 Exemplo de bug ao realizar o refatoramento <i>Extract Constant</i>	69
4.48 Isomorfismo entre programas: programa 1.	69
4.49 Isomorfismo entre programas: programa 2.	69
4.50 Bug no CATESTER com funções que retornam <code>float</code>	71
4.51 Forma aconselhada de se referenciar função de arquivo externo.	71
4.52 Código a ser refatorado com <i>Extract Function</i>	72
4.53 Código refatorado com bug de transformação.	72
4.54 Exemplo de Overly Strong Condition quando se tenta refatorar o nome do parâmetro.	73
4.55 Exemplo de bug de interface do Eclipse CDT. O mesmo refatoramento con- segue ser aplicado via API.	74
4.56 Refatorada função com espaço no nome. Erro de compilação: <i>variable has incomplete type 'void'</i>	74
4.57 Refatorada função com espaço no nome. Erro de compilação: <i>expected ';' after top level declarator</i>	74
A.1 Metamodelo de subconjunto C em Alloy.	99

Capítulo 1

Introdução

O processo de desenvolvimento de um software é constituído de várias etapas [45; 36]. Desde a sua concepção até a sua estabilização, um software recebe constantes atualizações e melhorias visando se adequar negocialmente e tecnologicamente aos requisitos definidos. Essas alterações fazem com que o código fique cada vez mais complexo, dificultando sua manutenção. Para facilitar a sua manutenção perfectiva, desenvolvedores costumam alterar o código do software visando a melhoria de sua estrutura interna, porém preservando seu comportamento externo. Preservar o comportamento externo significa que, para um mesmo conjunto de entradas, os programas original e alterado precisam apresentar as mesmas saídas. Opdyke [32] nomeou a atividade de transformação do código de um programa, mantendo seu comportamento observável, de refatoramento, e Fowler [12], mais tarde, apresentou passos para o desenvolvedor aplicar refatoramentos na prática.

Para realizar um refatoramento, condições precisam ser checadas para garantir a preservação do comportamento observável de um programa. Por exemplo, ao renomear o nome de uma função de f_1 para f_2 , é necessário garantir que não exista uma função com nome f_2 . Caso contrário, o refatoramento irá introduzir um erro de compilação no programa dependendo das regras de boa formação da linguagem, mudando seu comportamento. Para alguns tipos de refatoramentos e para programas pequenos, checar essas condições não é tão difícil. Porém, à medida que os programas crescem em quantidade de linhas de código e arquivos, é importante que as condições sejam checadas e as transformações aplicadas automaticamente para evitar inserção de bugs pelo desenvolvedor. Além disso, aplicar refatoramentos manualmente consome tempo e é propenso a erro.

Visando auxiliar os desenvolvedores, ferramentas de desenvolvimento passaram a implementar alguns tipos de refatoramentos. A primeira delas foi o Refactoring Browser, ferramenta proposta por Roberts [37] em sua tese, que define pré e pós condições para os refatoramentos. Ferramentas de refatoramentos automatizam os passos dos refatoramentos e checam as condições necessárias para aplicar as transformações.

1.1 Problema

Definir e implementar as condições necessárias para aplicar um refatoramento não são tarefas fáceis. De fato, provar corretude de refatoramento para uma linguagem é um desafio [38]. As condições para aplicar um refatoramento são definidas de maneira ad hoc de acordo com o conhecimento do desenvolvedor da semântica da linguagem. Por isso, é possível ver um refatoramento aplicado corretamente em uma ferramenta, mas não em outra.

Borba et al. [4] propuseram um conjunto de refatoramentos para a linguagem Refinement Object-Oriented Language, subconjunto da linguagem Java, e provaram a sua corretude formalmente de acordo com uma semântica formal. No entanto, Borba et al. não consideraram toda a linguagem Java. Ainda hoje, na literatura, não existe prova com relação à linguagem Java completa devido à grande complexidade de se fazer e manter após lançamentos de novas versões. Um cenário similar ocorre em C.

Opdyke [32] propôs um conjunto de sete condições para refatoramentos de programas C/C++ que preserva comportamento. Porém, esse conjunto não era completo como mostrado por Tokuda e Batory [46] posteriormente. Depois Garrido e Johnson [13] propuseram refatoramentos para C, mas não provaram formalmente.

Na prática, os desenvolvedores de IDEs implementam refatoramentos e utilizam testes unitários para avaliar a corretude dos mesmos. Porém, testes podem revelar a presença de bugs, mas não sua ausência [7]. Além disso, é importante ressaltar que a atividade de testes de implementações de refatoramentos envolve a escolha de entradas complexas (programas C), dificultando um pouco mais essa atividade. De fato, trabalhos têm revelado a presença de diversos bugs em ferramentas de refatoramento de linguagens tais como C [14] e Java [26; 41]. Isso pode levar à diminuição do uso dessas ferramentas, como verificado por Vakilian [48].

Daniel et al. [6] desenvolveram uma técnica para testar ferramentas de refatoramento a partir da geração de programas de forma imperativa. Ela é baseada no framework ASTGen, que gera exaustivamente os programas para o refatoramento, e avalia o resultado com seis oráculos. O ASTGen gera todas as combinações possíveis de um programa em um determinado escopo e os passa como entrada para os refatoramentos utilizando a abordagem *bounded-exhaustive-testing* [26]. O ASTGen detectou 21 bugs no Eclipse JDT [10] e 24 no NetBeans [28]. Dos 21 bugs do Eclipse JDT, 17 foram relacionados a erros de compilação, 3 relacionados a transformações incompletas e 1 relacionado à mudança comportamental. Gligoric et al. [15] evoluíram o ASTGen e propuseram o UDITA. O gerador do UDITA procura por todas as combinações das construções de Java para gerar programas. UDITA também permite ao testador a definição de restrições para filtrar a geração dos programas.

Soares et al. [40; 41] propuseram uma forma sistemática de testar implementações de refatoramentos Java em IDEs baseada na geração exaustiva de programas a partir de um modelo Alloy. Eles utilizaram o SAFEREFACTOR como oráculo, e detectaram mais de 50 bugs relacionados a erros de compilação e mais de 60 relacionados a mudanças comportamentais, porém não detectaram bugs em implementações de refatoramentos comportamentais. Gligoric et al. [14] também utilizam abordagem sistemática para detectar bugs em implementações de refatoramento Java e C, porém não possuem um oráculo para mudanças comportamentais.

1.2 Relevância

Ao aplicar um refatoramento, ferramentas podem introduzir erro de compilação, mudar o comportamento de um programa, não aplicar uma transformação quando deveria (*Overly Strong Condition*), quebrar a ferramenta ou transformar o código incorretamente. No contexto deste trabalho estamos interessados em dois tipos de bugs: bugs relacionados a erros de compilação e bugs relacionados à mudança comportamental.

Além disso, refatoramentos podem ser aplicados em elementos estruturais de uma linguagem, tais como funções, parâmetros ou variáveis globais, os quais chamamos de refatoramentos estruturais, ou podem ser aplicados em elementos dentro do corpo de funções, os quais chamamos de refatoramentos comportamentais.

Nesta seção, apresentamos dois exemplos de refatoramentos que introduzem bugs em

programas C. O primeiro (Seção 1.2.1) está relacionado com erro de compilação detectado após aplicação do refatoramento Renomear no Eclipse CDT [9], e o segundo (Seção 1.2.2) está relacionado à mudança comportamental após renomear uma variável dentro de uma função.

1.2.1 Exemplo 1

Considere a função `main` (linha 3) do Código Fonte 1.1. Nas linhas 6 e 7 é computado o valor de `x` elevado à segunda potência. Para reutilizar esse cálculo, já que é uma computação corriqueira neste programa, o desenvolvedor deseja definir uma função. Ele seleciona a linha 6 e aplica o refatoramento *Extract Function*, passando como nome da função o nome `pow`. O Eclipse CDT (Kepler Service Release 1) refatorou o programa e o código resultante se encontra no Código Fonte 1.2. Após essa transformação, o programa parou de compilar, pois uma variável global com mesmo nome já existia (linha 1). Desta forma, a implementação do refatoramento *Extract Function* introduziu um erro de compilação.

Código Fonte 1.1: Erro de compilação.

Programa original.

```

1 int pow = 2;
2 ...
3 void main(){
4     int x = 10;
5     ...
6     int power_2 = x * x;
7     printf("%d", power_2);
8     ...
9 }

```

Código Fonte 1.2: Erro de compilação.

Programa refatorado via *Extract Function*.

```

1 int pow = 2;
2 int pow(int x) {
3     int power_2 = x * x;
4     return power_2;
5 }
6 void main() {
7     int x = 10;
8     ...
9     int power_2 = pow(x);
10    printf("%d", power_2);
11    ...
12 }

```

É fácil perceber que a implementação do refatoramento *Extract Function* não deveria permitir extrair uma função com nome de um elemento já existente. Porém, não é tarefa

trivial definir as condições de um refatoramento em face à complexidade da semântica de algumas construções de uma linguagem de programação. Entendê-las isoladamente é mais fácil. Mas quando analisadas em conjunto, a tarefa é mais complexa. Soares et al. testaram implementações de refatoramentos aplicadas a estruturas (classes/métodos/funções) [41] de Java, porém não testaram transformações comportamentais, ou seja, transformações aplicadas a elementos dentro de corpo de função. A avaliação de Soares et al. apresenta essa limitação por que sua teoria em Alloy não suporta geração de métodos com instruções diferentes de `return`.

1.2.2 Exemplo 2

Considere a variável inteira `var` (linha 2) do Código Fonte 1.3. Ela recebe como atribuição o valor 10 e é retornada pela função `f` (linha 3). A função `main` (linha 5) imprime na saída padrão o valor 10 (linha 6). Após aplicar o refatoramento *Rename* na variável `var` (linha 2) para `*var`, o Eclipse CDT gerou o programa do Código Fonte 1.4. O programa compilou com sucesso, porém, ao executá-lo, foi lançado o sinal 11 (*Segmentation Fault*). O refatoramento, desta forma, inseriu um bug de mudança comportamental. Variáveis que são declaradas com o símbolo `*` são chamadas em C de ponteiro [22]. Ponteiro é uma estrutura que referencia um endereço de memória. Ao permitir alterar o valor da variável para `*var`, teria que ser alterado também o código de atribuição da linha 2, pois é necessária uma posição de memória para `var`, não sendo permitida a atribuição de literal 10.

Código Fonte 1.3: Mudança comportamental. Programa original.

```
1 int f(){
2   int var = 10;
3   return var;
4 }
5 void main(){
6   printf("%d", f());
7 }
```

Código Fonte 1.4: Mudança comportamental. Programa refatorado via *Rename*.

```
1 int f(){
2   int *var = 10;
3   return *var;
4 }
5 void main(){
6   printf("%d", f());
7 }
```

Neste caso, uma ferramenta de refatoramento poderia tanto alocar uma posição de memória para `*var` utilizando a função `malloc`, quanto não permitir esse tipo de refatoramento. A técnica de testes de ferramentas de refatoramento de Gligoric et al. [14] detectam automaticamente bugs de compilação, porém não possuem um oráculo para detectar bugs de mudança comportamental.

1.3 Solução

Neste trabalho, propomos uma abordagem para testar implementações de refatoramentos de programas C para detectar erros de compilação e mudanças comportamentais. Nossa abordagem é inspirada na de Soares et al. [41], porém utilizada à linguagem C e aplicada também em refatoramentos comportamentais, ou seja, refatoramentos aplicados a elementos dentro do corpo de funções. É importante testar esses tipos de refatoramentos comportamentais pois linguagens estruturadas como C, diferentemente de linguagens que implementam o paradigma orientado a objetos, não contém elementos estruturais tais como interface e classe e conceitos tais como herança e polimorfismo. Nossa abordagem tem o intuito de testar refatoramentos que mudam tanto a estrutura de um programa em C (declaração de funções, variáveis globais, macro `#define`) quanto seu comportamento (corpo das funções). Ela consiste em cinco fases. Primeiro, geramos automaticamente diversos programas C para serem refatorados a partir do CDOLLY, que foi proposto baseado em uma especificação em Alloy [19]. Depois, geramos automaticamente uma coleção de testes de unidade dos programas via o CATESTER, uma ferramenta proposta por nós para gerar testes de unidade para programas C a partir do código fonte. Em seguida, aplicamos o refatoramento através do CREXECUTOR nos programas gerados utilizando a API de refatoramentos da ferramenta a ser testada. Após isso, executamos a coleção de testes nos programas refatorados e, por fim, classificamos os erros de compilação e mudanças comportamentais identificados em bugs.

Fase 1: Geração de Programas

Na Fase 1, geramos programas em C que servirão de entrada para os testes das ferramentas de refatoramento. Os programas são gerados a partir de uma ferramenta que desenvolvemos chamada CDOLLY. CDOLLY contém um metamodelo de um subconjunto C especificado na linguagem formal Alloy [19] e executa o Alloy Analyzer [20] para gerar

instâncias do modelo. Cada instância é traduzida para um programa C, e os programas que compilam são usados na próxima fase.

CDOLLY foi baseada em JDolly [44], ferramenta desenvolvida por Soares que consiste num gerador de programas Java. Soares comparou JDolly com ASTGen e UDITA e mostrou que sua abordagem declarativa para geração de programas é mais fácil para o desenvolvedor do que a abordagem imperativa de ASTGen e UDITA.

Fase 2: Geração de Testes de Unidade

Nesta fase, definimos automaticamente testes de unidade a partir da ferramenta CATER. Para cada função de um programa, são gerados testes de unidade que servirão para avaliar mudanças comportamentais nos refatoramentos aplicados. O CATER é formado por três módulos: gerador de dados, gerador de sequências e oráculo. O gerador de sequências lê todas as funções de um programa C e gera sequências de execução para cada função. Uma sequência consiste em instruções em C para execução de uma função. Para passar os valores esperados a cada função, o gerador de dados escolhe aleatoriamente valores de acordo com os tipos dos argumentos das funções e o gerador de sequências utiliza esses valores para a chamada de função. Cada sequência é traduzida em um teste de unidade que utiliza como oráculo valores de retorno das execuções do programa e lançamento ou não de sinal para definir suas asserções.

Fase 3: Aplicação de Refatoramento

Nesta fase, aplicamos o refatoramento através do CREXECUTOR nos programas gerados na Fase 1 utilizando a API de refatoramento da ferramenta a ser testada. Desenvolvemos o CREXECUTOR para receber como entrada o refatoramento a ser aplicado, o elemento a ser refatorado e os programas gerados na Fase 1. O refatoramento é aplicado em cada programa, e os programas refatorados que compilam passam por testes de regressão na próxima fase. Os programas que pararam de compilar após o refatoramento serão analisados na Fase 5.

Fase 4: Execução dos Testes de Regressão

Nesta fase, os programas refatorados são testados com os testes de unidade gerados na Fase 2. Os testes realizados podem falhar por mudança comportamental, e essa mudança pode ser detectada ou por mudança no valor esperado pelo teste ou por um lançamento de sinal. Em ambos os casos, na Fase 5 serão categorizados os bugs.

Fase 5: Categorização dos Bugs

Nesta quinta e última fase, categorizamos os bugs causados pelo refatoramento em dois tipos: erros de compilação ou mudança comportamental. Para os erros de compilação, usamos o classificador automático proposto por Jagannath [21], que se baseia no agrupamento das mensagens de erro. Para os erros de mudança comportamental, usamos a noção de equivalência proposta por Opdyke [32]: todos os programas que são refatorados, mas que apresentam diferenças de resultados entre o original e o refatorado, são analisados para categorizarmos os bugs. Dividimos em dois grupos os programas que falharam por mudança comportamental: aqueles que lançaram sinal e aqueles que não lançaram. Como os programas que lançam sinais não concluem sua execução, os colocamos na lista dos programas com mudança comportamental. Os que não lançam sinais precisam ser avaliados quanto ao resultado da execução e comparar com o programa original. Os bugs dos programas que lançaram sinais são categorizados de acordo com o sinal. Os programas que não lançaram sinal são avaliados manualmente para categorização do bug.

1.4 Avaliação

Avaliamos nossa técnica com oito refatoramentos implementados pelo Eclipse CDT. Não analisamos o refatoramento *Toggle Function* pois CDOLLY não leva em consideração programas com mais de um arquivo. Não analisamos o refatoramento *Hide Method* porque ele é aplicado apenas em elementos da linguagem orientada a objetos C++, e nossa análise foi feita apenas para elementos da linguagem estruturada C. Dividimos o refatoramento *Rename* em quatro tipos distintos, perfazendo assim oito refatoramentos do Eclipse, pois facilita o entendimento e acreditamos que os elementos da linguagem C os quais o refatoramento *Rename* refatora são diferentes o suficiente para essa divisão. Geramos mais de 83.000 programas, e detectamos 41 bugs que introduzem erros de compilação no programa resultante, e seis bugs relacionados a mudanças comportamentais. Avaliando manualmente alguns dos bugs encontrados no Eclipse CDT identificamos 29 bugs em implementações de refatoramentos do NetBeans CND [31], Xrefactory [51] e OpenRefactory [17]. Detectamos, além de bugs relacionados a falhas de compilação e mudança comportamental, bug de interface gráfica, bug de transformação e *Overly Strong Condition* [43].

O resultado de nosso trabalho mostrou evidências de que a abordagem de Soares et

al. [41] funciona para uma linguagem de outro paradigma (C), e que também é útil para testar refatoramentos aplicados a corpo de função.

1.5 Contribuições

As principais contribuições deste trabalho são:

- um gerador automático de programas C (CDOLLY) baseado em Alloy (Capítulo 2);
- uma abordagem para testar implementações de refatoramentos estruturais e comportamentais de programas C baseada no CDOLLY e no CATESTER (Capítulo 3) que encontrou em 8 implementações de refatoramentos 47 bugs no Eclipse CDT e 29 bugs no NetBeans, no Xrefactory e OpenRefactory (Capítulo 4).

1.6 Organização

Este trabalho está organizado da seguinte forma. No Capítulo 2, apresentamos nosso gerador de programas C. No Capítulo 3, mostramos nossa técnica para testar ferramentas de refatoramento. No Capítulo 4, descrevemos os resultados da avaliação realizada com nossa técnica. No Capítulo 5, apresentamos e discutimos os trabalhos relacionados e no Capítulo 6 tecemos as considerações finais e apresentamos os trabalhos futuros.

Capítulo 2

CDOLLY: Um Gerador de Programas C

Apresentamos neste capítulo CDOLLY¹, um gerador de programas C baseado em Alloy. Especificamos o metamodelo de um subconjunto C na linguagem de especificação Alloy e usamos o Alloy Analyzer [20] para gerar instâncias do modelo. Por fim, traduzimos cada instância gerada do modelo para um programa C.

Na próxima seção apresentamos uma visão geral da linguagem de especificação Alloy. Na Seção 2.2 mostramos o CDOLLY. Na Seção 2.3 descrevemos o metamodelo de C. Em seguida mostramos como gerar programas com características específicas (Seção 2.5) e apresentamos as considerações finais (Seção 2.7).

2.1 Alloy

Nesta seção, apresentamos uma visão geral da linguagem de especificação Alloy. Alloy é uma linguagem declarativa baseada em lógica de primeira ordem. As estruturas definidas em Alloy usam um conjunto mínimo de noções matemáticas e notação de modelagem de objetos, tais como expressões de navegação, que facilitam a modelagem. Ela possui um analisador, o Alloy Analyzer [20], pelo qual são geradas instâncias do modelo especificado. As principais características de Alloy são:

- Linguagem Declarativa: em Alloy, especifica-se o que deve ser gerado, diferentemente de uma linguagem imperativa, que se especifica como gerar;

¹CDOLLY pode ser baixado em: <https://bitbucket.org/gugawag/cdolly>.

- **Análise Automática:** podemos analisar automaticamente os modelos de Alloy a partir da geração de exemplos;
- **Checagem de Escopo:** podemos definir escopo para geração de instâncias do modelo. O escopo delimita a quantidade de um determinado elemento num modelo. Ao delimitar um escopo, a análise do modelo torna-se finita.

Uma especificação em Alloy consiste em uma sequência de parágrafos, que podem ser:

- **Declaração de Assinaturas:** serve para definir novos tipos. Uma assinatura é marcada pela palavra reservada `sig`, e pode conter campos, cada um representando uma relação;
- **Declaração de Restrições:** serve para definir restrições e expressões. As restrições podem ser especificadas por fatos (`fact`), predicados (`pred`) ou funções (`func`);
- **Comandos:** representam instruções para o analisador executar análises específicas. Podem ser `run` ou `check`.

2.1.1 Exemplo

Para ilustrar o uso dos elementos da linguagem Alloy, mostraremos um exemplo representando parte do metamodelo de um subconjunto `C` reconhecido pelo CDOLLY (Figura 2.1). Esse modelo representa as variáveis, seus tipos e subtipos em `C`. Uma variável (`Variable`) pode ser de qualquer tipo, menos `void` (`Type-Void`). Um tipo `Type` pode ser `Void`, ou tipo básico (`BasicType`). Os tipos básicos representados nesse modelo podem ser inteiro (`Integer_`) ou ponto flutuante (`Float`).

Cada retângulo do modelo de objetos representa um conjunto de objetos. Na linguagem Alloy, cada retângulo é declarado como uma assinatura (`sig` na linguagem Alloy). As setas que ligam os objetos representam relações entre os conjuntos de objetos, e cada relação pode ter uma multiplicidade. Por exemplo, a relação entre `Variable` e `Type` tem multiplicidade 1, significando que uma variável só tem um tipo.

Uma assinatura pode ter subassinaturas (`GlobalVar`), representadas no modelo de objetos por uma relação com triângulo fechado. Cada subassinatura é um subconjunto do con-

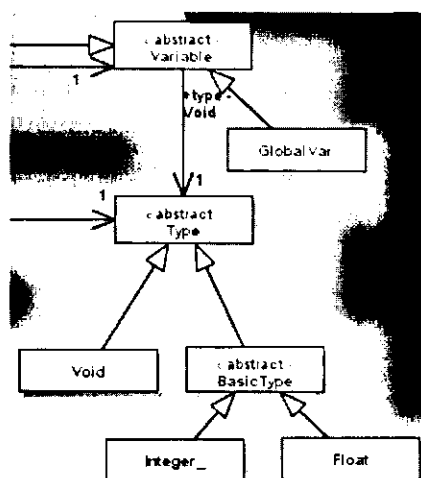


Figura 2.1: Parte do modelo de objetos da definição de variáveis usado no CDOLLY.

junto referenciado. Os tipos `Integer_` e `Float`, por exemplo, são subconjuntos disjuntos de `BasicType`. Nas próximas seções mostramos as estruturas de Alloy.

2.1.2 Assinaturas

Uma assinatura introduz a um modelo um conjunto de objetos. Ela pode conter relações dos objetos com outros objetos. Cada relação é declarada como um atributo da assinatura. O Código Fonte 2.1 mostra a assinatura `Variable` e sua subassinatura `GlobalVar`. Em cada relação é possível definir sua multiplicidade, que pode ser `set` (qualquer quantidade), `one` (apenas um), `lone` (zero ou um) ou `some` (um ou mais). Na assinatura `Variable` é definida uma relação com a assinatura `Type` de nome `type` (lado esquerdo na linha 2), com multiplicidade `one`. Também é especificado que a relação não conterá o tipo `Void` (`Type-Void`). A palavra `abstract` especifica que a assinatura só conterá elementos dos seus subconjuntos, e não elementos próprios.

Código Fonte 2.1: Assinatura que especifica o tipo variável e suas subassinaturas.

```

1 abstract sig Variable {
2   type: one Type-Void
3 }
4 sig GlobalVar extends Variable {}

```

Uma assinatura pode conter subassinaturas, fazendo com que uma subassinatura represente um subconjunto de uma assinatura. Uma subassinatura herda todas as relações da assi-

natura pai. No Código Fonte 2.1, é definida uma subassinatura chamada `GlobalVar` (linha 4). A palavra reservada `extends` define que `GlobalVar` é subconjunto de `Variable`.

Além de declaração de assinaturas e relações, é possível declarar restrições ao modelo. No CDOLLY, usamos essas restrições para definir as regras de boa formação da linguagem C. Na próxima seção mostramos como podemos definir restrições no Alloy.

2.1.3 Fatos e Predicados

Fatos são restrições consideradas sempre verdadeiras (invariantes). Um modelo pode ter qualquer quantidade de fatos. No Código Fonte 2.2, é mostrado um fato (`fact`) com a restrição especificando que funções com retorno `void` não têm instrução `return`. No lado esquerdo da restrição (linha 2), é usado o quantificador `all` (para todo), significando que a restrição é válida para todas as funções. Além desse quantificador, há outros, tais como `some` (para algum), `no` (para nenhum), `lone` (para até um) e `one` (para exatamente um). No lado direito, é especificado que se o tipo de retorno da função `f` for `Void`, isso implica que (`=>`) não haverá retorno (`no f.stmt.elems&Return`). O símbolo `&` é usado para união de conjuntos. Neste caso, o conjunto dos elementos das instruções de uma função (`f.stmt.elems`) e o conjunto `Return`.

Código Fonte 2.2: Fato definindo que se uma função retornar `void`, não há uma instrução `return` na função.

```
1 fact {
2     all f:Function | f.returnType = Void => no f.stmt.elems&Return
3 }
4 ...
```

Além de fatos, Alloy permite descrever restrições como predicados. Um predicado contém restrições que são verdadeiras sempre que o predicado for utilizado (executado), diferentemente do fato, que é sempre verdade. Por exemplo, no Código Fonte 2.3 é definido o predicado `show`, que especifica que deve ser gerada apenas uma função para cada instância do modelo (linhas 1-3). Para que isso ocorra, é necessário que o predicado seja executado (linha 4). Como pode ser visto neste exemplo, a sintaxe para se definir um predicado é similar ao de um fato, sendo diferente apenas a palavra reservada `pred`.

2.1.4 Alloy Analyzer

A linguagem Alloy possui parágrafos que realizam análises (asserções, execuções e checagens) no modelo especificado. Essa análise é feita pelo Alloy Analyzer, que pode ser usado para encontrar instâncias de um modelo, ou checar se uma propriedade do modelo é verdadeira para um escopo específico.

Código Fonte 2.3: Definição de um predicado em Alloy e seu escopo.

```

1 pred show[] {
2   #Function = 1
3 }
4 run show for 3

```

No Código Fonte 2.3 é mostrado o predicado `show` especificando a quantidade de funções `#Function` (1) para essa execução do modelo. Além disso, é especificado que o predicado `show` execute (`run`) com escopo 3 (linha 4). O comando `run` ordena ao Alloy Analyzer buscar todas as instâncias do modelo para um determinado escopo. É possível visualizar graficamente as instâncias do modelo encontradas pelo Alloy Analyzer. Um exemplo pode ser visto na Figura 2.2.

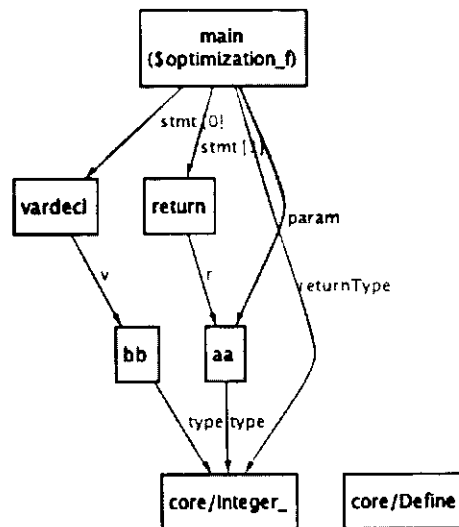


Figura 2.2: Exemplo de uma instância em Alloy.

Na próxima seção daremos uma visão geral de como o CDOLLY usa o Alloy para especificar um subconjunto da linguagem C. No Apêndice A pode ser vista toda a teoria Alloy do CDOLLY.

2.2 Visão Geral do CDOLLY

O CDOLLY é um gerador de programas C que se baseia na linguagem de especificação do Alloy. Como pode ser visto na Figura 2.3, a partir do metamodelo C (1), o Alloy Analyzer gera instâncias do modelo (2) e o CDOLLY traduz cada instância em um programa C (3). Como a quantidade de instâncias pode ser muito grande, usamos um escopo para reduzir o espaço de estados, reduzindo por consequência a quantidade de instâncias geradas pelo Alloy Analyzer. Um escopo delimita a quantidade máxima de elementos de cada tipo do modelo. Por exemplo, para gerar todos os programas em C que contenham apenas duas funções, executamos o modelo em Alloy com escopo 2 para funções. É possível também especificar o escopo por elemento do modelo. Por exemplo, podemos delimitar o escopo para gerar todos os programas que tenham apenas uma função com três instruções (*statements*) e uma variável global.

Como visto, além do escopo, Alloy permite especificar restrições (*constraints*). Por exemplo, podemos implementar uma restrição para que toda função retorne o tipo inteiro (*int*). Em suma, escopo delimita a quantidade de elementos que cada assinatura da especificação em Alloy pode receber, enquanto que restrições impõe regras de formação dos elementos.

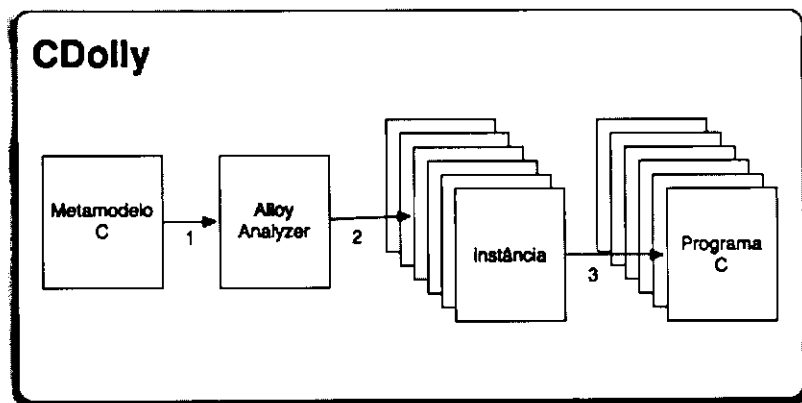


Figura 2.3: Passos para geração de um programa C com CDOLLY.

2.3 Metamodelo de C

Nesta seção, apresentamos o metamodelo do subconjunto da linguagem C implementado no CDOLLY. O subconjunto foi escolhido a partir de estudos sobre construções não triviais de C [50], tais como `#ifdef`, a partir de estudos de construções de C que apresentaram mais bugs [14] e a partir da nossa experiência como desenvolvedores. Definimos, assim, a sintaxe abstrata do modelo (Seção 2.3.1) e as regras de boa formação da linguagem C em Alloy (Seção 2.3.2).

2.3.1 Sintaxe Abstrata de C

Um típico programa C é formado por funções, que podem ter parâmetros e instruções. As instruções podem ser declarações e atribuições de variáveis, laços de repetição tais como `for` e `while`, controles de decisão, tais como `if` e `switch` e chamadas de funções. Deve conter uma função com nome `main`, e pode ser formado por um ou mais arquivos: os arquivos que contêm as declarações das funções, chamados de header (`.h`), e os arquivos com o código fonte em si, com extensão (`.c`).

A partir desses elementos que formam um programa C, definimos um subconjunto da linguagem, representado pela BNF da Listagem 2.4 e resumido pelo modelo de objetos representado na Figura 2.4. Por este subconjunto, os programas C podem conter variáveis globais (*GlobalVar*), macros do tipo `#define` e funções (*Function*). As funções podem ter parâmetros (*param*) e instruções (*stmt*). Uma instrução pode ser uma declaração de variável (*LocalVarDecl*), uma atribuição de variável (*VarAttrib*), uma macro do tipo `#ifdef` ou um retorno de função (*return*). As variáveis podem ser do tipo `int` (*Integer_*) ou `float` (*Float*), e uma função pode retornar `void` (*Void*), `int` (*Integer_*) ou `float` (*Float*).

Código Fonte 2.4: BNF incompleta da linguagem C reconhecida pelo CDOLLY.

```

1      <c_file> -> <defines> | <GlobalVars> | <functions>
2      <defines> -> <defines> | #define <Id>
3      <GlobalVars> -> <GlobalVars> | <GlobalVar>
4      <GlobalVar> -> <BasicType> <name>;
5      <functions> -> <functions> | <function>
6      <function> -> <Type> <name>
7      '(<LocalVar>{' <Stmt> }
```



```

8      <Return> -> "return" <LocalVar> | "return" <GlobalVar>
9      <name> -> a-z<name> | <name><number> | a-z
10     <Id> -> <name>
11     <number> -> <number>0-9 | 0-9
12     <Type> -> <BasicType> | void
13     <BasicType> -> int | float
14     <LocalVar> -> <BasicType> <name>
15     <LocalVarDecl> ->
16         <BasicType> <name> = <number>;
17     <VarAttrib> -> <name> = <number>;
18     <Stmt> -> <VarAttrib> | #ifdef <Id> | #else | #endif |
19         return <name>; | <LocalVarDecl>

```

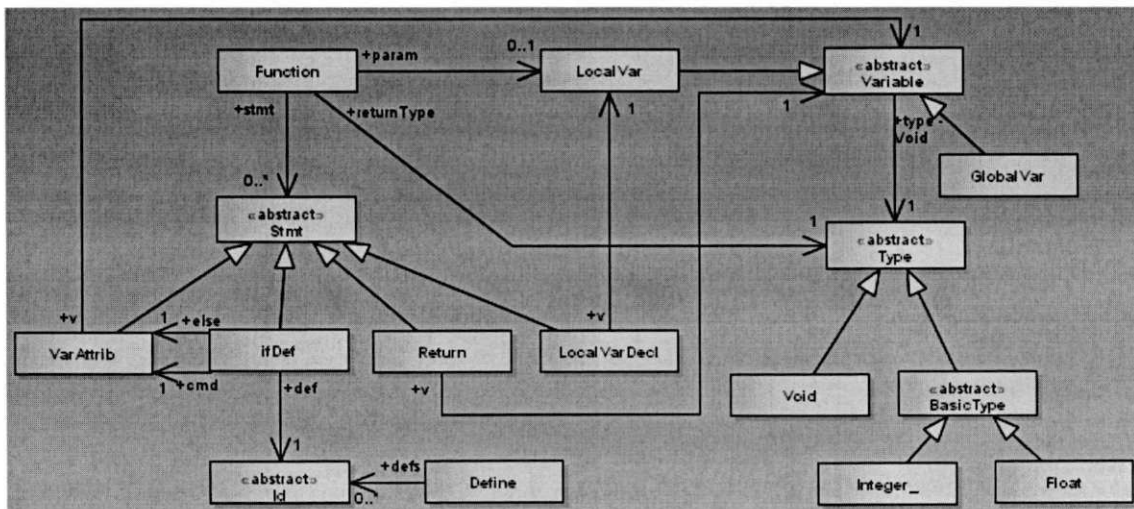


Figura 2.4: Modelo de objetos usado no CDOLLY.

Uma função pode conter um parâmetro, que pode ser do tipo `int` ou `float`. Seu retorno pode ser do tipo `int`, `float` ou `void`. Modelamos o corpo de uma função como sendo uma sequência de instruções (*Stmt*). Uma instrução pode ser uma declaração de variável (*LocalVarDecl*), do tipo `int x;`, uma atribuição de variável (*VarAttrib*), do tipo `int x = 1;`, um retorno (*Return*), do tipo `return x;` ou uma macro `#ifdef`, do tipo `#ifdef A`. A macro `#ifdef` contém dois atributos (*cmd*, *else*), representando, respectivamente, a instrução que deve ser executada caso a condição do `#ifdef` seja satisfeita (*cmd*), e a instrução que deve ser executada caso contrário (*else*). As instruções da macro `#ifdef` sempre são atribuições de variáveis no modelo.

Uma variável pode ser local (*LocalVar*) ou global (*GlobalVar*), podendo ser do tipo *Integer_* ou *Float*. O tipo *Void* é referenciado apenas por funções que não retornam valor. Os únicos elementos fora de uma função são as variáveis globais e macro `#define`. A macro `#define` referencia um *Id*, o mesmo *Id* referenciado por um `#ifdef`.

Um exemplo de todos os elementos suportados pelo subconjunto apresentado pode ser visto no Código Fonte 2.5. Neste arquivo C, é definido o *id* (linha 1), uma variável global do tipo `int` (linha 2) e uma função que recebe como parâmetro um `float` e retorna um número inteiro (linha 3). Nas instruções da função, uma variável local é declarada e o valor -1 é atribuído à variável (linha 4). Um `#ifdef` usa o *id* (linha 5) e atribui o valor 2 à variável local `localVar` (linha 6). Por fim, a função retorna o valor de `localVar` (linha 7).

Código Fonte 2.5: Exemplo de código C gerado pelo CDOLLY.

```
1 #define id
2 int globalVar = 3;
3 int func(float a){
4     float localVar = -1;
5     #ifdef id
6         localVar = 2;
7     #endif
8     return localVar;
9 }
```

2.3.2 Regras de Boa Formação

Toda linguagem de programação possui regras de boa formação que indicam se um programa é válido, como por exemplo *só se deve atribuir valor a uma variável após sua declaração*. Visando gerar programas válidos no CDOLLY, especificamos algumas regras de boa formação.

Em Alloy, para se definir tais regras é preciso especificar fatos. Um exemplo de um fato pode ser visto no Código Fonte 2.6, onde especificamos que se uma função não retorna `void`, deve haver um retorno ao final (linhas 2 e 3).

Código Fonte 2.6: Regra de boa formação. Se função não for void, há retorno ao final.

```

1 fact noVoidFunctionJustOneReturn{
2     all f:Function |
3         f.returnType != Void => one f.stmt.elems & Return
4 }

```

Outro exemplo de fato pode ser visto no Código Fonte 2.7, onde especificamos que uma variável que recebe uma atribuição ou foi passada como parâmetro de uma função ou foi declarada dentro da própria função (linhas 2 a 6).

Código Fonte 2.7: Regra de boa formação. Variável de uma atribuição numa função ou é um parâmetro ou foi declarada na função.

```

1 fact varDeclarationInParameterOrInFunction{
2     all f:Function |
3         (all st1: f.stmt.elems | st1 in VarAttrib =>
4             (some st2: f.stmt.elems |
5                 st2 in LocalVarDecl and st2.(LocalVarDecl<:v) = st1
6                     .(VarAttrib<:v) and f.stmt.idxOf [st2] < f.stmt.
7                     idxOf [st1] ) or
8                 st1.(VarAttrib<:v) = f.param.v)
9 }

```

Especificamos um total de dez regras de boa formação que podem ser vistas na Tabela 2.1. Entretanto, não especificamos todas as regras possíveis. Por exemplo, o Código Fonte 2.8 gerado pelo CDolly antes de especificarmos as regras de boa formação não compila, pois a função `function` faz referência à variável `localVar` (linha 7) que foi declarada dentro da função `func` (linha 3). Essa regra, particularmente, é simples de ser especificada. Porém, regras tais como recursão não o são, pois Alloy não dá suporte direto.

Regras de Boa formação
Variável que recebe atribuição foi declarada na função ou é parâmetro
Variável local declarada não pode ter mesmo nome de parâmetro
Se função for void não há retorno
Se função não for void há retorno ao final
Só há uma instrução return em função não void
Função não void tem retorno do tipo da função
Todo id de um ifDef foi declarado por um Define
Se há atribuição de variável num ifDef, variável foi declarada no parâmetro ou no corpo da função
Não há declaração de variável com mesmo nome de parâmetro
Não há duas declarações de variáveis com mesmo nome

Tabela 2.1: Regras de boa formação de programas C implementadas no CDOLLY.

Código Fonte 2.8: Exemplo de código gerado pelo CDOLLY que não compila, pois falta regra de boa formação.

```

1 void func(){
2   float localVar = -1;
3   int localVar = 1;
4 }
5 int function(){
6   float localVar2 = 2;
7   return localVar;
8 }

```

Além das regras de boa formação, especificamos algumas otimizações para guiar a geração dos programas. Por exemplo, no Código Fonte 2.9, especificamos que a quantidade de instruções dentro de uma função será no máximo 5 (linha 2), e que em toda instância gerada tem pelo menos uma função com instruções (linha 3). Além disso, especificamos que não há instrução duplicada no corpo de uma função (linha 4).

Código Fonte 2.9: Otimizações especificadas no CDOLLY.

```

1 pred optimization [] {
2   all f: Function | #f.stmt < 5
3   some f: Function | #f.stmt > 0
4   all f: Function | not f.stmt.hasDups
5   ...
6 }
```

2.4 Geração de Programas C

Uma vez especificado o modelo, executamos o Alloy Analyzer para encontrar as soluções (instâncias) para o modelo a partir de um predicado e passamos o escopo para o qual queremos gerar (Código Fonte 2.10). Para cada instância encontrada, o CDOLLY traduz seus elementos para a linguagem C.

Código Fonte 2.10: Execução do predicado *show* com escopo 2.

```

1 pred show [] { }
2 run show for 2
```

O CDOLLY gera um arquivo C para cada instância gerada pelo Alloy Analyzer. Cada elemento de Alloy na árvore da Figura 2.5 (a) é traduzido para um elemento que pode ser interpretado em C. Para isso, basta percorrer a árvore de elementos e interpretar seus tipos, fazendo a correlação com C. Por exemplo, ao se encontrar um elemento do tipo *Function*, o CDOLLY procura por seu tipo de retorno (*return*), seus parâmetros (*param*) e suas instruções (*stmt*) para gerar um programa C que seja válido. Quando o CDOLLY encontra uma declaração de uma variável *b* (*vardecl*) do tipo *Integer_*, ele traduz para: `int b;`. O programa C gerado a partir da árvore (Figura 2.5 (a)) encontra-se na Figura 2.5 (b).

Para atribuições de variáveis, usamos a estratégia de escolher valores aleatórios em um conjunto de valores predefinidos. Por exemplo, para atribuir um valor inteiro (*Integer_*) a uma variável, o CDOLLY escolhe aleatoriamente um valor de uma lista: {-1, 0, 1, 2, 4, 10, 100}. Esses valores foram escolhidos sem critério específico, e essa estratégia tem o intuito de aumentar a aleatoriedade na geração dos programas.

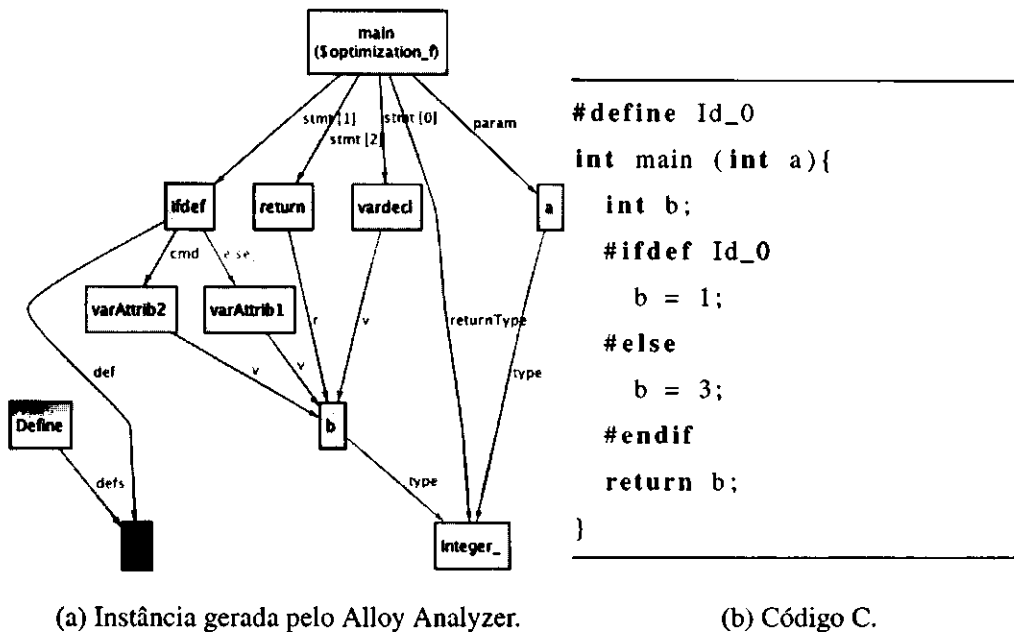


Figura 2.5: Tradução de Alloy para C.

2.5 Geração de Programas Específicos

Para gerar programas com determinadas características no CDOLLY, é possível acrescentar restrições (*constraints*) ao modelo. Por exemplo, o Código Fonte 2.11 restringe as soluções do modelo para programas que tenham funções (linha 1) com duas instruções (linha 8) que recebem parâmetros (linha 5) e retornem `int` (linha 6). Na linha 10 especificamos o predicado para escopo 2. Um exemplo de código C gerado a partir dessa restrição encontra-se no Código Fonte 2.12.

Código Fonte 2.11: Predicado para restringir as soluções do modelo.

```

1 one sig main extends Function {}
2 one sig return extends Return {}
3 one sig a extends LocalVar {}
4 pred show[] {
5   main.param = a
6   a.type=Integer_
7   main.stmt.last = return
8   #main.stmt = 2
9 }
10 run show for 2

```

Código Fonte 2.12: Exemplo de programa gerado pelas restrições de Alloy.

```

#define id
int main(int a){
    int localVar = 0;
    return a;
}
void function(int a){
    int localVar = 4;
    localVar = 2;
}

```

2.6 Discussão

O escopo delimita o espaço de estados no qual o Alloy Analyzer buscará soluções. Como exemplo do rápido crescimento do espaço de estados, executamos o metamodelo C que definimos com escopo 2 e foram gerados 8.856 programas, com taxa de compilação de 73% em 2 minutos (sem contar o tempo de compilação). Com escopo 3, foram gerados mais de 2.000.000 de programas em 7.5 horas² (Tabela 2.2).

Essa explosão de estados é facilmente explicada no CDOLLY, pois além das combinações de valores para assinaturas e relações do metamodelo C, é necessário testar todas as possíveis combinações de instruções (*statements*) dentro do corpo das funções. Essa é uma das diferenças do CDOLLY para o JDOLLY [41], já que este último gera métodos apenas com uma instrução.

2.7 Considerações Finais

Neste capítulo, além de explicarmos alguns elementos da linguagem Alloy, apresentamos CDOLLY, um gerador de programas C baseado em Alloy. A partir de um metamodelo C es-

²Paramos de executar com escopo 3 após cinco dias ininterruptos. A taxa de compilação não foi calculada para esse escopo, pois tal métrica é computada ao final da execução.

Escopo	#Programas Gerados	Compilação (%)	Tempo de Geração dos Programas (h)	Tempo Total Incluindo Compilação (h)
2	8.856	73	0.02	0.32
3	>2000000	-	7.5	120

Tabela 2.2: Escopo 2 e 3 do metamodelo C.

pecificado em Alloy, e para cada instância gerada deste modelo, o CDOLLY gera programas em C. O CDOLLY permite que seja especificado o escopo, que representa a quantidade de elementos gerados para cada entidade do modelo, e restrições, que são regras para formação dos elementos.

Capítulo 3

Uma Técnica para Testar

Implementações de Refatoramentos de Programas C

Neste capítulo apresentamos uma técnica para testar automaticamente implementações de refatoramentos de programas C. Na Seção 3.1 apresentamos uma visão geral da técnica. Em seguida (Seções 3.2 a 3.6) explicamos cada fase da técnica. Na Seção 3.7 mostramos um exemplo de uso da técnica e finalmente na Seção 3.8 apresentamos as considerações finais.

3.1 Visão Geral

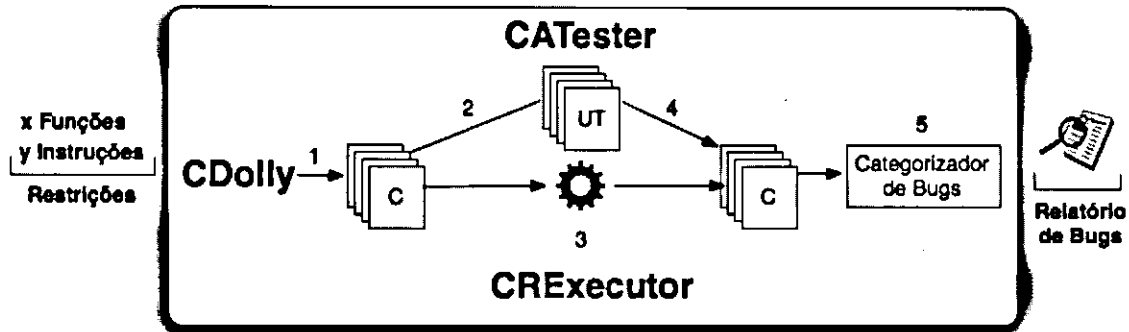
Nossa técnica tem o intuito de testar implementações de refatoramentos de programas C para detectar erros de compilação e mudanças comportamentais. Para testar um determinado tipo de refatoramento, nossa técnica 1) gera programas com características específicas utilizando o CDOLLY; 2) gera testes de unidade com o C Automatic Tester (CATESTER) ¹ para cada programa compilável gerado pelo CDOLLY; 3) aplica o refatoramento na ferramenta a ser testada através do C Refactoring Executor (CREXECUTOR ²) em cada programa; 4) executa os testes de regressão nos programas refatorados e 5) analisa os resultados dos testes de

¹O CATESTER pode ser baixado em <https://bitbucket.org/gugawag/catester>.

²Para aplicar o refatoramento, o CREXECUTOR utiliza a API de refatoramentos da ferramenta a ser testada. Ele pode ser baixado em <https://bitbucket.org/gugawag/crefactoringexecutor>

regressão e os categoriza em erros de compilação ou mudança comportamental (Figura 3.1).

Figura 3.1: Técnica para testar implementações de refatoramento C.



Para ilustrar o funcionamento desta técnica, vamos mostrar um exemplo em que se busca testar o refatoramento *Rename Function*. Primeiramente, especificamos em Alloy as características do programa a ser testado. Precisamos gerar programas que tenham pelo menos uma função com nome *func* (nome a ser refatorado). Para isso, especificamos as restrições do Código Fonte 3.1, onde um programa tem que ter pelo menos uma função com nome *func* (linha 1), e executamos o predicado (linha 2) com escopo 2 para os elementos e escopo 3 para as instruções das funções (linha 3).

Executamos o CDOLLY, passando as restrições em Alloy, e geramos todos os programas do modelo (Fase 1). Tentamos compilar cada programa gerado e apenas os que compilam passam para a Fase 2 (ex.: Código Fonte 3.2).

Na Fase 2, para cada programa, geramos testes de unidade com o CATESTER. Os testes unitários são gerados a partir da execução do programa com diferentes entradas, computando as diferentes saídas. Por exemplo, ao executar o programa do Código Fonte 3.2 passando 3 como parâmetro da função *func*, o valor 10 é retornado. O CATESTER gera um teste de unidade colocando como asserção `asserttrue(10==retint2);` (linha 9 do Código Fonte 3.3).

Código Fonte 3.1: Predicado para restringir as instâncias para refatoramento *Rename Function*.

```

1 one sig func extends Function {}
2 pred show[] {}
3 run show for 2 but 3 Stmt

```

Código Fonte 3.2: Programa gerado pelo CDOLLY para refatoramento *Rename Function*.

```

int func(int param){
    int localVar = 10;
    return localVar;
}

```

Código Fonte 3.3: Teste de unidade do Código Fonte 3.2 gerado pelo CATER.

```

1 ...
2 void test_1_func(void){
3     int int1 = 3;
4     int retint2 = func(int1);
5     //teste de igualdade
6     int ret_final = func(int1);
7     asserttrue(retint2 == ret_final
8                 );
9     //teste de regressao
10    asserttrue(10==retint2);
11 }

```

Na Fase 3 passamos para o CREXECUTOR os programas gerados na Fase 1 para que se aplique o refatoramento a ser testado. O CREXECUTOR recebe todos os programas e para cada um altera o nome da função *func* para um novo nome (vide Seção 3.4.1 - Entrada para o Refatoramento) executando a API de refatoramentos da ferramenta de desenvolvimento (ex.: Eclipse CDT).

Na Fase 4 os testes de regressão são executados nos programas refatorados e os resultados são analisados na Fase 5. Nesta última fase, são analisados todos os erros de compilação (Seção 3.6.1) e as mudanças comportamentais (Seção 3.6.2) decorrentes do refatoramento, categorizando as falhas em bugs. Na próximas seções detalhamos cada uma dessas fases.

3.2 Fase 1: Geração de Programas

Nesta fase, o objetivo é executar o CDOLLY para gerar todos os programas C. O CDOLLY recebe as restrições do modelo e o escopo dos elementos, executa o modelo e, para cada instância do Alloy Analyzer, gera um programa C. Apenas os programas que compilam são levados em consideração ao final desta fase. Como exemplo, ao executar o CDOLLY com as restrições e o escopo do Código Fonte 3.1, são gerados 19.500 programas, com taxa de compilação de 67%.

3.3 Fase 2: Geração de Testes de Unidade

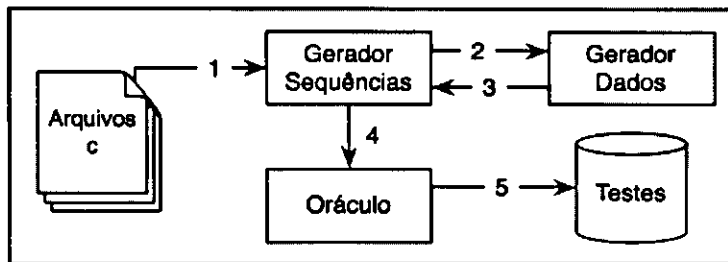
Após a geração dos programas, o objetivo é gerar uma coleção de testes de unidade para todos os programas compiláveis gerados pelo CDOLLY. Para isso, desenvolvemos o CATESTER, uma ferramenta de geração aleatória de testes de unidade para programas escritos em C. O CATESTER se baseia no Randoop [35] e implementa uma abordagem randômica de geração de testes orientada a retroalimentação (*feedback*): o valor de retorno da execução de uma função serve de entrada para execução de outra.

CATESTER recebe como entrada o código fonte de um programa e a estratégia de geração dos testes de unidade (por tempo definido ou por quantidade de testes por função). Ele analisa as funções encontradas no código fonte e gera sequências para testá-las. Possui três módulos: gerador de dados, gerador de sequências e oráculo (Figura 3.2). O gerador de sequências lê as funções do sistema (1), o gerador de dados escolhe aleatoriamente valores de acordo com os tipos dos argumentos das funções (2) e retorna para o gerador de sequências (3); o gerador de sequências gera instruções em C que exercitam as funções (4) e o oráculo define as asserções dos testes (5), sendo gerado ao final uma coleção de testes. Nas próximas seções detalhamos cada um dos componentes do CATESTER.

3.3.1 Geração de Dados

O gerador de dados é responsável por gerar valores que são usados como argumentos de uma função. Ele é necessário pois não se sabe previamente qual o comportamento de um programa, e para exercitá-lo é necessário gerar valores para suas funções. O gerador de da-

Figura 3.2: Componentes do CATESTER.



dos inicia o processo com alguns valores semente (Tabela 3.1), e à medida que sequências são geradas e executadas os valores de retorno das funções são armazenados para retroalimentação.

Além destes valores, o CATESTER, no início de sua execução, armazena todos os literais do programa a ser testado. Por exemplo, se no programa for encontrada a instrução `const int x = 45;`, o valor 45 será armazenado pelo gerador. A escolha do valor a ser utilizado como argumento de uma função é feita aleatoriamente.

Discussão

Gerar bons valores para testar um programa é um dos problemas na área de testes. Esse problema torna-se mais complexo quando não se tem informação do funcionamento de um programa. Uma possível abordagem é levar em consideração todos os possíveis valores de um determinado tipo. Por exemplo, o tipo `int` em C, numa arquitetura de 32 bits, tem-se valores entre -32768 e 32767. Um gerador de dados típico levaria em consideração todos os valores nessa faixa. Apesar dessa abordagem apresentar uma boa cobertura de *statements*, ela tem um alto custo computacional.

Para reduzir esse problema, ao invés de utilizar todos os valores possíveis dos tipos primitivos de C, reduzimos inicialmente a faixa para apenas alguns valores, como pode ser visto na Tabela 3.1. Por exemplo, para o tipo `int` os valores iniciais são `{-1, 0, 1, 2, 3, 100, 999999999999999}`. Esses valores foram definidos ao acaso,³ e são escolhidos aleatoriamente para serem passados como argumento de uma função. À medida que as sequências retornam novos valores, o gerador de dados pode utilizá-los como argumento de uma função. Se, por exemplo, a sequência do Código Fonte 3.5 retornar o valor 20, esse valor será arma-

³Especificamente o valor 999999999999999 foi escolhido para aumentar a chance de se detectar bugs do tipo índice fora da faixa de um array.

Tabela 3.1: Valores inicialmente utilizados como argumentos de funções no CATESTER.

Tipo	Valores
int	{-1, 0, 1, 2, 3, 100, 999999999999999}
double	{-10.4, -1, 0, 1.1, 2, 3.0, 100.5, 999999999999999}
float	{-1.0F, 0.0F, 1.0F, 1.10F, 10.54F, 100.67F, 999999999999999}
short	{-32768, -1, 0, 1, 10, 100, 32767}
long	{-1L, 0L, 1L, 10L, 110L, 1054L, 100989483967L}
char	{a, b, c, Z, F, 4, 5}

zenado na lista de inteiros para ser utilizado pelo gerador de dados em novas sequências. Desta forma, usamos argumentos para as funções com valores mais próximos aos esperados pelo programa testado, além de aumentar a probabilidade de diferentes partes do programa serem exercitadas.

Apesar do CATESTER reconhecer todos os tipos apresentados na Tabela 3.1, neste trabalho levamos em consideração apenas os tipos `int` e `float`.

3.3.2 Geração de Sequência

O gerador de sequências tem o intuito de exercitar trechos do programa. Para isso, são geradas sequências de instruções necessárias para executar uma função de um programa C. Os dados gerados pelo gerador de dados são passados às sequências. Para gerar testes de unidade de um programa, o CATESTER, inicialmente, lê todos os arquivos de um programa C (arquivos `.h` e `.c`) e armazena todas as declarações de funções. Para cada função, são geradas diferentes sequências a fim de exercitar o programa e descobrir automaticamente seu comportamento.

Uma sequência é formada por instruções, e uma instrução pode ser uma declaração de variável, uma atribuição de valor ou uma execução de função. Cada sequência tem um tipo e um valor de retorno: o tipo da sequência é o tipo de retorno da função que está sendo executada e o valor de retorno representa o valor retornado pela última instrução da sequência (chamada de função). O tipo da sequência e a sequência em si são passados para o gerador

de dados para serem usados em novas sequências a fim de permitir a retroalimentação dos testes. Sendo assim, comumente sequências estendem outras.

Considere as funções $f1$ e $f2$ do Código Fonte 3.4, cujas sequências estão representadas nos Códigos Fonte 3.5 e 3.6, respectivamente. No Código Fonte 3.5 é definida uma variável do tipo inteira (linha 1) e passada como parâmetro para $f1$ (linha 2). O tipo do retorno dessa sequência é `int` e o valor será o resultado de $f1(10)$. No Código Fonte 3.6 a função $f2$ (linha 4) tem como parâmetros os tipos `int` e `double`. Como existe uma sequência com tipo de retorno `int`, o CATESTER dá prioridade à extensão desta sequência (linha 2 do Código Fonte 3.5) e seu valor de retorno é passado como argumento para a função $f2$ (linha 4 do Código Fonte 3.6).

Código Fonte 3.4: Exemplos de funções.

```
1 int f1(int x);
2 double f2(int x, double y);
```

Código Fonte 3.5: Sequência da função $f1$.

```
1 int x = 10;
2 int ret1 = f1(x);
```

Código Fonte 3.6: Sequência da função $f2$ (estendendo sequência do Código Fonte 3.5).

```
1 int x = 10;
2 int ret1 = f1(x);
3 double w = 1.1;
4 double ret2 = f2(ret1, w);
```

Como pode ser visto nestes exemplos, as sequências geradas pelo CATESTER não contém função `main`. Isso deve por dois motivos. O primeiro diz respeito à técnica de geração de testes orientada à retroalimentação (*feedback*). Como uma sequência pode ser estendida, se houvesse um `main` na sequência original daria conflito de nomes de função na sequência estendida. O segundo motivo diz respeito à geração dos testes de unidade. Um teste de unidade não pode interferir no resultado de outro, por definição. Por isso, o CATESTER gera um programa C para cada teste de unidade. Se a sequência gerada fosse envolvida numa função `main`, daria igualmente conflito de nomes de função. Sendo assim, apenas no

momento de executar a sequência e os testes de unidade o CATESTER envolve cada sequência gerada em uma função `main`, compila a sequência e a executa para armazenar o valor de retorno.

Discussão

A extensão de sequências, inspirada na ferramenta Randoop, é importante por dois motivos. Primeiro, porque ela proporciona a diminuição da quantidade de dados inválidos passados para as funções, já que o valor final de uma sequência é gerado pelo próprio programa testado. Segundo, porque possibilita a detecção de bugs mais complexos que ocorrem quando diferentes partes do programa são exercitados. Apesar do CATESTER também ser uma ferramenta voltada à detecção de bugs em sistemas em C, neste trabalho a utilizamos apenas como gerador de testes de unidade.

3.3.3 Oráculo

Uma vez que os testes estejam definidos, é necessário saber quais os resultados esperados. Um oráculo de testes é utilizado para este fim. Normalmente, um oráculo é formado por informações de especialistas do programa a ser testado. Na falta destes é necessário criar um mecanismo para obter a saída esperada do sistema.

O oráculo do CATESTER leva em consideração três regras básicas: 1) que uma função quando executada duas vezes com os mesmos argumentos retorna o mesmo valor (identidade),⁴ 2) que este valor pode ser usado como asserção de um teste e 3) que um programa não pode lançar sinal. Para garantirmos a regra 1) executamos duas vezes uma função, com os mesmos argumentos, e comparamos seus retornos, como pode ser visto no exemplo do Código Fonte 3.7 (linha 3). Para garantirmos a regra 2), armazenamos todos os valores de retorno de funções e definimos asserções a partir destes, como pode ser visto no exemplo do Código Fonte 3.8. Para garantir a regra 3), cadastramos nos testes de unidade todos os sinais de C possíveis de serem lançados. Quando algum sinal é lançado durante a execução dos testes de unidade de um programa, capturamos esse sinal e armazenamos o programa para análise.

Código Fonte 3.7: Oráculo: exemplo da Regra 1.

⁴Levando em consideração apenas programas determinísticos.

```

1 int ret1 = f1(10);
2 int ret2 = f1(10);
3 asserttrue(ret1==ret2);

```

Código Fonte 3.8: Oráculo: exemplo da Regra 2.

```

1 int ret1 = f1(10);
2 ...
3 asserttrue(f1(10) == 5);

```

3.3.4 Expressividade de C no CATester

Atualmente o CATESTER reconhece apenas um subconjunto da linguagem C. Em resumo, reconhece funções que retornam `int`, `float` ou `void` e que recebem como parâmetros todos os tipos primitivos, estruturas (`struct`) contendo tipos primitivos, arrays de tipos primitivos e ponteiros para tipos primitivos. A BNF incompleta da linguagem C reconhecida pelo CATESTER pode ser vista no Código 3.9.

Código Fonte 3.9: BNF incompleta da linguagem C reconhecida pelo CATESTER.

```

<função> -> <retorno-função> <nome>
          '(' <declaração-parâmetros> ')' { <corpo-função> }
<retorno-função> -> int
<nome> -> a-z<nome> | <número><nome> |
          <nome><número> | a-z
<número> -> <número>0-9 | 0-9
<tipo-variável> -> int | double | float |
                 char | byte | struct
<declaração-parâmetros> ->
          <declaração-parâmetro> |
          <declaração-parâmetros> , <declaração-parâmetro>
<declaração-parâmetro> -> <tipo-variável> <nome> |
          <declaração-ponteiro> | <declaração-array>
<declaração-array> -> <tipo-variável> <nome> []
<declaração-ponteiro> -> <tipo-variável> * <nome>
<declaração-variável> -> <declaração-parâmetro>; |
          <declaração-parâmetro> = <nome>; |
          <declaração-parâmetro> = <numero>;

```

```
<corpo-função> -> <instruções>  
<instruções> -> <declaração-variável><instruções> | ()  
<instrução> -> <declaração-variável>
```

3.3.5 Exemplo

Mostraremos um exemplo de uso do CATESTER através da geração de testes de unidade do arquivo C mostrado no Código Fonte 3.10. Primeiramente, o CATESTER realiza uma busca em todo o código fonte por assinaturas de funções. O CATESTER encontra a função *f* (linha 1), e para cada argumento, escolhe aleatoriamente a partir do gerador de dados os valores a serem passados para a função, de acordo com o tipo do argumento. No caso do exemplo, como a função *f* tem parâmetro do tipo *int*, o gerador de dados pega aleatoriamente um valor na lista de valores inteiros armazenados (valor 10 na linha 2 do Código Fonte 3.11). O gerador de sequência gera a sequência (Código Fonte 3.11), compila e a executa. Um dos oráculos do CATESTER é que uma função deve retornar o mesmo valor para duas chamadas distintas a essa função quando passados os mesmos parâmetros. Por isso, o CATESTER define o teste de igualdade (linha 6 do Código Fonte 3.12) após executar a função *f* duas vezes (linhas 3 e 4). Após o teste de igualdade, o CATESTER define o teste em si (linha 8 do Código Fonte 3.12). Na linha 8 é feita a igualdade entre o retorno da função e o valor 20, valor de retorno da sequência gerada. A sequência é armazenada na lista do tipo *int* do gerador de dados para potencialmente ser usada em próximas sequências.

Código Fonte 3.10: Programa em C.	Código Fonte 3.11: Sequência gerada pelo CATESTER.	Código Fonte 3.12: Teste de unidade definido pelo CATESTER.
<pre> 1 int f(int x){ 2 return 2*x; 3 } </pre>	<pre> 1 int seq1(){ 2 int var1 = 10; 3 int ret1 = f(var1); 4 return ret1; 5 } </pre>	<pre> 1 void test_1_seq1(){ 2 int var1 = 10; 3 int ret1 = f(var1); 4 int ret2 = f(var1); 5 //teste de igualdade 6 asserttrue(ret1 == 7 ret2); 7 //teste de retorno 8 asserttrue(ret1 == 9 20); 9 } </pre>

3.4 Fase 3: Aplicação de Refatoramento

Nesta fase o CREXECUTOR executa o refatoramento nos programas gerados na Fase 1 utilizando a API de refatoramento da ferramenta de desenvolvimento a ser testada. O CREXECUTOR recebe como entrada os programas a serem refatorados, o tipo de refatoramento que se deseja aplicar e os valores de entrada (ex.: novo nome de função para refatoramento *Rename Function*). O CREXECUTOR utiliza a API de refatoramento da ferramenta que se deseja testar para checar as pré-condições do refatoramento e para aplicar o refatoramento em si. Após aplicação do refatoramento, o CREXECUTOR separa os programas que compilam dos que deixaram de compilar. Os programas que compilam serão usados na próxima fase. Os programas que não compilam serão analisados na fase 5 apresentada na Seção 3.6.1.

Os testes de unidade gerados na fase anterior não fazem parte do refatoramento. Esperamos analisar possíveis mudanças comportamentais no código refatorado testando com a coleção de testes original, já que um programa refatorado precisa manter o comportamento do original.

Tabela 3.2: Tipos e valores passados aos refatoramentos.

Tipo	Valor
Palavra Reservada	int
Literal	0
Operador *	*[oldName]
Operador &	&[oldName]
Nome com Espaço	a b
Nome Existente de Função	Function_0
Nome Existente de Variável Global	GlobalVar_0
Identificador de Define Existente	Id_0
Novo Nome	new

3.4.1 Entradas para o Refatoramento

Cada tipo de refatoramento, além de um programa, precisa receber como entrada um novo valor do elemento a ser refatorado. Por exemplo, ao se extrair uma constante, é necessário informar qual o nome da constante a ser definida.

Escolhemos alguns tipos de entradas para os refatoramentos a partir de bugs identificados anteriormente para Java/C [40; 41; 14], a partir de nosso conhecimento como desenvolvedores e definimos as classes de partição. Esses tipos, bem como exemplo de valores, podem ser vistos na Tabela 3.2. Para cada programa a ser refatorado, o CREXECUTOR escolhe aleatoriamente uma dessas entradas e passa para a API de refatoramento.

Foram escolhidos esses tipos de entrada para que fossem testados os refatoramentos e para dar diretrizes de quais condições são checadas pelas implementações dos refatoramentos. Pretendemos posteriormente usar commits de diferentes projetos para entender os tipos de transformações usualmente aplicados pelos desenvolvedores e incrementar os tipos de entrada.

3.5 Fase 4: Execução dos Testes de Regressão

Nesta fase, o objetivo é executar os testes de regressão gerados na Fase 2 nos programas refatorados na Fase 3. Se os testes de regressão de um programa falharem, o programa e os testes serão armazenados para serem analisados na Fase 5. Um teste de regressão pode falhar por mudança no resultado dos testes, ou por lançamento de um sinal em C.

Um sinal em C pode ser lançado quando ocorre uma condição excepcional ao programa, como por exemplo uma divisão por zero, ou quando ocorre um evento externo ao programa, como por exemplo um click do mouse a ser tratado pelo programa. O CATESTER, ao gerar os testes de unidade, cadastra todos os possíveis sinais a serem lançados em C e, ao executar os testes de regressão, os sinais lançados pelo programa são capturados. Tanto mudança comportamental do programa por lançamento de sinal quanto por mudança no resultado dos testes serão analisados na Seção 3.6.2.

3.6 Fase 5: Categorização dos Bugs

Nesta quinta e última fase, categorizamos os bugs causados pelo refatoramento em dois tipos: erros de compilação ou mudança comportamental. Para os erros de compilação, usamos o classificador automático proposto por Jagannath [21], que se baseia no agrupamento das mensagens de erro. Para os erros de mudança comportamental, usamos a noção de equivalência proposta por Opdyke: todos os programas que são refatorados, mas que apresentam diferenças de resultados entre o original e o refatorado, são analisados para categorizarmos os bugs.

3.6.1 Erros de Compilação

Analisar manualmente todos os erros de compilação de cada programa refatorado é custoso. Nossa técnica gera milhares de programas, e uma análise manual seria exaustiva e propensa a erro. Normalmente é gerada uma grande quantidade de falhas para um bug encontrado.

Por isso, usamos o classificador automático proposto por Jagannath [21], que consiste em agrupar as mensagens de erro, classificando cada grupo em um bug. Adaptamos esse classificador para, além da mensagem do erro de compilação, levarmos em consideração o tipo

de entrada (Tabela 3.2) do refatoramento para definir os grupos. Sendo assim, cada chave **mensagem;tipo_entrada** é caracterizada como um novo bug. Por exemplo, ao se aplicar o refatoramento *Rename Local Variable*, se levássemos em consideração apenas a mensagem de erro para agrupar os bugs, a mensagem *expected identifier or '('* agruparia tanto o bug devido ao novo nome conter endereço de ponteiro (&) quanto o bug devido ao novo nome ser um literal (0). O tipo de entrada foi acrescentado na chave do agrupamento pois acreditamos que os tipos de entrada são suficientemente diferentes para que os refatoramentos implementem diferentes pré-condições.

Como exemplo de categorização de bugs de erro de compilação, ao tentar refatorar o nome da função `func` para a palavra reservada `int` (linha 1 do Código Fonte 3.13), o Eclipse CDT aplicou o refatoramento (linha 1 do Código Fonte 3.14). Porém, ao compilar o programa refatorado, foi lançada a seguinte mensagem de erro: *cannot combine with previous 'int' declaration specifier*. Buscando generalizar o bug, removemos da mensagem toda expressão que seja específica ('int'), tornando a mensagem genérica: *cannot combine with previous '*' declaration specifier*. Sendo assim, se ao compilar um programa refatorado cuja função retorne `float` for lançada a mensagem *cannot combine with previous 'float' declaration specifier*, agrupamos ambos os casos e computamos apenas um bug: *cannot combine with previous '*' declaration specifier;PALAVRA_RESERVADA*.

Código Fonte 3.13: Programa gerado pelo CDOLLY para refatoramento *Rename Function*.

```
1 int func(int param){
2     return param;
3 }
```

Código Fonte 3.14: Código refatorado pelo CREXECUTOR, porém não compila.

```
1 int int (int param){
2     return param;
3 }
```

3.6.2 Mudança Comportamental

Para analisar mudanças comportamentais, utilizamos uma abordagem semi-automática. Primeiramente, dividimos as falhas em dois tipos: falhas que lançam sinais e falhas que não lançam. As falhas que lançam sinais são agrupadas por tipo de sinal (ex.: *segmentation*

fault) e cada grupo é considerado um bug. Já as falhas que não lançam sinal são inspecionadas manualmente.

Agrupamos as falhas que lançam sinais por tipo de sinal da mesma forma que agrupamos as mensagens de erro de compilação. Entendemos que um sinal lançado ao executar um programa refatorado seja similar a uma mensagem devolvida pelo compilador.

Como exemplo, o CREXECUTOR refatorou o Código Fonte 3.15, renomeando a variável global `global` (linha 1) para `*global` (linha 1 do Código Fonte 3.16). Ao executar o código refatorado, foi lançado o sinal *segmentation fault*, pois a variável `global` passou a ser um ponteiro para inteiro, porém uma posição de memória não foi alocada para o ponteiro.

Código Fonte 3.15: Programa gerado pelo CDOLLY para refatoramento *Rename Global Variable*.

```
1 int global = 100;
2 int func(){
3     float localVar = 100;
4     return global;
5 }
```

Código Fonte 3.16: Código refatorado pelo CREXECUTOR. Ao executar, lança sinal *segmentation fault*.

```
1 int *global = 100;
2 int func(){
3     float localVar = 100;
4     return *global;
5 }
```

Para agruparmos as falhas que lançam sinais, não acrescentamos o tipo de entrada do refatoramento (ex.: `PALAVRA_RESERVADA`), como fizemos nos erros de compilação, pois acreditamos que os sinais são diferentes o suficiente para serem lançados apenas para tipos específicos de entrada de refatoramento. De fato, ao analisarmos vários refatoramentos aplicados com diferentes entradas, o sinal *segmentation fault* só foi lançado quando houve refatoramento incluindo o operador de ponteiro (*) ao nome refatorado. Inspecionamos manualmente as falhas que não lançam sinal, por não haver critério objetivo de agrupamento das falhas.

3.7 Exemplo de Aplicação da Técnica

Nesta seção iremos mostrar um exemplo completo de uso da técnica proposta. Testaremos o refatoramento *Extract Local Variable* do Eclipse CDT.

Geração de Programas

Primeiramente, especificamos em Alloy as características dos programas a serem gerados pelo CDOLLY para o refatoramento *Extract Local Variable*. No Código Fonte 3.17 especificamos que deve ser gerada uma declaração de variável local com nome *varDecl* (linha 2), uma função com nome *func* (linha 3) e que a variável deve fazer parte das instruções da função (linha 5). Especificamos escopo 3 para as instruções da função e 2 para os demais elementos (linha 8). Executamos o CDOLLY e foram gerados 10.500 programas em 5.6h, com taxa de compilação de 74%. Um exemplo de programa gerado pode ser visto no Código Fonte 3.18.

Código Fonte 3.17: Especificando programas com características específicas para o refatoramento *Extract Local Variable*.

```
1 open core
2 one sig varDecl extends LocalVarDecl{}
3 one sig func extends Function{}
4 fact {
5   varDecl in func.stmt.elems
6 }
7 pred show[] {}
8 run show for 2 but 3 Stmt
```

Geração de Testes de Unidade

Com os programas gerados, executamos o CATESTER para gerar os testes de unidade de todos os programas que compilaram. Para o Código Fonte 3.18, foram gerados os testes de unidade *test_1_func* e *test_2_func2* (Código Fonte 3.19). Para a função *func*, foi gerado o *test_1_func*, que passa como parâmetro o valor 100 (linha 3), executa uma segunda vez a função *func* (linha 5) para realizar o teste de igualdade e compara com o resultado anterior (linha 6), e finalmente define o teste de regressão (linha 8).

Código Fonte 3.18: Programa gerado pelo CDOLLY para refatoramento *Extract Local Variable*.

```

1 int func(float param){
2   int varDecl = 100;
3   return varDecl;
4 }
5 int func2(){
6   int varDecl = 4;
7   return varDecl;
8 }

```

Código Fonte 3.19: Teste gerado pelo CATester do Código Fonte 3.18.

```

1 void test_1_func(void){
2   float float1 = 100F;
3   int retint1 = func(float1);
4   //teste de igualdade
5   int ret_final = func(float1);
6   asserttrue(retint1==ret_final);
7   //teste de regressao
8   asserttrue(100==retint1);
9 }
10 void test_2_func2(void){
11   int retint1 = func2();
12   //teste de igualdade
13   int ret_final = func2();
14   asserttrue(retint1==ret_final);
15   //teste de regressao
16   asserttrue(4==retint1);
17 }

```

Aplicação do Refatoramento

Após gerar os testes de unidade, aplicamos o refatoramento *Extract Local Variable* em cada programa compilável gerado na Fase 1. Para realizar esse refatoramento, é necessário informar à API do *Extract Local Variable* a literal a ser extraída, além do nome da nova variável. O CREXECUTOR procura no Código Fonte 3.18 a literal que se encontra após a declaração de variável (`int varDecl = 100;`) (linha 2) e escolhe aleatoriamente o nome da variável de acordo com as entradas apresentadas na Tabela 3.2.

Por exemplo, o Código Fonte 3.18 foi refatorado para o Código Fonte 3.20. O valor `int` (linha 2) foi escolhido aleatoriamente como nome da variável, e o Eclipse CDT refatorou o programa, passando a não mais compilar. O refatoramento foi aplicado nos 7.812 programas compiláveis, usando aleatoriamente como nome de variável os valores da Tabela 3.2.

Código Fonte 3.20: Programa refatorado por *Extract Local Variable*.

```

1 int func(float param){

```

```
2  int int = 100;
3  return varDecl;
4  }
5  int func2(){
6  int varDecl = 4;
7  return varDecl;
8  }
```

Execução dos Testes de Regressão

Após os refatoramentos terem sido aplicados, executamos os testes de regressão nos códigos refatorados. Dos 7.812 programas testados, 846 apresentaram mudança comportamental por lançamento de sinal.

Categorização dos Bugs

Por fim, analisamos os erros de compilação após a aplicação do refatoramento e as mudanças comportamentais após a execução dos testes de regressão. Foram computadas 3.883 falhas de compilação. Como exemplo, após ter tentado compilar o Código Fonte 3.20, o compilador lançou a seguinte mensagem: *cannot combine with previous 'int' declaration specifier*. Agrupamos todas as mensagens similares, com todos os tipos de entrada do refatoramento, e chegamos aos seguintes grupos:

- *expected identifier or '(';OPERADOR_**
- *expected '(' at end of declaration;NOME_ESPAÇO*
- *expected identifier or '(';IDENTIFICADOR_DEFINE_EXISTENTE*
- *expected identifier or '(';LITERAL*
- *cannot combine with previous '*';PALAVRA_RESERVADA*

Desta forma, classificamos as falhas em 5 bugs devido a erros de compilação. Foram detectadas 846 falhas de mudança comportamental devido a lançamento de sinal. Como exemplo, o Código Fonte 3.21 foi refatorado para o Código Fonte 3.22. Foi gerado o teste de regressão do Código 3.23 e após sua execução foi lançado o sinal 11 (*segmentation fault*), devido a atribuir a um ponteiro um literal (linha 2 do Código Fonte 3.22). Classificamos essas falhas em um bug de mudança comportamental.

Código Fonte 3.21: Programa gerado pelo CDOLLY para refatoramento *Extract*

Local Variable.

```
int func(int param){
    int varDecl = 100;
    return param;
}
```

Código Fonte 3.22: Código Fonte 3.21 após refatoramento.

```
1 int func(int param){
2     int *old = 100;
3     int varDecl = *old;
4     return param;
5 }
```

Código Fonte 3.23: Teste de unidade do Código Fonte 3.21.

```
...
void test_1_func(void){
    int int1 = 9999999999999999;
    int retint1 = func(int1);
    //teste de igualdade
    int ret_final = func(int1);
    asserttrue(retint1 == ret_final);
    //teste de regressao
    asserttrue(-1530494977 == retint1);
}
...
```

3.8 Considerações Finais

Neste capítulo apresentamos nossa abordagem para testar implementações de refatoramentos de programas C composta por 5 fases: geração de programas, geração de testes de unidade, aplicação de refatoramento, execução dos testes de regressão e categorização dos bugs. Explicamos cada uma dessas fases e apresentamos um exemplo que demonstra que a técnica é capaz de detectar bugs por erro de compilação e por mudança comportamental.

Capítulo 4

Avaliação

Neste capítulo, apresentamos uma avaliação¹ de nossa técnica aplicada em diferentes tipos de refatoramento do Eclipse CDT. Apresentamos a definição do experimento (Seção 4.1), onde mostramos seu objetivo; o planejamento (Seção 4.2), onde elencamos as questões e hipóteses do experimento; a operação (Seção 4.3), onde são apresentados os resultados obtidos; a discussão (Seção 4.4), onde interpretamos os resultados do experimento. Apresentamos na Seção 4.5 as respostas aos questionamentos, na Seção 4.6 as ameaças à validade e, finalmente, na Seção 4.7 fazemos as considerações finais.

4.1 Definição

O objetivo deste experimento é analisar nossa técnica de testar implementações de refatoramentos com o propósito de avaliar a detecção de bugs de compilação e mudança comportamental do ponto de vista de desenvolvedores no contexto de implementações de refatoramentos em C. Para alcançar esse objetivo, fizemos as seguintes perguntas:

Q 1 *Nossa abordagem é capaz de detectar bugs de compilação?*

Q 2 *Nossa abordagem é capaz de detectar bugs de mudança comportamental?*

Q 3 *Nossa abordagem é capaz de detectar bugs em implementações de refatoramentos aplicados a corpo de funções?*

¹Os arquivos resultantes do experimento podem ser vistos em <https://bitbucket.org/gugawag/experimento-crexecutor>.

Caso haja resposta positiva às questões de Q1-Q3, faremos a pergunta:

Q 4 *Os bugs detectados ocorrem em outras implementações de refatoramentos?*

Para responder a essa questão, cada bug detectado nas implementações dos refatoramentos do Eclipse CDT será replicado no NetBeans, no Xrefactory e no OpenRefactory.

4.2 Planejamento

Nesta seção apresentamos os fatores e variáveis de resposta do experimento (Seção 4.2.1), a seleção das unidades experimentais (Seção 4.2.2) e a instrumentação do experimento (Seção 4.2.3).

4.2.1 Fatores e Variáveis de Resposta

Fatores de um experimento são as entradas que afetam o resultado final. Definimos como fatores o tipo de refatoramento e o tipo do nome do elemento refatorado. Como níveis:

- tipo de refatoramento: *Rename Parameter, Rename Function, Rename Global Variable Global, Renome Local Variable, Rename #define, Extract Function, Extract Constant e Extract Local Variable;*
- tipo do nome do elemento refatorado: *nome com espaço, palavra reservada, operador '*', operador &, literal, novo nome, nome de variável global existente, nome de função existente e identificador de define existente.*

Como variáveis de resposta do experimento, computamos:

- Tempo total da análise por refatoramento;
- Quantidade de programas gerados;
- Percentual de programas compilados;
- #falhas de compilação;
- #bugs de compilação;

- #falhas de mudança comportamental; e
- #bugs de mudança comportamental.

4.2.2 Seleção das Unidades Experimentais

Nesta seção apresentamos as unidades experimentais escolhidas para o experimento.

Ferramentas de Refatoramento

Definimos como ferramentas de refatoramento o Eclipse CDT [9] (Kepler Service Release 1), NetBeans CND (7.4) [31], Xrefactory (2.0.14) [51] e OpenRefactory (demo de fevereiro/2014) [33; 17]. O Eclipse CDT será a ferramenta a ser usada no experimento, e com o NetBeans, Xrefactory e OpenRefactory faremos testes manuais a partir dos bugs encontrados no Eclipse. Escolhemos o Eclipse CDT como ferramenta a ser analisada porque utiliza uma plataforma consolidada no desenvolvimento de sistemas, é *opensource* e contém 8 implementações de refatoramentos C, quantidade maior de implementações do que de outras ferramentas similares. Além disso, o Eclipse CDT possui uma comunidade atuante: até fevereiro de 2014, foram reportados mais de 10 mil *issues*.² Escolhemos o NetBeans porque também é uma ferramenta sólida para desenvolvimento de sistemas e porque tem uma comunidade atuante, com mais de 10 mil *issues* abertas até fevereiro de 2014.³ Escolhemos o Xrefactory por ter sido testada em trabalho relacionado ao nosso por Gligoric et al. [14].⁴ Finalmente, escolhemos o OpenRefactory⁵ por ser uma ferramenta proposta por Gligoric et al. [14], trabalho relacionado ao nosso, como plataforma de pesquisa para implementação de refatoramentos.

Tipos de Refatoramentos

Analizamos nossa abordagem em quatro dos seis tipos de refatoramentos do Eclipse CDT. Os tipos de refatoramentos que o Eclipse CDT suporta podem ser vistos na Tabela 4.1. O refatoramento *Hide Method* é aplicado apenas em elementos da linguagem C++, e não o consideramos neste experimento. O refatoramento *Toggle Function* extrai o cabeçalho

²Pesquisa realizada no Bugzilla <https://bugs.eclipse.org/bugs/> buscando todas as *issues* abertas para o Eclipse CDT.

³Pesquisa realizada no Bugzilla <https://netbeans.org/bugzilla/> buscando todas as *issues* abertas para o NetBeans CND.

⁴Não identificamos um *issue tracker* para esta ferramenta.

⁵Não identificamos um *issue tracker* para esta ferramenta.

Refatoramentos Eclipse CDT
Rename
Extract Function
Extract Constant
Extract Local Variable
Toggle Function
Hide Method

Tabela 4.1: Tipos de refatoramentos suportados pelo Eclipse CDT.

Refatoramentos
Rename Parameter
Rename Function
Rename Global Variable
Rename Local Variable
Rename #define
Extract Function
Extract Constant
Extract Local Variable

Tabela 4.2: Tipos de refatoramentos avaliados no experimento.

de uma função para um arquivo de cabeçalho (.h). Como atualmente a teoria do CDOLLY não especifica programas com mais de um arquivo, não consideramos esse tipo de refatoramento. Dividimos o refatoramento *Rename* em quatro tipos distintos para facilitar o entendimento e por acharmos que os elementos da linguagem C os quais o *Rename* refatora são diferentes o suficiente para essa divisão. De fato, o Eclipse JDT faz essa divisão em sua interface, e também o Fowler [12]. A Tabela 4.2 mostra todos os refatoramentos que analisamos. Definimos, assim, os refatoramentos a serem testados: {*Rename Parameter*, *Rename Function*, *Rename Global Variable*, *Rename Local Variable*, *Rename #define*, *Extract Function*, *Extract Constant*, *Extract Local Variable*}. Dessa lista, definimos os refatoramentos que refatoram elementos interno às funções: {*Rename Local Variable*, *Extract Function*, *Extract Constant*, *Extract Local Variable*}.

Para ilustrar como cada tipo de refatoramento presente no Eclipse CDT funciona, iremos exemplificá-los. O refatoramento *Toggle Function* move uma função para um arquivo de cabeçalho (.h). O Código Fonte 4.1, formado pelo arquivo f.c e t.h (vazio), imprime um número elevado à segunda potência. Para reutilizar a função `pow_2` (linha 3) em outros

arquivos, aplica-se o refatoramento *Toggle Function* na função, levando a função `pow_2` para o arquivo `t.h`, como pode ser visto no Código Fonte 4.2 (linhas 9-11).

Código Fonte 4.1: Exemplo de aplicação do refatoramento *Toggle Function*. Será selecionada a função `pow_2`.

```
1  _____ f.c _____
2  #include <t.h>
3  int pow_2(int num){
4      return num * num;
5  }
6  void main(){
7      int var;
8      ...
9      printf("%d", pow_2(var));
10 }
11 _____ t.h _____
12 //empty
```

Código Fonte 4.2: Resultado após aplicação do refatoramento *Toggle Function* na função `pow_2`.

```
1  _____ f.c _____
2  #include <t.h>
3  void main(){
4      int var;
5      ...
6      printf("%d", pow_2(var));
7  }
8  _____ t.h _____
9  int pow_2(int num){
10     return num * num;
11 }
```

O refatoramento *Rename* renomeia elementos tais como parâmetros de função, nome de função, variável global, variável local e identificador de `#define`, e atualiza todas as referências ao elemento refatorado. Um exemplo da aplicação do *Rename Function* pode ser visto nos Códigos Fonte 4.3 e 4.4. Ao aplicar o refatoramento na função `pow_2` (linha 1 do Código Fonte 4.3), passando como novo nome `power_2`, o Eclipse CDT renomeia a função (linha 1 do Código Fonte 4.4), bem como suas referências (linha 7).

Código Fonte 4.3: Exemplo de aplicação do refatoramento *Rename Function*. Será refatorada a função `pow_2`.

```

1 int pow_2(int num){
2     return num * num;
3 }
4 void main(){
5     int var;
6     ...
7     printf("%d", pow_2(var));
8 }

```

Código Fonte 4.4: Resultado após aplicação do refatoramento *Rename Function* na função `pow_2`.

```

1 int power_2(int num){
2     return num * num;
3 }
4 void main(){
5     int var;
6     ...
7     printf("%d", power_2(var));
8 }

```

O refatoramento *Extract Function* define uma nova função a partir de elementos selecionados. Por exemplo, o trecho de código `var * var` (linha 4 do Código Fonte 4.5) pode ser reusado várias vezes no programa, sendo possível extraí-lo para uma função. Ao selecionar esse trecho de código e aplicar o refatoramento *Extract Function*, passando como nome de função `pow_2`, o Eclipse CDT define a função (linha 1 do Código Fonte 4.6) e atualiza o código para fazer referência a essa nova função (linha 7).

Código Fonte 4.5: Antes da aplicação do refatoramento *Extract Function* em `var * var`.

```

1 void main(){
2     int var;
3     ...
4     int result = var * var;
5     printf("%d", result);
6 }

```

Código Fonte 4.6: Depois da aplicação do refatoramento *Extract Function*.

```

1 int pow_2(int num){
2     return num * num;
3 }
4 void main(){
5     int var;
6     ...
7     int result = pow_2(var);
8     printf("%d", result);
9 }

```

O refatoramento *Extract Constant* define uma constante a partir da seleção de uma expressão ou literal. Por exemplo, o desenvolvedor deseja definir o valor 100 (linha 3 do

Código Fonte 4.7) como constante. Ao selecionar e aplicar a esse literal o refatoramento *Extract Constant*, passando como nome MAX_NUMBER, o Eclipse CDT define a constante (linha 1 do Código Fonte 4.8) e atualiza a referência à literal 100 (linha 4).

Código Fonte 4.7: Antes da aplicação do refatoramento *Extract Constant* no literal 100.

```
1 void main() {  
2     int i;  
3     for (i = 0; i < 100; i++){  
4         ...  
5     }  
6 }
```

Código Fonte 4.8: Depois da aplicação do refatoramento *Extract Constant*.

```
1 static const int MAX_NUMBER =  
    100;  
2 void main() {  
3     int i;  
4     for (i = 0; i < MAX_NUMBER; i  
        ++)  
5         ...  
6     }  
7 }
```

Por fim, o refatoramento *Extract Local Variable* define uma variável local a partir da seleção de uma expressão ou literal. Por exemplo, o desenvolvedor deseja definir uma variável local a partir da expressão $100+i$ (linha 4 do Código Fonte 4.9) para atualizar as referências a essa expressão visando melhorar a manutenibilidade do programa. Ao selecionar e aplicar a essa expressão o refatoramento *Extract Local Variable*, passando como nome de variável *w*, o Eclipse CDT define a variável local (linha 4 do Código Fonte 4.10) e atualiza a referência à expressão (linha 5). Para aumentar a manutenibilidade, o desenvolvedor pode alterar a expressão $100+i$ (linha 6) para *w*.

Código Fonte 4.9: Antes da aplicação do refatoramento *Extract Local Variable* na expressão $100+i$.

```
1 void main(){
2   int x,i;
3   ...
4   if (x == (100+i)){
5     printf("%d", 100+i);
6   }
7 }
```

Código Fonte 4.10: Depois da aplicação do refatoramento *Extract Local Variable*.

```
1 void main(){
2   int x,i;
3   ...
4   int w = 100 + i;
5   if (x == (w)){
6     printf("%d", 100+i);
7   }
8 }
```

Entrada para os Refatoramentos

Para cada refatoramento, é necessário passar o nome do elemento para o qual será refatorado. Na Tabela 3.2 são mostrados todos os tipos de nomes e seus valores que usamos no experimento. Escolhemos esses valores a partir de estudos de bugs encontrados na literatura [40; 41; 14].

Programas C

Geramos programas com o CDOLLY com as restrições necessárias para cada tipo de refatoramento.

4.2.3 Instrumentação

O experimento foi executado em um computador MacBook Pro com 8GB de memória RAM e processador Core i7 de 2.66GHz. Utilizamos o sistema operacional Mac OS X 10.9.1. Compilamos os programas C com o compilador Apple LLVM version 5.0 (clang-500.2.79). Utilizamos o CDOLLY 1.0 para gerar os programas, o CATESTER 1.0 para gerar os testes de unidade dos programas gerados e o CREXECUTOR 1.0 para realizar os refatoramentos, a execução dos testes de regressão e a análise das falhas.

O CATESTER implementa duas estratégias para geração de testes de unidade. Uma das estratégias é baseada na quantidade de sequências geradas por função, e a outra é baseada no tempo para geração dos testes de um programa. Utilizamos esta última estratégia no nosso experimento, configurando 500ms como tempo máximo de geração de testes de unidade por

Refatoramento	PG	PC (%)	Tempo Total (h)	Erro de Compilação		Mudança Comportamental	
				Falha	Bug	Falha	Bug
Rename Parameter	18.486	52	12.00	3.875	4	333	1
Rename Function	19.500	67	33.54	6.345	5	0	0
Rename Global Variable	2.400	85	1.70	909	4	118	1
Rename Local Variable	10.500	74	5.60	3.367	4	451	1
Rename #define	1.464	71	0.70	411	4	0	0
Extract Function	10.500	74	8.86	4.856	7	871	1
Extract Constant	10.500	74	9.23	4.174	8	738	1
Extract Local Variable	10.500	74	12.76	3.883	5	846	1
Total	83.850	71	84.39	27.820	41	3.357	6

Tabela 4.3: Resultado da avaliação de refatoramentos do Eclipse CDT pelo CREXECUTOR.

programa.

Para os testes de refatoramento, utilizamos o Eclipse CDT Juno (build 8.1.1.201209170703), o NetBeans CND 7.4, o Xrefactory 2.0.14 e o OpenRefactory demo de fevereiro.

4.3 Operação

Avaliamos nossa técnica com diferentes refatoramentos do Eclipse CDT, e os resultados encontram-se na Tabela 4.3. Foram detectados 47 bugs, sendo 41 relacionados a erros de compilação e seis relacionados à mudança comportamental.⁶

Escopo

Para todos os tipos de refatoramentos, executamos o CDOLLY com escopo 3 para as instruções das funções, e 2 para o restante dos elementos (linha 8 do Código Fonte 4.11). Foi

⁶Uma lista com todos os bugs e o status de cada um pode ser encontrado em https://sites.google.com/site/cautomaticefactor/ides_refactoring_bugs.

especificado escopo 3 para as instruções das funções para permitir ao CDOLLY gerar códigos com declaração de variável local, macro do tipo `#ifdef` e instrução de retorno (`return`). Foi especificado escopo 2 para os demais elementos para não haver uma explosão de estados.

Refatoramento *Rename Parameter*

Para o refatoramento *Rename Parameter*, especificamos as restrições em Alloy do Código Fonte 4.11. Especificamos duas subassinaturas, `main` (linha 2), do tipo `Function`, e `param` (linha 3), do tipo `LocalVar`. Especificamos também um fato (linhas 4 a 6), onde definimos que a subassinatura `param` faz parte da relação `main.param`, ou seja, o `param` é parâmetro da função `main`. A linha 1 referencia o arquivo principal da teoria em Alloy do CDOLLY.

Código Fonte 4.11: Restrições Alloy para geração de programas pelo CDOLLY para refatoramento *Rename Parameter*.

```
1 open core
2 one sig main extends Function {}
3 one sig param extends LocalVar {}
4 fact {
5     param in main.param
6 }
7 pred show[] {}
8 run show for 2 but 3 Stmt
```

O CDOLLY gerou 18.486 programas, com taxa de compilação de 52%. A execução completa da técnica para esta implementação de refatoramento levou 12h. Ocorreram 3.875 erros de compilação que foram classificados em 4 bugs distintos: o Eclipse permitiu refatorar o nome do parâmetro para *nome com espaço, palavra reservada, operador & e literal*. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.12 e 4.13. O Eclipse permitiu refatorar o parâmetro de `param` (linha 2 do Código Fonte 4.12) para `int` (linha 2 do Código Fonte 4.13). O compilador lançou a mensagem de erro *cannot combine with previous 'float'*.

Código Fonte 4.12: Programa gerado pelo CDOLLY para refatoramento *Rename Parameter*. Eclipse permitiu refatorar `param` para `int`.

```

1 #define id1
2 void main (float param){
3     int var = 100;
4     #ifdef id1
5         var = 10;
6     #endif
7 }
```

Código Fonte 4.13: Bug devido a refatoramento para nome igual à palavra reservada `int` de C.

```

1 #define id1
2 void main (float int){
3     int var = 100;
4     #ifdef id1
5         var = 10;
6     #endif
7 }
```

Ocorreram 333 falhas de mudança comportamental que foram classificadas em 1 bug: o Eclipse permitiu refatorar o nome do parâmetro para um nome com `*` (operador de ponteiro) como prefixo. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.14 e 4.15. O Eclipse permitiu refatorar o parâmetro de `param` (linha 1 do Código Fonte 4.14) para `*param` (linha 1 do Código Fonte 4.15).

É importante citar que um parâmetro em C pode ser do tipo ponteiro. Porém, ao se permitir refatorar o nome de um parâmetro para um ponteiro, é necessário que o comportamento do programa se mantenha. Neste caso específico, o Eclipse refatorou as referências de `param` para `*param`, e os testes de unidade gerados pelo CATESTER continuaram a compilar com tais alterações. Ao executar os testes, foi lançado um sinal do tipo *Segmentation Fault*.

Código Fonte 4.14: Programa gerado pelo CDOLLY para refatoramento *Rename Parameter*. Eclipse permitiu refatorar `param` para `*param`

```

1 int main (int param){
2     return param;
3 }
```

Código Fonte 4.15: Bug devido a refatoramento para nome com operador `*`.

```

1 int main (int *param){
2     return *param;
3 }
```

Refatoramento *Rename Function*

Para o refatoramento *Rename Function*, especificamos as restrições em Alloy do Código Fonte 4.16, que restringe a geração dos programas para conter uma função com nome *func* (linha 2).

Código Fonte 4.16: Restrições Alloy para geração de programas pelo CDOLLY para refatoramento *Rename Function*.

```

1 open core
2 one sig func extends Function {}
3 pred show[] {}
4 run show for 2 but 3 Stmt

```

O CDOLLY gerou 19.500 programas, com taxa de compilação de 67%. A execução completa da técnica para esta implementação de refatoramento levou 33.54h. Ocorreram 6.345 erros de compilação que foram classificados em 5 bugs distintos: o Eclipse permitiu refatorar o nome do parâmetro para *nome com espaço, palavra reservada, operador '*', operador & e literal*. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.17 e 4.18. O Eclipse permitiu refatorar a função *func* (linha 2 do Código Fonte 4.17) para **func* (linha 2 do Código Fonte 4.18). O compilador lançou a mensagem de erro *returning 'float' from a function with incompatible result type 'float *'; take the address with &*.

Código Fonte 4.17: Programa gerado pelo CDOLLY para refatoramento *Rename Function*. Eclipse permitiu refatorar *func* para **func*.

```

1 float global = 4;
2 float func(){
3     float var = 4;
4     return global;
5 }

```

Código Fonte 4.18: Bug de *Rename Function* devido a refatoramento para nome com operador ***.

```

1 float global = 4;
2 float *func(){
3     float var = 4;
4     return global;
5 }

```

Particularmente, o bug de mudança de nome de função para conter o operador *** foi detectado a partir de uma única falha. Isso demonstra que para alguns casos é necessário aumentar o escopo dos elementos para aumentar a probabilidade de se detectar um bug.

Refatoramento *Rename Global Variable*

Para o refatoramento *Rename Global Variable*, especificamos as restrições em Alloy do Código Fonte 4.19. Especificamos que os códigos gerados teriam uma variável global `global` (linha 2).

Código Fonte 4.19: Restrições Alloy para geração de programas pelo CDOLLY para refatoramento *Rename Global Variable*.

```

1 open core
2 one sig global extends GlobalVar {}
3 pred show[] {}
4 run show for 2 but 3 Stmt

```

O CDOLLY gerou 2.400 programas, com taxa de compilação de 85%. A execução completa da técnica para esta implementação de refatoramento levou 1.7h. Ocorreram 909 erros de compilação que foram classificados em 4 bugs distintos: o Eclipse permitiu refatorar o nome da variável global para *nome com espaço, palavra reservada, operador '*' e literal*. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.20 e 4.21. O Eclipse permitiu refatorar a variável `global` (linha 1 do Código Fonte 4.20) para `0` (linha 1 do Código Fonte 4.21). O compilador lançou a mensagem de erro *cannot combine with previous 'int'*.

Código Fonte 4.20: Programa gerado pelo CDOLLY para refatoramento *Rename Global Variable*. Eclipse permitiu refatorar `global` para o literal `0`.

```

1 int global = 10;
2 int func(){
3     return 0;
4 }

```

Código Fonte 4.21: Bug de *Rename Global Variable* devido a refatoramento para nome igual à literal `0`.

```

1 int 0 = 10;
2 int func(){
3     return 0;
4 }

```

Ocorreram 118 falhas de mudança comportamental que foram classificadas em 1 bug: o Eclipse permitiu refatorar o nome da variável global para um nome com `*` (operador de ponteiro) como prefixo. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.22 e 4.23. O Eclipse permitiu refatorar a variável `global` (linha 1 do Código Fonte 4.22) para

*global (linha 1 do Código Fonte 4.23). Ao executar os testes de unidade, foi lançado um sinal do tipo *Segmentation Fault*, pois o ponteiro *global (linha 1) não foi alocado em posição de memória.

Código Fonte 4.22: Programa gerado pelo CDOLLY para refatoramento *Rename Global Variable*. Eclipse permitiu refatorar global para *global.

```
1 float global = 100;
2 float func(int param){
3     return global;
4 }
```

Código Fonte 4.23: Bug devido a refatoramento para nome com operador *.

```
1 float *global = 100;
2 float func(int param){
3     return *global;
4 }
```

Refatoramento *Rename Local Variable*

Para o refatoramento *Rename Local Variable*, especificamos as restrições em Alloy do Código Fonte 4.24. Especificamos que os códigos gerados teriam uma declaração de variável varDecl (linha 2), uma função func (linha 3) e que a declaração de variável seria uma das instruções da função (linha 5).

Código Fonte 4.24: Restrições Alloy para geração de programas pelo CDOLLY para refatoramento *Rename Local Variable*.

```
1 open core
2 one sig varDecl extends LocalVarDecl{}
3 one sig func extends Function{}
4 fact {
5     varDecl in func.stmt.elems
6 }
7 pred show[] {}
8 run show for 2 but 3 Stmt
```

O CDOLLY gerou 10.500 programas, com taxa de compilação de 74%. A execução completa da técnica para esta implementação de refatoramento levou 5.6h. Ocorreram 3.367 erros de compilação que foram classificados em quatro bugs distintos: o Eclipse permitiu refatorar o nome da variável local para *nome com espaço, palavra reservada, operador &*

e literal. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.25 e 4.26. O Eclipse permitiu refatorar a variável local `varDecl` (linha 2 do Código Fonte 4.25) para `a v` (linha 2 do Código Fonte 4.26). O compilador lançou a mensagem de erro *expected ';' at end of declaration*.

Código Fonte 4.25: Programa gerado pelo CDOLLY para refatoramento *Rename Local Variable*. Eclipse permitiu refatorar `varDecl` para nome com espaço (`a v`).

```
1 void func(){
2   float varDecl = 4;
3   int local = -1;
4 }
```

Código Fonte 4.26: Bug do *Rename Local Variable* devido a refatoramento para nome com espaço.

```
1 void func(){
2   float a v = 4;
3   int local = -1;
4 }
```

Ocorreram 451 falhas de mudança comportamental que foram classificadas em 1 bug: o Eclipse permitiu refatorar o nome da variável local para um nome com `*` (operador de ponteiro) como prefixo. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.27 e 4.28. O Eclipse permitiu refatorar o nome da variável `varDecl` (linha 2 do Código Fonte 4.27) para `*varDecl` (linha 2 do Código Fonte 4.28). Ao executar os testes de unidade, foi lançado um sinal do tipo *Segmentation Fault*, pois o ponteiro `*varDecl` (linha 2) não foi alocado em posição de memória.

Código Fonte 4.27: Programa gerado pelo CDOLLY para refatoramento *Rename Local Variable*. Eclipse permitiu refatorar `varDecl` para `*varDecl`.

```
1 void func(){
2   int varDecl = 0;
3   varDecl = 2;
4 }
```

Código Fonte 4.28: Bug do *Rename Local Variable* devido a refatoramento para nome com operador `*`.

```
1 void func(){
2   int *varDecl = 0;
3   *varDecl = 2;
4 }
```

Refatoramento *Rename #define*

Para o refatoramento *Rename #define*, especificamos as restrições em Alloy do Código Fonte 4.29. Especificamos que os códigos gerados teriam uma declaração de `#define` com `id identifier` (linhas 2 e 4).

Código Fonte 4.29: Restrições Alloy para geração de programas pelo CDOLLY para refatoramento *Rename #define*.

```

1 open core
2 one sig identifier extends Id {}
3 fact {
4   id in Define.defs
5 }
6 pred show[] {}
7 run show for 2 but 3 Stmt

```

O CDOLLY gerou 1.464 programas, com taxa de compilação de 71%. A execução completa da técnica para esta implementação de refatoramento levou 0.7h. Ocorreram 411 erros de compilação que foram classificados em quatro bugs distintos: o Eclipse permitiu refatorar o nome da variável local para *palavra reservada*, *operador '*'*, *operador &* e *literal*. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.30 e 4.31. O Eclipse permitiu refatorar o identificador `id` (linha 1 do Código Fonte 4.30) para `&id` (linha 1 do Código Fonte 4.31). O compilador lançou a mensagem de erro *macro names must be identifiers*.

Código Fonte 4.30: Programa gerado pelo CDOLLY para refatoramento *Rename #define*. Eclipse permitiu refatorar `id` para `&int`.

```

1 #define id
2 int func(int param){
3   float varDecl = -1;
4   return param;
5 }

```

Código Fonte 4.31: Bug do *Rename #define* devido a refatoramento para nome com operador `&`.

```

1 #define &id
2 int func(int param){
3   float varDecl = -1;
4   return param;
5 }

```

Refatoramento *Extract Function*

Para o refatoramento *Extract Function*, especificamos as restrições em Alloy do Código Fonte 4.32. Especificamos que os códigos gerados teriam uma declaração de variável `varDecl` (linha 2), uma função `func` (linha 3) e que a declaração de variável seria uma das instruções da função (linha 5).

Código Fonte 4.32: Restrições Alloy para geração de programas pelo CDOLLY para refatoramento *Extract Function*.

```
1 open core
2 one sig varDecl extends LocalVarDecl {}
3 one sig func extends Function {}
4 fact {
5   varDecl in func.stmt.elems
6 }
7 pred show[] {}
8 run show for 2 but 3 Stmt
```

O CDOLLY gerou 10.500 programas, com taxa de compilação de 74%. A execução completa da técnica para esta implementação de refatoramento levou 8.86h. Ocorreram 4.856 erros de compilação que foram classificados em 7 bugs distintos: o Eclipse permitiu refatorar o nome da função para *nome com espaço*, *palavra reservada*, *operador '*'*, *operador &*, *literal*, *nome de variável global existente*, *nome de função existente e identificador de define existente*. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.33 e 4.34. O Eclipse permitiu extrair o literal 2 (linha 2 do Código Fonte 4.33) para a função `func2` (linha 1 do Código Fonte 4.34), porém uma função com mesmo nome já existia (linha 8 do Código Fonte 4.34). O compilador lançou a mensagem de erro *redefinition of 'func2'*.

Código Fonte 4.33: Programa gerado pelo CDOLLY para refatoramento *Extract Function*. Eclipse permitiu extrair função para nome de outra já existente.

```

1 int func(){
2     int varDecl = 2;
3     return varDecl;
4 }
5 int func2(){
6     int varDecl = 10;
7     return varDecl;
8 }

```

Código Fonte 4.34: Bug do *Extract Function* devido a refatoramento para nome de função já existente.

```

1 int func2(){
2     return 2;
3 }
4 int func(){
5     int varDecl = func2();
6     return varDecl;
7 }
8 int func2(){
9     int varDecl = 10;
10    return varDecl;
11 }

```

Ocorreram 871 falhas de mudança comportamental que foram classificadas em 1 bug: o Eclipse permitiu refatorar o nome de uma função para um nome com operador * como prefixo. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.35 e 4.36. O Eclipse permitiu extrair o literal 4 (linha 2 do Código Fonte 4.35) para uma função com nome *old (linha 1 do Código Fonte 4.36). Ao executar os testes de unidade, foi lançado um sinal do tipo *Segmentation Fault*, pois a função *old (linha 1) não retorna um ponteiro para inteiro, e sim um literal.

Código Fonte 4.35: Programa gerado pelo CDOLLY para refatoramento *Extract Function*. Eclipse permitiu refatorar *func* para *old

```

1 void func(){
2     float local = 4;
3 }

```

Código Fonte 4.36: Bug devido a refatoramento para nome com operador *.

```

1 int *old(){
2     return 4;
3 }
4 void func(){
5     float local = *old();
6 }

```

Refatoramento *Extract Constant*

Para o refatoramento *Extract Constant*, especificamos as restrições em Alloy do Código Fonte 4.37. Especificamos que os códigos gerados teriam uma declaração de variável `varDecl` (linha 2), uma função `func` (linha 3) e que a declaração de variável seria uma das instruções da função (linha 5).

Código Fonte 4.37: Restrições Alloy para geração de programas pelo CDOLLY para refatoramento *Extract Constant*.

```
1 open core
2 one sig varDecl extends LocalVarDecl{}
3 one sig func extends Function{}
4 fact {
5   varDecl in func.stmt.elems
6 }
7 pred show[] {}
8 run show for 2 but 3 Stmt
```

O CDOLLY gerou 10.500 programas, com taxa de compilação de 74%. A execução completa da técnica para esta implementação de refatoramento levou 9.24h. Ocorreram 4.174 erros de compilação que foram classificados em oito bugs: o Eclipse permitiu refatorar o nome da função para *nome com espaço*, *palavra reservada*, *operador '*'*, *operador &*, *literal*, *nome de variável global existente*, *nome de função existente e identificador de define existente*. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.38 e 4.39. O Eclipse permitiu extrair o literal 10 (linha 3 do Código Fonte 4.38) para a constante `global` (linha 1 do Código Fonte 4.39), porém uma variável global com mesmo nome já existia (linha 2 do Código Fonte 4.39). O compilador lançou a mensagem de erro *redefinition of 'global' with a different type: 'int' vs 'const int'*.

Código Fonte 4.38: Programa gerado pelo CDOLLY para refatoramento *Extract Constant*. Eclipse permitiu extrair constante 10 para nome de variável global já existente.

```

1 int global = 0;
2 float func(){
3     float varDecl = 10;
4     return varDecl;
5 }
```

Código Fonte 4.39: Bug do *Extract Constant* devido a refatoramento para nome com espaço.

```

1 static const int global = 10;
2 int global = 0;
3 float func(){
4     float varDecl = global;
5     return varDecl;
6 }
```

Ocorreram 738 falhas de mudança comportamental que foram classificadas em um bug: o Eclipse permitiu extrair um literal para uma constante com operador * como prefixo. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.40 e 4.41. O Eclipse permitiu extrair o literal 2 (linha 2 do Código Fonte 4.40) para uma constante com nome *old (linha 1 do Código Fonte 4.41). Ao executar os testes de unidade, foi lançado um sinal do tipo *Segmentation Fault*, pois a constante *old (linha 1) não foi alocada em uma posição de memória.

Código Fonte 4.40: Programa gerado pelo CDOLLY para refatoramento *Extract Constant*.

```

1 void func(){
2     float varDecl = 2;
3 }
```

Código Fonte 4.41: Bug devido a refatoramento para nome com operador *.

```

1 static const int *old = 2;
2 void func(){
3     float varDecl = *old;
4 }
```

Refatoramento *Extract Local Variable*

Para o refatoramento *Extract Local Variable*, especificamos as restrições em Alloy do Código Fonte 4.42. Especificamos que os códigos gerados teriam uma declaração de variável varDecl (linha 2), uma função func (linha 3) e que a declaração de variável seria uma das instruções da função (linha 5).

Código Fonte 4.42: Restrições Alloy para geração de programas pelo CDOLLY para refatoramento *Extract Local Variable*.

```

1 open core
2 one sig varDecl extends LocalVarDecl{}
3 one sig func extends Function{}
4 fact {
5   varDecl in func.stmt.elems
6 }
7 pred show[] {}
8 run show for 2 but 3 Stmt

```

O CDOLLY gerou 10.500 programas, com taxa de compilação de 74%. A execução completa da técnica para esta implementação de refatoramento levou 12.76h. Ocorreram 3.883 erros de compilação que foram classificados em cinco bugs distintos: o Eclipse permitiu extrair variável local para nomes *nome com espaço, palavra reservada, operador '*', operador &, literal e identificador de define existente*. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.43 e 4.44. O Eclipse permitiu extrair o literal 2 (linha 2 do Código Fonte 4.43) para a variável local `&old` (linha 2 do Código Fonte 4.44), porém o operador `&` só deve ser usado para referenciar o endereço em memória de um ponteiro. O compilador lançou a mensagem de erro *expected identifier or '('*.

Código Fonte 4.43: Programa gerado pelo CDOLLY para refatoramento *Extract Local Variable*. Eclipse permitiu extrair variável local para nome `&old`.

```

1 int func() {
2   int varDecl = 2;
3   return varDecl;
4 }

```

Código Fonte 4.44: Bug do *Extract Local Variable* devido a refatoramento para nome com operador `&`.

```

1 int func() {
2   int &old = 2;
3   int varDecl = &old;
4   return varDecl;
5 }

```

Ocorreram 846 falhas de mudança comportamental que foram classificados em um bug: o Eclipse permitiu extrair um literal para uma variável local com `*` (operador de ponteiro) como prefixo. Um exemplo de um bug pode ser visto nos Códigos Fonte 4.45 e 4.46. O

Refatoramento	NetBeans CND	Xrefactory	OpenRefactory
Rename Parameter	1	3	1
Rename Function	1	2	1
Rename Global Variable	-	-	-
Rename Local Variable	4	-	5
Rename #define	3	8	-
Extract Function	-	-	-
Extract Constant	-	-	-
Extract Local Variable	-	-	-
Total	9	13	7

Tabela 4.4: Bugs detectados nas ferramentas NetBeans CND, Xrefactory e OpenRefactory. Maiores detalhes sobre cada um dos bugs podem ser encontrados em <http://www.dsc.ufcg.edu.br/~spg/cautomatictester>.

Eclipse permitiu extrair o literal 1 (linha 2 do Código Fonte 4.45) para uma variável local com nome `*old` (linha 2 do Código Fonte 4.46). Ao executar os testes de unidade, foi lançado um sinal do tipo *Segmentation Fault*, pois a variável `*old` (linha 2 do Código Fonte 4.46) não foi alocada em uma posição de memória.

Código Fonte 4.45: Programa gerado pelo CDOLLY para refatoramento *Extract Local Variable*. Eclipse permitiu extrair variável local para nome `*old`.

```

1 float func(){
2     float varDecl = 1;
3     return varDecl;
4 }
```

Código Fonte 4.46: Bug devido a refatoramento para nome com operador `*`.

```

1 float func(){
2     int *old = 1;
3     float varDecl = *old;
4     return varDecl;
5 }
```

Bugs no NetBeans, Xrefactory e OpenRefactory

Replicamos manualmente os bugs detectados no Eclipse CDT tanto no NetBeans CND, quanto no Xrefactory e OpenRefactory. Detectamos 9 bugs no NetBeans, 13 no Xrefactory e sete no OpenRefactory, que podem ser vistos na Tabela 4.4.

Refatoramentos NetBeans CND
Rename
Move
Copy
Safely Delete
Change Function Parameters
Encapsulate Fields
Introduce Variable
Introduce Constant
Introduce Field
Introduce Method

Tabela 4.5: Tipos de refatoramentos suportados pelo NetBeans CND.

Refatoramentos Xrefactory
Rename
Extract Method
Extract Local
Extract Constant
Toggle Function

Tabela 4.6: Tipos de refatoramentos suportados pelo Xrefactory.

Refatoramentos OpenRefactory
Rename
Add Local Variable
Add Reflexive Assignment
Move Expression Assignment
Change Integer Type
Add Integer Cast
Replace Arithmetic Operator
Add Null Statement Refactoring
Format Condition Statement Refactoring
Add Else Statement
Declare One Variable per Line Refactoring
Add Default to Switch Statement
Convert For to While
Extract Unary Arithmetic Refactoring
Expand Arithmetic Operator

Tabela 4.7: Tipos de refatoramentos suportados pelo OpenRefactory.

O fato de termos detectado mais bugs no Eclipse CDT (47) que nas outras ferramentas não significa que o Eclipse CDT seja menos testada ou que seja menos confiável. Na prática não pudemos testar a maioria dos refatoramentos do Eclipse CDT nas outras ferramentas pois cada uma destas implementam tipos de refatoramentos próprios. Nas Tabelas 4.5, 4.6 e 4.7 são mostrados os tipos de refatoramentos do NetBeans CND, Xrefactory e OpenRefactory, respectivamente. Como nem todos os tipos de refatoramento do Eclipse CDT são encontrados nessas outras ferramentas, nem todos os bugs detectados no Eclipse puderam ser replicados nessas ferramentas.

4.4 Discussão

Nesta seção, discutimos os resultados do experimento.

4.4.1 Técnica

Discutimos nesta seção alguns itens relacionados à técnica aplicada no experimento.

Entradas para os Refatoramentos

Definimos os tipos de entradas para as implementações de refatoramentos (Tabela 3.2) de acordo com bugs detectados em outros estudos [40; 41; 14]. Consideramos que cada um dos tipos de entrada definidos são distintos, pois representam semânticas diferentes na sintaxe C. Por exemplo, o operador `*` é usado na sintaxe de ponteiros, e pode significar que é um tipo de ponteiro (`int *x;`), ou que é o conteúdo da posição para onde o ponteiro aponta (`printf("%d", *x);`). Já o operador `&` representa o endereço de um ponteiro (`printf("%p", &x);`).

Além disso, cada implementação de refatoramento têm condições que precisam ser implementadas para preservar o comportamento de um programa. Desta forma, espera-se que um tipo de entrada passado para diferentes refatoramentos seja tratado, potencialmente, de formas distintas. Para mostrar indícios de que isto ocorre, colocamos na Tabela 4.8 os bugs detectados de cada tipo de refatoramento (linhas) e os diversos tipos de entrada (colunas). Para o tipo de entrada operador `&`, por exemplo, nem a implementação do refatoramento *Rename Global Variable* nem a *Extract Function* apresentaram bugs, porém todos os outros apresentaram.

Percentual de Programas Refatorados

Analisamos também se as pré-condições de cada refatoramento foram implementadas, dada a quantidade de bugs detectados. Por exemplo, ao realizar o refatoramento *Rename Local Variable*, 11% falharam devido à tentativa de refatoramento para nome de variável ou macro já existente. Já o refatoramento *Rename Function* apresentou 18% de falhas. Além disso, a ferramenta Xrefactory, por exemplo, permite renomear o nome de uma função `f` para `&f`, porém não permite para `*f`. Esses são indícios que as pré-condições estão sendo verificadas pelas implementações de refatoramentos, mesmo que essas pré-condições não sejam completas, como é o caso das implementações que apresentaram bugs.

Refatoramento	Novo Nome	Palavra Reservada (int)	Literal (0)	Operador * (compilação)	Operador * (comportam.)	Operador &	Nome Espaço	Nome de Função Existente	Nome de Variável Global Existente	Identific. Define Existente
Rename Parameter		x	x		x	x	x			
Rename Function		x	x	x		x	x			
Rename Global Variable		x	x		x		x			
Rename Local Variable		x	x		x	x	x			
Rename #define		x	x	x		x				
Extract Function		x	x		x		x	x	x	x
Extract Constant		x	x	x	x	x	x	x	x	x
Extract Local Variable		x	x		x	x	x			x

Tabela 4.8: Bugs detectados nos refatoramentos por tipo de entrada do nome do elemento a ser refatorado.

Cobertura de Código

Não medimos em nosso experimento a cobertura de instruções de código. Tomamos essa decisão devido aos programas gerados serem simples, com poucas instruções e funções, e o corpo das funções terem apenas um ou dois *branches* (dois *branches* quando se tem `#ifdef`). A tendência é que a cobertura de instruções de código tenha chegado próximo a 100%.

Escopo do CDOLLY

Para a geração de programas com o CDOLLY, utilizamos escopo 3 para instruções de função e 2 para os restantes dos elementos. É um escopo bastante reduzido. De fato, o bug mostrado no Código Fonte 4.47, em que o Eclipse permitiu extrair um literal 10 (linha 4) para uma constante com nome contendo operador *, foi detectado em 23 programas que falharam, num universo de 10.500 programas. Se aumentarmos o escopo, a tendência é de aumentar os casos falhos e diminuir o fator sorte. Encontrar bugs em escopo reduzido é mais um indício da hipótese do escopo pequeno (*Small Scope Hypothesis*) [19], que diz que a maioria dos bugs têm contraexemplos pequenos, e que a maior parte dos bugs ocorre em menos de cinco entidades [2].

Código Fonte 4.47: Exemplo de bug ao realizar o refatoramento *Extract Constant*.

```

1 static const int *old = 10;
2 float global = *old;
3 void func(){
4     int varDecl = *old;
5 }

```

Isomorfismo dos Programas Gerados

Os programas gerados pelo CDOLLY são usados para testar ferramentas de refatoramento. Quanto mais distintos estruturalmente forem os programas, maior a probabilidade de se encontrar bugs diferentes. Alloy usa um solucionador SAT chamado KodKod [47; 1] que tende a eliminar instâncias similares. Porém, como o CDOLLY tem suas próprias regras de formação, é possível que instâncias similares (isomórficas) sejam geradas.

Um exemplo de programas isomórficos gerados pelo CDOLLY pode ser visto nos Códigos Fontes 4.48 e 4.49. Os programas são estruturalmente iguais, apenas se diferenciando nos valores passados para as variáveis (linhas 3-4 e 7-8).

Código Fonte 4.48: Isomorfismo entre programas: programa 1.

```

1 #define Id
2 void f1 (int var1){
3     float var2 = -1;
4     var2 = 4;
5 }
6 void f2 (int var1){
7     float var2 = 1;
8     var2 = 0;
9 }

```

Código Fonte 4.49: Isomorfismo entre programas: programa 2.

```

1 #define Id
2 void f1 (int var1){
3     float var2 = 2;
4     var2 = 100;
5 }
6 void f2 (int var1){
7     float var2 = -1;
8     var2 = 1;
9 }

```

Soares et al. [41] analisaram a taxa de isomorfismo do JDolly [44], ferramenta na qual o CDOLLY se inspirou. Utilizando escopos 1 e 2 não foram gerados programas isomórficos. Já utilizando escopos 3 e 4, foram gerados 21 e 35% de programas isomórficos, respectivamente. A tendência é que o CDOLLY apresente taxas de isomorfismo similares ao JDolly.

Quantidade de Programas Gerados

A geração de programas pelo CDOLLY é guiada pela teoria de C implementada em Alloy. A necessidade de se ter elementos específicos nos programas para que um refatoramento possa ser aplicado levou à necessidade de se implementar restrições para geração de determinados programas, restringindo assim o espaço de soluções. Por exemplo, para que se possa aplicar o refatoramento *Rename Global Variable*, é necessário que se tenha pelo menos uma variável global definida no programa (linha 2 do Código Fonte 4.19).

Devido a essas restrições, a quantidade de programas gerados varia de acordo com o tipo de refatoramento que se deseja aplicar. Por exemplo, ao se restringir o espaço de solução para programas que tenham elementos globais tais como variável global ou #define, a quantidade de programas gerados foi 2.400 e 1.464, respectivamente, bem abaixo da quantidade de programas dos outros tipos de refatoramentos, tais como *Rename Function* que foi de 19.500. Isso é explicado pois a maioria das instâncias geradas pelo CDOLLY não contém variáveis globais nem #define. Comparando o *Rename Function* com o *Rename Parameter* percebe-se que a quantidade de programas gerados pelo *Rename Function* é um pouco maior que a do *Rename Parameter* (18.486), pois nem todas as funções geradas pelo CDOLLY têm parâmetro. A quantidade de programas gerados para os refatoramentos de elementos de dentro do corpo de função, tais como *Rename Local Variable*, *Extract Function*, *Extract Constant* e *Extract Local Variable* manteve-se igual (10.500) pois são refatoramentos que são aplicados aos mesmos elementos, tendo as mesmas restrições para geração dos programas.

Tempo para Geração dos Programas

O tempo total de aplicação de cada tipo de refatoramento variou. Por exemplo, a aplicação do refatoramento *Extract Local Variable* durou duas vezes mais que a aplicação do refatoramento *Rename Local Variable*. Isso se deve ao fato dos refatoramentos terem pré e pós-condições diferentes. No caso do refatoramento *Rename Function*, que durou mais que 33h, eventualmente aparecia uma tela para o usuário informando se gostaria de prosseguir com o refatoramento, levando 15s para fechar. Isso aumentou muito o tempo total deste refatoramento.

Quantidade de Bugs Encontrados

A quantidade de bugs encontrados por tipo de refatoramento variou entre 4 e 5. Porém, os refatoramentos *Extract Function* e *Extract Constant* apresentaram a maior quantidade de

bugs: 7 e 8, respectivamente. Isso é explicado pelo fato de ao extrair elementos para uma área global, esses elementos passam a interagir com outros, aumentando a probabilidade de falha. Acreditamos, assim, que os testadores de ferramentas que implementam refatoramentos deveriam aumentar seus esforços nos testes destes tipos de refatoramentos.

Falsos Positivos

Além das falhas de mudança comportamental que deram origem à detecção dos bugs, o CREXECUTOR também detectou falhas que se revelaram falsos positivos. Esses falsos positivos ocorreram devido a um bug no CATESTER relacionado a funções que retornam valores do tipo `float`. O CATESTER não estava colocando os cabeçalhos das funções a serem testadas antes da função `main` e, ao executar uma sequência que referenciava uma função com retorno `float` (linha 3 do Código Fonte 4.50), estava sendo retornado um valor não esperado pelos testes de unidade (`1888236288.000000`). Pelas regras da linguagem C, é aconselhado que se coloque o cabeçalho de uma função que se encontra em arquivo externo antes da chamada desta função (linha 1 do Código Fonte 4.51). O comportamento de um programa que não segue esta regra é inesperado. O mesmo não ocorreu para funções que retornam `int`. Nenhum desses casos foram considerados mudança comportamental.

Código Fonte 4.50: Bug no CATESTER com funções que retornam `float`.

```

1 void main(void){
2   int x = 1;
3   float retfloat = func(x);
4   printf("%f",retfloat);
5 }
```

Código Fonte 4.51: Forma aconselhada de se referenciar função de arquivo externo.

```

1 float func(int num);
2 void main(void){
3   int x = 1;
4   float retfloat = func(x);
5   printf("%f",retfloat);
6 }
```

4.4.2 Outros Tipos de Bugs

Uma ferramenta que implementa refatoramentos, além de bugs de compilação e mudança comportamental, pode apresentar bugs de usabilidade [49], bugs de interface gráfica, bugs do tipo *Overly Strong Conditions* [43], bugs de transformação e bugs de quebra da ferramenta

(*crash*). Apesar do foco deste trabalho ter sido apenas em bugs de compilação e mudança comportamental, encontramos alguns outros tipos de bugs, como veremos a seguir. Todos esses bugs foram reportados. Pretendemos, como trabalho futuro, propor uma técnica para sistematicamente detectar esses bugs.

Bug de Transformação

Bugs de transformação são caracterizados por implementação errada das regras de um refatoramento. Ao analisar um dos códigos refatorados com *Extract Constant*, detectamos um bug de transformação. No Código Fonte 4.52, foi extraído o literal 1 (linha 3) para a constante `old` (linhas 1 e 4 do Código Fonte 4.53). Porém, o Eclipse refatorou não apenas o literal selecionado, mas todas as literais 1 do código, como pode ser visto na linha 2 do Código Fonte 4.53. No Eclipse JDT, por exemplo, este bug não ocorre. Pretendemos estender a técnica para detectar automaticamente esse tipo de bug usando a ferramenta Ref-Finder [23] proposta por Kim et al., que detecta se houve refatoramento no programa resultante a partir de *template matching*. O foco desta ferramenta é a linguagem Java. Pretendemos fazer algo similar para C.

Código Fonte 4.52: Código a ser refatorado com *Extract Function*.

```

1 float global = -1;
2 void func() {
3     int x = 1;
4     x = 4;
5 }

```

Código Fonte 4.53: Código refatorado com bug de transformação.

```

1 static const int *old = 1;
2 float global = -*old;
3 void func() {
4     int x = *old;
5     x = 4;
6 }

```

Bug de *Overly Strong Condition*

Overly Strong Conditions são condições fortes implementadas por ferramentas de refatoramento. Essas condições normalmente são definidas pelos desenvolvedores dessas ferramentas visando proteger o código de transformações erradas. Porém, elas podem impedir um refatoramento de ser aplicado. Durante a execução do refatoramento *Rename Parameter*, percebemos que a duração para refatorar alguns programas passou de 30s. Investigando, descobrimos que o refatoramento em programas que continham `#ifdef` eram os causadores

da lentidão. Um exemplo deste tipo de programa pode ser visto no Código Fonte 4.54. Ao tentar refatorar o nome do parâmetro `param` (linha 2), o CREXECUTOR retornou uma falha, informando a mensagem *The selected name could not be analyzed*. Ao executar este refatoramento via interface do Eclipse, é mostrada a mensagem *Found Problems: Refactoring contains 1 potencial match*, sendo possível, mesmo assim, continuar com o refatoramento. Ao tentar replicar esse bug no Xrefactory, o refatoramento não atualizou todas as referências de `param`, e ocorreu erro de compilação. Acreditamos que essa condição forte tenha sido implementada porque há referência a `param` dentro de elementos de pré-processamento (linhas 4 e 6), e o desenvolvedor da implementação do refatoramento deixou a cargo do usuário tomar a decisão da aplicação ou não do refatoramento.

Código Fonte 4.54: Exemplo de Overly Strong Condition quando se tenta refatorar o nome do parâmetro.

```
1 #define id
2 void func(float param){
3     #ifdef id
4         param = 2;
5     #else
6         param = 0;
7     #endif
8 }
```

Bug de Interface Gráfica

Bugs de Interface Gráfica são bugs causados por implementação errada de elementos de interface que executam um refatoramento. No Código Fonte 4.55, quando o CREXECUTOR tentou refatorar o `id A` (linha 1) via API de refatoramento do Eclipse, foram refatorados tanto o `id` da linha 1 quanto a referência ao `id` pelo `#ifdef` (linha 4). Porém, ao executar o mesmo refatoramento pela interface gráfica, apenas o `id` na linha 1 foi refatorado.

Código Fonte 4.55: Exemplo de bug de interface do Eclipse CDT. O mesmo refatoramento consegue ser aplicado via API.

```

1 #define A
2 int main(float param){
3     int varDecl = 1;
4     #ifdef A
5         varDecl = 4;
6     #endif
7     return varDecl;
8 }
```

4.4.3 Classificador de Bugs

O classificador de bugs utilizado na nossa técnica tem limitações. Para detecção dos bugs do refatoramento *Rename Function*, por exemplo, o classificador informou seis bugs. Analisamos os grupos de bugs e percebemos que havia dois que representavam o mesmo bug:

- *variable has incomplete type 'void';NOME_ESPAÇO*; e
- *expected ';' after top level declarator;NOME_ESPAÇO*.

Isso ocorreu porque o compilador lançou diferentes mensagens, pois os programas tinham diferentes elementos e tipos, como pode ser visto nos Códigos Fontes 4.56 (linha 1) e 4.57 (linha 2). Pretendemos, como trabalho futuro, analisar novas formas de classificação para aumentar a precisão de detecção de bugs.

Código Fonte 4.56: Refatorada função com espaço no nome. Erro de compilação: *variable has incomplete type 'void'*.

```

1 void a v(float param){
2 }
3 ...
```

Código Fonte 4.57: Refatorada função com espaço no nome. Erro de compilação: *expected ';' after top level declarator*.

```

1 float global = 1;
2 float a v(){
3     return global;
4 }
5 ...
```

4.4.4 Status dos Bugs Reportados

Até o presente momento, todos os 47 bugs reportados ao Eclipse CDT encontram-se com status *new*. Dos nove bugs reportados ao NetBeans CND, três foram marcados como duplicados entre si: renomear parâmetro para nome de parâmetro já existente, renomear variável para nome de parâmetro já existente e renomear identificador de *#define* para um identificador já existente. Acreditamos que esses bugs foram marcados como duplicados por terem códigos iguais no NetBeans. Dos outros bugs, dois estão com status *new* e quatro estão com status *reopened*. Foram marcados como reabertos após dúvidas endereçadas a nós pelos próprios desenvolvedores quanto à teoria de refatoramento. Enviamos por email os bugs do Xrefactory e do OpenRefactory por não termos encontrado um *issue tracker* para essas ferramentas.

4.5 Respostas aos Questionamentos

A seguir apresentamos as respostas às questões de pesquisa.

Q1 *Nossa abordagem é capaz de detectar bugs de compilação?*

Sim, a abordagem é capaz de detectar bugs de compilação. Detectamos automaticamente 41 bugs relacionados a erro de compilação no Eclipse CDT.

Q2 *Nossa abordagem é capaz de detectar bugs de mudança comportamental?*

Sim, a abordagem é capaz de detectar bugs de mudança comportamental. Detectamos automaticamente seis bugs relacionados a erros de mudança comportamental.

Q3 *Nossa abordagem é capaz de detectar bugs em implementações de refatoramentos aplicados a corpo de funções?*

Sim, a abordagem é capaz de detectar bugs de refatoramento que trabalha com corpos de função, tais como *Rename Local Variable*, *Extract Function*, *Extract Constant* e *Extract Local Variable*.

Q4 *Os bugs detectados ocorrem em outras implementações de refatoramentos?*

Sim. Replicamos no NetBeans, no Xrefactory e no OpenRefactory os bugs encontrados, além de variações destes bugs. Encontramos 9 bugs no NetBeans, 13 no Xrefactory e 6 no OpenRefactory.

4.6 Ameaças à Validade

Nesta seção apresentamos as ameaças à validade interna (Seção 4.6.1), à validade externa (Seção 4.6.2) e à validade de construção (Seção 4.6.3) do nosso experimento.

4.6.1 Validade Interna

Uma ameaça à validade interna está relacionada com a técnica aplicada para categorização de falhas. Como especificamos um padrão para encontrar mensagens de falhas, é possível ter dois bugs distintos com mensagem similares. Porém, como os programas gerados são reduzidos, e como as transformações realizadas nesses programas são pequenas (localizadas), a probabilidade que isso ocorra é mínima. Por exemplo, no trabalho de Soares et al. [41] com programas um pouco maiores não ocorreu casos de mais de um bug para uma mesma mensagem.

Devido à aleatoriedade da escolha entre o programa gerado e o tipo de nome do refatoramento, é possível que alguns tipos de bugs não sejam detectados. Além disso, há uma ameaça relacionada à representatividade dos programas C gerados. Quando comparado ao universo de possíveis construções da linguagem C, o que usamos para gerar os programas via CDOLLY foi muito pouco. Por exemplo, não usamos cabeçalhos (`.h`), estruturas (`struct`) ou `arrays`. Porém, mesmo tendo avaliado programas simples, conseguimos detectar diversos bugs. Pretendemos utilizar outras construções da linguagem C para minimizar essa ameaça.

O nosso classificador de mudança comportamental dividiu as falhas em falhas causadas por sinais e não sinais. Não detectamos falhas de mudança comportamental sem que fossem lançados sinais. Isso se explica devido aos programas simples utilizados para análise. Para as falhas por sinais detectamos os bugs automaticamente. Porém, inspecionamos manualmente as falhas que não lançaram sinais. Como foram milhares de falhas desse tipo, é possível que não tenhamos detectado bug por falha humana. Geraremos programas maiores e com mais construções para que mudanças comportamentais que não sejam por lançamento de sinais sejam detectadas.

4.6.2 Validade Externa

Uma ameaça à validade externa está relacionada aos bugs reportados. Até o presente momento, todos os bugs do Eclipse CDT reportados encontram-se com status novo, impossibilitando analisarmos se os bugs reportados de fato são distintos.

Outra ameaça diz respeito aos programas utilizados no experimento. Como eles são gerados pela teoria do CDOLLY, e não representam programas reais (programas desenvolvidos para uso de terceiros), não podemos afirmar que os bugs encontrados ocorrem normalmente com os programadores. Essa questão diz respeito ao quão importante são os bugs encontrados. Minimizamos essa ameaça gerando programas com elementos da linguagem normalmente encontrados em programas reais C, tais como #define, constantes, funções e variáveis.

Por fim, a escolha da ferramenta de implementações de refatoramentos pode não ter representado tão bem o universo de refatoramentos disponíveis para C. Minimizamos essa ameaça replicando manualmente os bugs encontrados em outras ferramentas tais como NetBeans CND, Xrefactory e OpenRefactory.

4.6.3 Validade de Construção

Uma ameaça à construção diz respeito à escolha dos tipos de nomes de entrada para os refatoramentos. Apesar de acreditarmos que são diferentes o suficiente para detectar diferentes bugs, eles podem ser interpretados como iguais para os diferentes tipos de refatoramentos, representando assim o mesmo bug. Essa ameaça pode ser diminuída aumentando a quantidade de valores de entrada.

Outra ameaça diz respeito à geração de testes pelo CATESTER. Os testes são gerados aleatoriamente, sem conhecimento específico do comportamento do programa. Sendo assim, é possível que sejam gerados testes com valores de parâmetros inválidos para um determinado programa. Essa ameaça foi minimizada no nosso experimento devido a termos controle na geração dos programas.

4.7 Considerações Finais

Neste capítulo apresentamos nossa técnica para detectar automaticamente bugs em ferramentas de refatoramento C. Nossa abordagem gera programas a partir de um modelo C em Alloy, gera testes de unidade para esses programas e executa a API da ferramenta a ser testada para aplicar o refatoramento. Categorizamos automaticamente as falhas de erro de compilação e de forma semi-automática as falhas de mudança comportamental em bugs.

Aplicamos nossa técnica na análise de bugs de refatoramento no Eclipse CDT, e detectamos 47 bugs. Analisamos esses bugs no NetBeans, no Xrefactory e no OpenRefactory e detectamos 29 bugs nessas ferramentas. Além de bugs relacionados à falha de compilação e mudança comportamental, detectamos bugs de interface gráfica, de transformação e *Overly Strong Condition*.

Capítulo 5

Trabalhos Relacionados

Neste capítulo, apresentamos e relacionamos com nossa abordagem trabalhos empíricos sobre testes (Seção 5.1) e refatoramentos (Seção 5.2)

5.1 Geração de Testes e Programas

Hanford [18] foi o primeiro a propor uma técnica para geração automática de testes. Ele desenvolveu um gerador aleatório de programas baseado em gramática (*Grammar-Based Test Generation - GBTB*) [5] com o intuito de testar compiladores de linguagens sensíveis ao contexto. Os programas gerados serviam como entrada para os compiladores e o resultado da compilação servia para análise dos bugs dos compiladores. Um dos grandes problemas de geração aleatória de testes é o quão efetivo estes são quando comparados com testes sistemáticos (desenvolvidos por testadores). Posteriormente, Duran e Ntafos [8] mostraram que a estratégia de testes aleatórios pode ser tão eficiente quanto testes sistemáticos. Além disso, argumentam que com geração aleatória de testes não é necessário tanto esforço para mantê-los como o é para testes sistemáticos. Apesar disso, os testes aleatórios normalmente não abrangem todo o código, apresentando assim baixa cobertura de instruções (*statement coverage*). Nossa técnica também utiliza geração automática de testes. Porém, esses são usados para análise dos programas pós refatoramentos. Na técnica de Hanford, os programas são usados para testar compiladores. Não avaliamos cobertura de instruções pois os programas gerados pelo CDOLLY tinham poucas instruções nas funções, com um ou dois *branches*.

Com o intuito de aumentar a cobertura de testes aleatórios, Godefroid et al. [16] desen-

volveram DART, uma ferramenta para analisar programas C que utiliza a técnica de execução simbólica [24]. Execução simbólica é uma técnica utilizada para determinar quais entradas fazem com que os diferentes caminhos (*branches*) de um programa sejam executados. DART analisa o programa a ser testado e executa a partir da função `main`, direcionando a geração dos dados para executar caminhos não percorridos. Uma das restrições de DART é que ele não gera dados para estruturas dinâmicas tais como ponteiros. Tal restrição foi eliminada por Sen et al. [39] na ferramenta CUTE, uma ferramenta que implementa a técnica de execução concreta e simbólica (Concolic) [25] para testar programas C. Nossa abordagem utiliza a técnica de geração aleatória de testes orientada a *feedback* [35]. Utilizando essa técnica, o CATESTER gera testes a partir das assinaturas das funções a serem testadas, e não leva em consideração as instruções dos corpos das funções. Dessa forma, a tendência é ter menor cobertura de instruções que o DART e CUTE, pois com execução simbólica busca-se executar todos os *branches* para resolver as expressões. Porém, execução simbólica demanda um poder computacional maior que a nossa abordagem.

Baresi e Miraz [3] argumentam que gerar testes de forma completamente aleatória não é eficiente devido à necessidade de muitos ciclos de computador até gerar testes com boa cobertura. Eles desenvolveram TestFul, uma ferramenta de geração de testes aleatórios gerados de forma guiada por um algoritmo genético multi-objetivos, buscando assim gerar testes com maior cobertura de instruções para programas Java. Apesar de ser um critério de aceitação de testes [53] importante, nossa abordagem não se baseia em cobertura de testes para guiar a geração dos testes de unidade. Nossa abordagem é baseada no reuso de valores de execuções passadas para diminuir a chance de se ter valores de entradas inválidas para as funções a serem testadas.

Pacheco et al. desenvolveram uma técnica de geração de testes aleatórios chamada *feedback-directed random test generation* [35; 34], que consiste na geração de dados de acordo com os valores de retorno de execuções de métodos de um programa. A técnica se baseia na geração de sequências de trechos de código do programa a ser testado e no armazenamento dos retornos para uso em sequências futuras. Desta forma, a técnica diminui a quantidade de valores inválidos passados às funções do programa utilizando valores gerados pelo próprio programa. Pacheco et al. usaram alguns oráculos para os testes de programas Java. Para métodos, usaram os seguintes oráculos: nenhuma chamada de mé-

todo pode lançar `NullPointerException`; nenhuma chamada de método pode lançar `AssertionError`. Para objetos, usaram os seguintes oráculos: `o.equals(o)` deve ser sempre `true` (identidade) e não pode lançar exceção; `o.hashCode()` e `o.toString()` não devem lançar exceção. Para reduzir a quantidade de testes gerados, a técnica faz uma análise das sequências sendo geradas e elimina as similares. Pacheco et al. implementaram Randoop e detectaram 12 bugs distintos nas JVMs da IBM e da SUN. Pacheco et al. também aplicaram a técnica na linguagem .Net [34]. Nossa abordagem de geração de testes de unidade de programas C com o CATESTER se baseia na técnica de *feedback-directed*, porém não eliminamos sequências similares. Além dos valores iniciais, passamos para as sequências valores extremos visando detectar bugs do tipo array fora da faixa. No CATESTER, a geração de testes de unidade pode ser tanto por tempo, como no Randoop, quanto por quantidade de testes por função. Randoop se baseia no paradigma orientado a objetos, enquanto o CATESTER é voltado para programas estruturados C.

Soares et al. alteraram o Randoop para gerar testes de unidade apenas para um conjunto de métodos específicos [41]. Eles utilizam o Randoop no SAFEREFACTOR, uma ferramenta para testes de implementações de refatoramentos Java. O SAFEREFACTOR gera um conjunto de programas Java, realiza os refatoramentos nesses programas, analisa os métodos em comuns e gera os testes de unidade apenas para esses métodos com o Randoop. Se os testes executados no programa original e no refatorado derem resultados distintos, o refatoramento em questão introduziu um bug no programa. Analisar previamente os métodos em comum das versões dos programas e gerar testes apenas para estes faz com que a mesma coleção de testes execute nas duas versões do programa, facilitando a análise das mudanças comportamentais. Além disso, o tempo de execução da técnica proposta diminui. Nossa abordagem gera testes de unidade para todas as funções dos programas gerados pelo CDOLLY, mesmo para aquelas que tiveram suas assinaturas alteradas pós refatoramento. Além disso, como a geração dos testes de unidade faz parte da segunda fase em nossa técnica, são gerados testes de unidade também para programas que não conseguiram ser refatorados e que apresentam erros de compilação pós refatoramento. Isso faz com que aumente o tempo de execução de nossa técnica. Pretendemos alterar a ordem de geração de testes para avaliar o impacto.

Soares et al. [42] desenvolveram o JDolly, uma ferramenta para geração de programas Java. Ela é baseada na linguagem de especificação formal Alloy. Através de um metamodelo

Java, JDolly gera exaustivamente programas de acordo com uma dada estrutura (número de classes, campos e métodos). Soares et al. utilizaram JDolly para gerar programas de entrada para testes de implementações de refatoramento. Nos baseamos no JDolly para desenvolver CDOLLY, uma ferramenta de geração de programas C utilizada em nossa técnica. O CDOLLY contém um metamodelo da linguagem C e gera programas a partir da tradução de instâncias geradas pelo Alloy Analyzer para C. Enquanto JDolly foca na geração estrutural de programas Java, o CDOLLY foca em geração de programas C com corpo de função.

API Sanity Checker [11] é um gerador automático de testes de unidade para API de bibliotecas compartilhadas de C/C++. A partir dos cabeçalhos das funções em arquivos .h, o API Sanity Checker gera testes visando capturar problemas tais como *crashes* nas bibliotecas. Utilizamos o CATESTER para gerar testes de unidade em nossa técnica. O CATESTER é uma ferramenta que gera testes de unidade para funções de qualquer programa em C, estando em arquivos de cabeçalhos (.h) ou arquivos fonte (.c). Já o API Sanity Checker foca em testes de bibliotecas, e apenas gera testes para funções em arquivos de cabeçalho.

Testes Diferenciais é uma técnica proposta por McKeeman [27] para testar dois ou mais programas similares frente a uma coleção de testes gerada aleatoriamente. Resultados diferentes são analisados pois são indícios de bug. Ele usou essa técnica para testar compiladores C e reportou vários bugs. Nossa abordagem usa testes diferenciais para comparar o resultado do refatoramento analisando o programa original e o refatorado.

Yan et al. [52] também usaram testes diferenciais no CSmith para testar compiladores C tais como GCC e LLVM e reportaram centenas de bugs críticos. O diferencial do CSmith consiste em diminuir os falsos positivos através da geração de programas não ambíguos a partir da eliminação dos comportamentos não especificados e não definidos do padrão C99. A exploração desses comportamentos pode ajudar na detecção de bugs, pois o mal uso da linguagem C (comportamentos não especificados tais como acesso a índice fora da faixa de um array) pode levar a códigos que tenham saídas não determinísticas. Exploramos isso através da variação do padrão C na compilação dos códigos gerados pelo CATESTER, porém não aplicamos essa técnica neste trabalho.

5.2 Refatoramentos

O termo refatoramento foi proposto por Opdyke [32] e popularizado por Fowler [12] para definir transformações aplicadas a um programa para melhorar sua estrutura interna, mantendo seu comportamento observável. Para garantir preservação de comportamento, Opdyke definiu um conjunto de condições a serem seguidas por um refatoramento. Porém, não foi desenvolvida prova formal da corretude dessas condições. A partir daí, ferramentas de desenvolvimento passaram a implementar refatoramentos, mas sem validação formal das condições necessárias para a transformação. A primeira delas foi Refactoring Browser, ferramenta proposta por Roberts [37] em sua tese, que definia pré e pós condições para os refatoramentos. Na prática, os desenvolvedores de ferramentas de refatoramento testam suas implementações utilizando coleções de testes. O problema dessa abordagem é que não é possível testar todas as possibilidades, pois as linguagens de programação normalmente não são definidas formalmente. Tokuda e Batory [46] mostraram que as condições dos refatoramentos não eram suficientes para preservar comportamento. Borba et al. [4] propuseram um conjunto de refatoramentos para um subconjunto da linguagem Java e provaram sua corretude utilizando uma semântica formal. Apesar de ser uma abordagem que garante a corretude do refatoramento, Borba et al. não consideraram todas as construções de Java. Provar formalmente refatoramentos exige que a linguagem do programa a ser refatorado seja definida formalmente, o que não ocorre na prática. Utilizamos verificação formal na geração de programas C com o CDOLLY para auxiliar nos testes de implementação de refatoramentos.

Daniel et al. [6] desenvolveram uma técnica para testar ferramentas de refatoramento a partir da geração de programas de forma imperativa. Ela é baseada no framework ASTGen, que gera exaustivamente os programas para o refatoramento, e avalia o resultado com seis oráculos. O ASTGen gera todas as combinações possíveis de um programa em um determinado escopo e os passa como entrada para os refatoramentos utilizando a abordagem *bounded-exhaustive-testing* [26]. Para avaliar o resultado do refatoramento, o ASTGen utiliza seis oráculos baseados na análise de compilação, mensagens de erro e estrutura do programa refatorado. O ASTGen detectou 21 bugs no Eclipse JDT e 24 no NetBeans. Dos 21 bugs do Eclipse JDT, 17 foram relacionados com erros de compilação, 3 relacionados a transformações incompletas e 1 relacionado a mudança comportamental. Nossa técnica

também se baseia na abordagem *bounded-exhaustive-testing*, porém, para gerar programas, utilizamos uma abordagem declarativa com modelagem na linguagem Alloy. Os oráculos para detecção de mudança comportamental do ASTGen são baseados na análise da árvore sintática (AST), enquanto os nossos oráculos são baseados na execução e análise dos resultados de testes de unidade. No Eclipse CDT, detectamos 41 bugs relacionados com erros de compilação e 6 relacionados com mudança comportamental.

Gligoric et al. [15] evoluíram o ASTGen e propuseram UDITA. O gerador do UDITA procura por todas as combinações das construções de Java para gerar programas. UDITA também permite ao testador a definição de restrições para filtrar a geração dos programas. Enquanto o UDITA usa o *model checker* Java Path Finder na busca das combinações, o CDOLLY usa um solucionador SAT, permitindo em ambas as abordagens eliminarem soluções similares. Além disso, as restrições para geração dos programas são escritas numa linguagem similar a Java, enquanto no CDOLLY as restrições são especificadas em Alloy, que é uma linguagem declarativa. Soares et al. [41] compararam a geração de programas pelo JDolly, ferramenta na qual nos inspiramos para desenvolver o CDOLLY, e UDITA. Quanto à especificação para geração dos programas, a do JDolly foi escrita em 14 LOC, enquanto a do UDITA foi escrita em 46 LOC. Isso se explica pelo fato da especificação em Alloy apresentar um alto nível de abstração em relação a uma linguagem similar a Java. Quanto aos programas gerados, o JDolly gerou programas mais isomórficos, aumentando assim o tempo para testar ferramentas de refatoramento.

Gligoric et al. evoluíram a técnica [14] e propuseram uma abordagem sistemática para testar ferramentas de refatoramentos a partir de programas reais. A abordagem, que tem foco em erros de compilação, consiste em refatorar programas reais, compilar os programas para detectar falhas, agrupar as falhas em bugs e minimizar o bug. Para refatorar os programas, Gligoric et al. usaram as APIs das implementações de refatoramentos. Para agrupar as falhas, a abordagem usou o classificador automático proposto por Jagannath [21]. Para minimizar os bugs, Gligoric et al. avaliaram manualmente e reportaram os bugs. Eles avaliaram a técnica nos Eclipses JDT e CDT e detectaram 77 bugs de Java e 43 de C. Os 43 bugs de C detectados estão relacionados a lançamento de exceção ou erros de compilação. Eles testaram os 5 tipos de refatoramentos disponíveis para programas C: *Rename*, *Extract Function*, *Extract Local Variable*, *Extract Constant* e *Toggle Function*. O refato-

ramento *Rename* foi aplicado para variáveis globais, variáveis locais, parâmetros, funções, membros de estrutura e macros. Desses, nossos testes só não aplicaram *Rename* para membros de estrutura, pois não levamos em consideração estruturas (*struct*) no modelo C do CDOLLY. Nossa abordagem, em contraste com a de Gligoric et al., utiliza programas sintéticos gerados pelo CDOLLY como entrada para os refatoramentos. A técnica proposta refatora todos os possíveis elementos de um programa para um dado tipo de refatoramento, o que aumenta o tempo de execução da técnica. Nossa abordagem, diferentemente, refatora um elemento por programa. Apesar de Gligoric et al. argumentarem que utilizar programas reais como entrada é mais efetivo que códigos sintéticos, na prática há diferentes estudos que usam códigos sintéticos, inclusive este, que encontraram diversos bugs [41; 43; 40]. Além disso, refatorar aplicações reais necessita do trabalho adicional de configurar o ambiente da aplicação, que muitas vezes pode se tornar complexo. Nossa abordagem detectou, além de 41 bugs de compilação, 6 bugs de mudança comportamental. Detectamos também, analisando manualmente os resultados dos refatoramentos, bugs relacionados à interface, *overly strong condition* e bug de transformação, tipos de bugs que Gligoric et al. não detectaram. Analisando a quantidade de bugs detectados por implementação de refatoramento, já que Gligoric et al. não informaram a lista de bugs reportada ao Eclipse, detectamos bugs diferentes. Reportamos 24 bugs de *Rename*, enquanto eles reportaram 2, e reportamos 9 bugs de *Extract Constant*, enquanto eles reportaram 4.

Soares et al. [41; 43] desenvolveram uma técnica para testar ferramentas de refatoramentos Java. A técnica consiste em quatro passos. Primeiramente, programas são gerados a partir do JDolly, que utiliza Alloy como linguagem de modelagem. Em seguida, o refatoramento a ser testado é aplicado a cada programa. Após isso, a ferramenta SAFEREFACTOR, que engloba a geração e execução de testes de unidade via Randoop, analisa as falhas de compilação e as mudanças comportamentais dos programas. Finalmente, as falhas são categorizadas em bugs. Soares et al. avaliaram a técnica em 29 implementações de refatoramentos do Eclipse JDT, NetBeans e JRRT e detectaram 57 bugs relacionados a erro de compilação e 63 bugs relacionados a mudanças comportamentais. Nossa abordagem é inspirada na técnica de Soares et al. e teve como principal objetivo mostrar evidências de que a técnica funcionaria em outra linguagem de outro paradigma. Além disso, consideramos na teoria em Alloy corpo de função para testar refatoramentos comportamentais, diferentemente de Soares et al., que

focaram em refatoramentos estruturais. Com essa abordagem, foi possível avaliar refatoramentos tais como *Extract Function*, *Extract Constant*, *Extract Local Variable* e *Rename Local Variable*, onde detectamos 28 bugs. O SAFEREFACTOR reduz o tempo de análise dos programas avaliando apenas os métodos em comum entre a versão refatorada e a original, ao passo que nossa abordagem avalia todas as funções de um programa. Detectamos também, analisando manualmente os resultados dos refatoramentos, bugs relacionados à interface e bug de transformação, tipos de bugs que Soares et al. não detectaram.

Mongiovi et al. [29] desenvolveram o SAFEREFACTOR IMPACT, uma ferramenta para testar implementações de refatoramentos. Diferentemente do SAFEREFACTOR, o SAFEREFACTOR IMPACT gera testes apenas para os métodos impactados pela transformação utilizando o analisador de impacto de mudanças Safira. Mongiovi et al. analisaram sua técnica com 45 transformações aplicados a programas orientados a objetos e orientados a aspectos. Compararam com SAFEREFACTOR e detectaram mudanças comportamentais que o SAFEREFACTOR não detectou e reduziu a quantidade de métodos passados para o gerador de testes de unidade Randoop. Pretendemos evoluir nossa técnica para gerar testes apenas para as funções impactadas pelas implementações de refatoramento.

Jagannath et al. [21] propuseram três técnicas para reduzir o tempo de detecção da primeira falha em técnicas baseadas em geração exaustiva de testes de ferramentas de refatoramentos (*bounded-exhaustive-testing*). Eles constataram que para detectar um bug são geradas diversas falhas. A primeira técnica visa diminuir a quantidade de falhas espaçando a análise dos programas gerados. A segunda visa reduzir a quantidade de programas gerados agrupando-os em programas maiores. A terceira visa diminuir a análise individual de todas as falhas agrupando-as por mensagens de erro, identificando cada grupo como um bug. Eles utilizaram o framework ASTGen e detectaram dezenas de bugs em refatoramentos do Eclipse JDT, demonstrando que mesmo reduzindo a quantidade de programas analisados chegaram a resultados similares de detecção de bugs em relação a técnicas que não utilizam essa abordagem, com a vantagem da redução do tempo. Nossa técnica analisa todos os programas gerados, e pretendemos espaçar essa análise para diminuição do tempo de detecção de bugs, além da diminuição do tempo de execução da técnica como um todo. Usamos abordagem similar para agrupar os bugs. Porém, acrescentamos como chave do grupo, além da mensagem de erro, o tipo de entrada do refatoramento, pois detectamos casos em que mensagens

iguais representavam bugs distintos.

Vakilian e Johnson [49] propuseram uma técnica para detectar bugs de usabilidade [30] em ferramentas de refatoramento baseada na metodologia do incidente crítico (critical incident technique - CIT). Essa metodologia detecta falhas de usabilidade analisando problemas nas interações do usuário com o sistema. Pela técnica proposta, qualquer caminho alternativo na tentativa de realizar um refatoramento é analisada como um possível problema de usabilidade. Vakilian e Johnson conduziram um estudo de caso a partir da instalação de um plugin no Eclipse de desenvolvedores e coletaram os dados. Analisaram manualmente as mensagens de refatoramento mais comuns coletadas e reportaram 15 bugs, sendo 13 novos. A técnica de Vakilian e Johnson detecta bugs de usabilidade, enquanto nossa técnica detecta bugs relacionados a erro de compilação e mudança comportamental nos programas refatorados. Eles analisaram manualmente as mensagens de caminhos alternativos dos refatoramentos, enquanto nossa técnica detecta bugs automaticamente a partir de agrupamento de mensagens.

Alguns elementos de uma linguagem são complexos de se refatorar devido à necessidade de se definir as condições do refatoramento. Em C, um desses elementos são as diretivas condicionais (ex.: `#ifdef`) que são analisadas por pré-processadores. Pré-processadores são ferramentas utilizadas para transformar o código de acordo com alguma diretiva de compilação. Ferramentas de análise de C normalmente lidam apenas com os programas após as diretivas condicionais terem sido analisadas, ou seja, após o código ter sido transformado pelos pré-processadores. Porém, para refatorar um programa, é necessário que seja dado como entrada o programa original. Garrido e Johnson [13] desenvolveram uma técnica para refatorar diretivas condicionais através da transformação de diretivas incompletas em completas e da análise da AST do programa. Também definiram condições para refatoramentos de diretivas condicionais de C. Nossa abordagem permite testar o refatoramento de elementos de diretivas de pré-processamento, e pode ser usado como técnica para testar as condições do refatoramento.

Capítulo 6

Conclusões

Neste trabalho, propomos uma abordagem para testar implementações de refatoramentos de programas C para detectar erros de compilação e mudanças comportamentais. Nossa abordagem tem o intuito de testar refatoramentos que mudam tanto a estrutura de um programa em C (funções) quanto seu comportamento (corpo das funções). Ela consiste em cinco fases. Primeiro, geramos automaticamente diversos programas C para serem refatorados a partir do CDOLLY¹, que foi proposto baseado em uma especificação em Alloy. Depois, geramos automaticamente uma coleção de testes de unidade dos programas via o CATESTER². Em seguida, aplicamos o refatoramento através do CREXECUTOR³ nos programas gerados utilizando a API de refatoramentos da ferramenta a ser testada. Após isso, executamos a coleção de testes nos programas refatorados e, por fim, classificamos os erros de compilação e mudanças comportamentais identificados em bugs. Como parte de nossa técnica, desenvolvemos um gerador de programas C (CDOLLY) que utiliza a linguagem de especificação formal Alloy. O CDOLLY recebe as restrições estruturais dos programas a serem gerados e traduz todas as instâncias do modelo criadas pelo Alloy Analyzer em programas C. Além disso, desenvolvemos o CATESTER para gerar automaticamente testes de unidade para os programas gerados. Ele é baseado na abordagem de geração exaustiva de testes direcionada a *feedback* (*feedback-directed random testing generation*). Avaliamos nossa abordagem em 8 refatoramentos do Eclipse CDT [9] e detectamos 41 bugs relacionados a erros de compilação

¹CDOLLY pode ser baixado em: <https://bitbucket.org/gugawag/cdolly>.

²CATESTER pode ser baixado em <https://bitbucket.org/gugawag/catester>.

³CREXECUTOR pode ser baixado em <https://bitbucket.org/gugawag/creactoringexecutor>.

e 6 relacionados à mudança comportamental. Analisamos manualmente alguns refatoramentos e detectamos um bug de *Overly Strong Condition*, um bug de interface gráfica e um bug de transformação. Replicamos os bugs no NetBeans CND [31], Xrefactory [51] e OpenRefactory [17] e detectamos 29 bugs nessas ferramentas.

IDEs tais como Eclipse e NetBeans possuem automatizações que auxiliam o desenvolvedor nos refatoramentos, diminuindo a chance de erros causados por falhas manuais e reduzindo o tempo das transformações. Porém, essas mesmas IDEs podem introduzir bugs nos programas refatorados a partir de transformações que levam a erros de compilação e mudanças comportamentais. Soares et al. [41] propuseram uma técnica para testar ferramentas de refatoramento Java. Soares et al. avaliaram a técnica em 29 implementações de refatoramentos do Eclipse JDT, NetBeans e JRRT e detectaram 57 bugs relacionados a erro de compilação e 63 bugs relacionados a mudanças comportamentais. Nossa abordagem é inspirada na técnica de Soares et al. e teve como principal objetivo mostrar evidências de que a técnica funcionaria em outra linguagem de outro paradigma, o que na prática mostramos. Além disso, consideramos na teoria em Alloy corpo de função para testar refatoramentos comportamentais, diferentemente de Soares et al., que focaram em refatoramentos estruturais. Com essa abordagem, foi possível avaliar refatoramentos tais como *Extract Function*, *Extract Constant*, *Extract Local Variable* e *Rename Local Variable*, onde detectamos 28 bugs. Gligoric et al. [14] testaram ferramentas de refatoramento Java e C, porém detectaram apenas bugs relacionados a erro de compilação, por não terem um oráculo para mudanças comportamentais. Mostramos que nossa técnica detecta automaticamente bugs relacionados a erros de compilação e mudança comportamental. Além disso, analisamos manualmente alguns refatoramentos e vimos que podemos evoluir nossa técnica para detectar bugs de interface gráfica, bugs de transformação e *Overly Strong Condition*.

Overly Strong Conditions são condições fortes implementadas por ferramentas de refatoramento. Essas condições normalmente são definidas pelos desenvolvedores dessas ferramentas visando proteger o código de transformações erradas. Porém, elas impedem um refatoramento de ser aplicado. Durante a execução do refatoramento *Rename Parameter*, detectamos manualmente um bug de *Overly Strong Condition*. Ao tentar refatorar o nome de um parâmetro que foi referenciado dentro de diretivas condicionais (`#ifdef`), o CREXECUTOR retornou uma falha, informando a mensagem *The selected name could not be*

analyzed. Ao executar este refatoramento via interface do Eclipse, foi mostrada a mensagem *Found Problems: Refactoring contains 1 potencial match*, sendo possível, mesmo assim, continuar com o refatoramento. Pretendemos analisar esse tipo de bug através da avaliação dos resultados da aplicação de um refatoramento por duas IDEs distintas assim como foi feito por Soares et al. [43] utilizando a técnica de testes diferenciais [27].

Um bug de transformação é caracterizado por uma implementação errada das regras de um refatoramento. Ao analisar um dos códigos refatorados pelo refatoramento *Extract Constant*, detectamos um bug de transformação. Ao extrair um literal para uma variável, todos os literais semelhantes ao refatorado foram substituídos pela variável extraída. No Eclipse JDT, por exemplo, este bug não ocorre. Pretendemos estender a técnica para detectar automaticamente esse tipo de bug analisando a AST do código refatorado, já que o oráculo para detectá-lo foi manual.

O escopo delimita o espaço de estados no qual o Alloy Analyzer buscará soluções. Como exemplo do rápido crescimento do espaço de estados, executamos o metamodelo C com escopo 2 e foram gerados 8.856 programas, com taxa de compilação de 73% em 2 minutos (sem contar o tempo de compilação). Com escopo 3, foram gerados mais de 2.000.000 de programas em 7.5 horas. Paramos de executar com escopo 3 após cinco dias ininterruptos. Essa explosão de estados é facilmente explicada no CDOLLY, pois além das combinações de valores para assinaturas e relações do metamodelo C, é necessário testar todas as possíveis combinações de instruções (*statements*) dentro do corpo das funções. Essa é uma das diferenças do CDOLLY para o JDOLLY [41], já que este último gera métodos apenas com uma instrução. Pretendemos analisar em nossa técnica os fatores escopo e performance a partir da aplicação de técnicas de espaçamento de avaliação de falhas como a de Jagannath et al. [21].

Para a geração de programas com o CDOLLY, utilizamos escopo 3 para instruções de função e 2 para os restantes dos elementos. É um escopo bastante reduzido. De fato, alguns bugs foram detectados após a falha de uma pequena quantidade de programas refatorados. Por exemplo, ao renomear uma constante contendo o operador *, ocorreram 23 falhas, num universo de 10.500 programas. Se aumentarmos o escopo, a tendência é de aumentar os casos falhos e diminuir o fator sorte. Encontrar bugs em escopo reduzido é mais um indício da hipótese do escopo pequeno (*Small Scope Hypothesis*) [19], que diz que a maioria dos bugs têm contraexemplos pequenos, e que a maior parte dos bugs ocorre em menos de 5

entidades [2].

6.1 Trabalhos Futuros

Pretendemos comparar nossa técnica com outras técnicas de testes de implementações de refatoramento, tais como a de Gligoric et al. [14], a partir de experimentos. Nesse trabalho fizemos tal comparação informalmente.

Pretendemos evoluir nossa técnica para avaliar *Overly Strong Conditions* [43]. Para isso, usaremos o CATESTER como gerador de testes e aplicaremos a técnica de testes diferenciados [27] para avaliar diferenças nos resultados de ferramentas de refatoramentos. Se para um tipo de refatoramento uma IDE refatora um programa e uma outra não refatora, esta última apresenta um bug de *overly strong condition*.

Pretendemos incrementar a teoria Alloy de C do CDOLLY para gerar programas mais complexos, com construções do tipo `struct`, `array` e ponteiros visando aumentar a chance de detecção de bugs com mudanças comportamentais. Com essa teoria, pretendemos detectar automaticamente bugs de interface gráfica, bugs de transformação e *Overly Strong Condition*.

Após isso, pretendemos evoluir também nosso oráculo para detectar mais bugs de mudança comportamental através de lançamento de sinais. Como atualmente os programas gerados pelo CDOLLY são simples, não conseguimos detectar erros a partir de lançamento de sinal diferente de *segmentation fault*.

Pretendemos também analisar mais implementações de refatoramentos em outras IDEs, além de automatizar a avaliação nas ferramentas NetBeans, Xrefactory e OpenRefactory. Para isso, iremos evoluir o CREXECUTOR para automaticamente chamar a API de refatoramento dessas ferramentas.

Para geração de testes, pretendemos utilizar outros geradores automáticos de testes para C tais como API Sanity Checker [11]. Nos nossos estudos, não encontramos um gerador de testes que, a partir do código C, gere testes de unidade. O API Sanity Checker gera testes apenas de bibliotecas compartilhadas de C/C++.

Por fim, pretendemos otimizar nossa técnica de testes de ferramentas que implementam refatoramentos. Para isso, avaliaremos o ganho de mudar a ordem da geração dos testes para

depois da fase da aplicação dos refatoramentos. Além disso, pretendemos aplicar a técnica de saltos proposta por Jagannath et al. [21] e analisar seus efeitos na diminuição de tempo para detecção de bugs.

Bibliografia

- [1] Alloy. KodKod. At <http://alloy.mit.edu/kodkod/>, 2014.
- [2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". Technical report, In *POPL: Proceedings of the 29th ACM Symposium on the Principles of Programming Languages*, 2002.
- [3] Luciano Baresi and Matteo Miraz. Testful: automatic unit-test generation for Java classes. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, pages 281–284. ACM, 2010.
- [4] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for Object-Oriented programming. *Science of Computer Programming*, 52:53–100, August 2004.
- [5] A. Celentano, S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software: Practice and Experience*, 10(11):897–918, November 1980.
- [6] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the 15th Foundations of Software Engineering, ESEC-FSE*, pages 185–194. ACM, 2007.
- [7] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
- [8] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.
- [9] Eclipse.org. CDT Core Component. At <http://www.eclipse.org/cdt/core/>, 2014.

-
- [10] Eclipse.org. JDT Core Component. At <http://www.eclipse.org/jdt/core/>, 2014.
- [11] The Linux Foundation. Api sanity checker. At http://ispras.linuxbase.org/index.php/API_Sanity_Checker, 2014.
- [12] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Company, Inc., Boston, MA, USA, 1999.
- [13] Alejandra Garrido and Ralph E. Johnson. Refactoring c with conditional compilation. In *ASE*, pages 323–326. IEEE Computer Society, 2003.
- [14] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic Testing of Refactoring Engines on Real Software Projects. In *Proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP*, pages 629–653. Springer-Verlag, 2013.
- [15] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*, pages 225–234. ACM, 2010.
- [16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, PLDI*, pages 213–223. ACM, 2005.
- [17] Munawar Hafiz and Jeffrey Overbey. Openrefactory/c: An infrastructure for developing program transformations for c programs. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH*, pages 27–28. ACM, 2012.
- [18] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9:242–257, December 1970.
- [19] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

- [20] Daniel Jackson, Ian Schechter, and Hya Shlyachter. Alcoa: the Alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering, ICSE*, pages 730–733. ACM, 2000.
- [21] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the Costs of Bounded-Exhaustive Testing. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, FASE*, pages 171–185. Springer-Verlag, 2009.
- [22] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.
- [23] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, pages 371–372. ACM, 2010.
- [24] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
- [25] Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12, SSYM*, pages 9–9. USENIX Association, 2003.
- [26] Darko Marinov and Sarfraz Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE*, pages 22–34. IEEE Computer Society, 2001.
- [27] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.
- [28] Sun Microsystems. NetBeans IDE. At <http://www.netbeans.org/>, 2014.
- [29] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 2014.

-
- [30] Emerson Murphy-Hill and Andrew P. Black. Refactoring Tools: Fitness for Purpose. *IEEE Software*, 25(5):38–44, 2008.
- [31] NetBeans.org. NetBeans. At <http://netbeans.org/features/cpp/>, 2014.
- [32] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [33] OpenRefactory. OpenRefactory. At <http://www.openrefactory.org/>, 2014.
- [34] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA*, pages 87–96. ACM, 2008.
- [35] Carlos Pacheco, Shuvendu K. Lahiri, Michael Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering, ICSE*, pages 75–84. IEEE Computer Society, 2007.
- [36] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001.
- [37] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- [38] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge proposal: verification of refactorings. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification, PLPV*, pages 67–72. ACM, 2008.
- [39] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE*, pages 263–272. ACM, 2005.
- [40] Gustavo Soares. Making Program Refactoring Safer. In : *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (Student Research Competition)*, pages 521–522. ACM, 2010.

-
- [41] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.
- [42] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making Program Refactoring Safer. *IEEE Software*, 27:52–57, July 2010.
- [43] Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. Identifying overly strong conditions in refactoring implementations. In *Proceedings of the 27th IEEE International Conference on Software Maintenance, ICSM*, pages 173–182, Washington, USA, September 2011.
- [44] Gustavo Araújo Soares. Uma abordagem para aumentar a segurança em refatoramentos de programas. Master’s thesis, Universidade Federal de Campina Grande, 2010. Master Thesis at Federal University of Campina Grande.
- [45] Ian Sommerville. *Software Engineering*. International computer science series. Addison-Wesley, 6 edition, 2001.
- [46] Lance Tokuda and Don Batory. Evolving Object-Oriented Designs with Refactorings. *Automated Software Engineering*, 8:89–120, January 2001.
- [47] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009. AAI0821754.
- [48] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the International Conference on Software Engineering, ICSE*, pages 233–243. IEEE Press, 2012.
- [49] Mohsen Vakilian and Ralph E. Johnson. Alternate refactoring paths reveal usability problems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1–11, 2014.
- [50] Peter van der Linden. *Expert C Programming: Deep C Secrets*. Prentice Hall (SunSoft), 1994.

-
- [51] Xrefactory. Xrefactory. At <http://www.xref.sk/xrefactory/main.html>, 2014.
- [52] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 283–294. ACM, 2011.
- [53] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Survey*, 29:366–427, December 1997.

Apêndice A

Teoria em Alloy do CDOLLY

Código Fonte A.1: Metamodelo de subconjunto C em Alloy.

```
abstract sig Id {}

one sig Define {
  defs: set Id
}

abstract sig Variable {
  type: one Type-Void
}

sig LocalVar, GlobalVar extends Variable {}

sig Function {
  returnType: one Type,
  param: lone LocalVar,
  stmt: seq Stmt
}

fact voidFunctionNoReturn{
  all f:Function |
    f.returnType = Void => no f.stmt.elems & Return
}

fact noVoidFunctionLastStatementIsReturn{
```

```

all f:Function |
  f.returnType != Void => f.stmt.last in Return
}

fact noVoidFunctionJustOneReturn{
  all f:Function |
    f.returnType != Void => one f.stmt.elems & Return
}

fact noVoidFunctionVariableReturnedEqualsFunctionReturnType{
  all f:Function |
    f.returnType != Void => f.stmt.last.r.type = f.returnType
}

fact ifDefIdWasPreviouslyDefined{
  all x:ifDef | x.def in Define.defs
}

fact varAttributionInFunctionReferencesParameterOrDeclaredVariable{
  all f:Function |
    (all st1: f.stmt.elems | st1 in VarAttrib =>
      (some st2: f.stmt.elems |
        st2 in LocalVarDecl and st2.(LocalVarDecl<:v) = st1.(
          VarAttrib<:v)) or
          st1.(VarAttrib<:v) = f.param.v)
    )
}

fact varAttributionInIfDefReferencesParameterOrDeclaredVariable{
  all f:Function |
    (all st1: f.stmt.elems | st1 in ifDef =>
      (some st2: f.stmt.elems |
        st2 in LocalVarDecl and st2.(LocalVarDecl<:v) = st1.cmd.(
          VarAttrib<:v) and
          st2.(LocalVarDecl<:v) = st1.cmd.(VarAttrib<:v) and f.stmt.
            idxOf[st2] <
              f.stmt.idxOf[st1]) or (st1.cmd.(VarAttrib<:v) =
                f.param.v))
    )
}

```

}

fact varAttributionInElseReferencesParameterOrDeclaredVariable{

all f:Function |

 (**all** st1: f.stmt.elems | st1 **in** ifDef =>

 (**some** st2: f.stmt.elems |

 st2 **in** LocalVarDecl **and** st2.(LocalVarDecl<:v) = st1.else_.(

 VarAttrib<:v) **and**

 st2.(LocalVarDecl<:v) = st1.cmd.(VarAttrib<:v) **and** f.stmt.

idxOf[st2] <

 f.stmt.idxOf[st1]) **or** (st1.else_.(VarAttrib<:v)

= f.param.v))

}

fact noVarDeclarationWithSameNameParameter{

all f:Function |

 (**all** lvd:f.stmt.elems | lvd **in** LocalVarDecl =>

 (**some** p:f.param | lvd.(LocalVarDecl<:v) != p))

}

fact noSameNameToDistinctLocalVarDeclaration{

all f:Function |

 (**all** lvd:f.stmt.elems | lvd **in** LocalVarDecl =>

 (**some** lvd2:f.stmt.elems | lvd.(LocalVarDecl<:v) != lvd2.(
 LocalVarDecl<:v)))

}

pred optimization[] {

 Stmt **in** Function.stmt.elems + Function.stmt.elems.cmd + Function.stmt.
 elems.else_

 Type **in** Variable.type + Function.returnType

 LocalVar **in** Function.param + LocalVarDecl.v

all f:Function | #f.stmt < 5

some f:Function | #f.stmt > 0

all f:Function | not f.stmt.hasDups

}

```
abstract sig Stmt {}

sig Return extends Stmt {
  r: Variable
}

sig LocalVarDecl extends Stmt {
  v: LocalVar
}

sig VarAttrib extends Stmt {
  v: Variable
}

sig ifDef extends Stmt {
  def: Id,
  cmd: VarAttrib,
  else_: lone VarAttrib
}

abstract sig Type {}
lone sig Void extends Type {}
abstract sig BasicType extends Type {}
lone sig Integer_, Float extends BasicType {}

fact {
  optimization[]
}
```
