UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RODRIGO DOTTORI DE OLIVEIRA

REPARO AUTOMÁTICO DE PROGRAMAS USANDO CORRETUDE RELATIVA

RIO DE JANEIRO
2023

RODRIGO DOTTORI DE OLIVEIRA

REPARO AUTOMÁTICO DE PROGRAMAS USANDO CORRETUDE RELATIVA

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. João Carlos Pereira da Silva

RIO DE JANEIRO

2023

CIP - Catalogação na Publicação

O48r

Oliveira, Rodrigo Dottori de
Reparo Automático de Programas Usando Corretude Relativa / Rodrigo Dottori de Oliveira. -- Rio de Janeiro, 2023.
83 f.

Orientador: João Carlos Pereira da Silva.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2023.

1. Inteligência artificial. 2. Reparo automático de programas. 3. Bugs de software. I. Silva, João Carlos Pereira da, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

RODRIGO DOTTORI DE OLIVEIRA

REPARO AUTOMÁTICO DE PROGRAMAS USANDO CORRETUDE RELATIVA

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 12 de janeiro de 2023

BANCA EXAMINADORA:

Prof. João Carlos Pereira da Silva
D.Sc. (UFRJ)

Profa. Silvana Rossetto
D.Sc. (PUC-RJ)

Anamaria Martins Moreira
D.Sc. (INPG-França)

AGRADECIMENTOS

Agradeço primeiramente ao meu professor orientador pelo apoio, compreensão e direção. Também gostaria de agradecer aos membros da banca examinadora e a todo o Instituto de Computação.

Finalmente, agradeço ao professor Besma Khaireddine, da Universidade de Tunis El Manar, por disponibilizar o código utilizado no seu artigo (KHAIREDDINE; MARTINEZ; MILI, 2019). Esse trabalho não teria sido possível sem sua ajuda.

"É mais sobre 'bom o suficiente' do que sobre certo e errado."

James Bach

RESUMO

O reparo automático de programas consiste em identificar e consertar falhas em um código até que o mesmo passe a se adequar a uma especificação fornecida. Tal especificação é geralmente feita através de uma suíte de testes. Neste caso, um código é considerado absolutamente correto se passa em todos os testes. Ferramentas atuais nesse ramo fazem uso de técnicas como a programação genética, que se baseia em princípios da computação evolutiva, para construir programas corretos a partir de incorretos. Tais ferramentas são capazes de realizar consertos em diversos casos, mas têm sua utilidade limitada por se basearem em uma noção muito simples de corretude, onde programas são considerados apenas absolutamente corretos ou absolutamente incorretos, sem se levar em conta programas intermediários. Essa dificuldade aparece frequentemente em casos com múltiplas falhas. O presente trabalho propõe expandir o escopo de algumas das ferramentas existentes através de uma teoria de corretude relativa, na qual é possível determinar quais programas podem ser considerados estritamente mais corretos que outros. O uso dessa teoria permite a criação de um paradigma novo de reparo de programas focado em aprimoramentos graduais. O trabalho também relata estudos de caso que visam verificar que, com esse novo paradigma, há a possibilidade de consertar programas que não eram reparáveis pelos métodos originais.

Palavras-chave: inteligência artificial; reparo automático de programas; corretude relativa; corretude absoluta; bugs de software; remoção de falhas.

ABSTRACT

Automatic program repair consists in identifying and fixing faults in a code so that it follows a particular specification. This specification is usually done by means of a test suite. In these cases, a code is considered absolutely correct if it passes all the tests. Current repair tools make use of techniques such as genetic programming, which is based on the principles of evolutionary computation, to build correct programs starting from incorrect ones. These tools are capable of producing correct fixes in many cases, but their usefulness is limited due to basing themselves on an overly simplistic notion of correctness, where programs are considered either absolutely correct or absolutely incorrect, without taking intermediate programs into account. This problem frequently manifests in cases with multiple faults. This work attempts to expand the scope of certain existing tools by using a theory of relative correctness, in which it is possible to determine which programs are strictly more correct than others. The use of this theory enables the development of a program repair paradigm focused on gradual improvements. This work also describes case studies that attempt to verify that, through this new paradigm, it is possible to fix programs that were not previously repairable by the original methods.

Keywords: artificial intelligence; automatic program repair; relative correctness; absolute correctness; software bugs; fault removal.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Em média, desenvolvedores de software passam quase metade de seu tempo de trabalho consertando bugs, com aproximadamente 36% do tempo sendo dedicado à programação de funcionalidades novas (LATOZA; VENOLIA; DELINE, 2006). Em projetos grandes de código aberto, bugs demoram anos (em média) para serem consertados após sua introdução no código (EYOLFSON; TAN; LAM, 2011).

Sabendo o quão custoso o reparo de software pode ser, é de claro interesse investigar a possibilidade de automatizar essas tarefas. Tentativas de tratar o reparo de software como um problema formal e computável existem desde os anos 90, com o número de artigos sobre o tópico crescendo nas duas décadas seguintes (GAZZOLA; MICUCCI; MARIANI, 2019). A área ainda está em sua infância, mas tem presenciado grandes avanços nos últimos anos. Algoritmos primitivos que funcionam apenas em experimentos simples estão abrindo espaço para aqueles com aplicações práticas no mundo real, e os fundamentos teóricos que servem de base para muitos desses estão adquirindo mais rigor.

De forma básica, o problema do reparo de software consiste em achar um programa correto a partir de um incorreto, onde a corretude é definida por uma especificação fornecida inicialmente. Chamaremos de *falha* uma porção de código em um programa que pode ser substituída ou removida de forma a torná-lo mais próximo da versão correta (apresentaremos uma definição mais rigorosa em seções seguintes). O problema de reparo também pode ser definido como a eliminação sucessiva de falhas.

1.1 TRABALHOS RELACIONADOS

Há múltiplas maneiras de abordar esse problema. (GAZZOLA; MICUCCI; MARIANI, 2019) dividem métodos de reparo automático em duas categorias principais: orientado por semântica (*semantics-driven*) e geração e validação (*generate-and-validate*). Os métodos da primeira categoria criam um modelo formal do problema a ser resolvido e o usam como base. Exemplos de métodos de reparo orientados por semântica incluem o SEMFIX (NGUYEN et al., 2013) e o NOPOL (XUAN et al., 2017).

Os métodos da segunda categoria criam reparos possíveis a partir de transformações específicas e os testam até encontrarem uma solução, sendo o GenProg (GOUES et al., 2012), que usa um algoritmo evolutivo para aprimorar o programa incorreto de forma gradual, um dos principais. Os métodos JAFF (ARCURI, 2011), pyEDB (ACKLING; ALEXANDER; GRUNERT, 2011) e Marriagent (KOU; HIGO; KUSUMOTO, 2016) também fazem uso de algoritmos evolutivos para geração e validação, mas diferem em quesitos como a localização de falhas e a representação dos programas.

O framework Astor (MARTINEZ; MONPERRUS, 2019) é uma tentativa de generali-

zar as técnicas de geração e validação para reparo de programas em Java, apresentando um algoritmo geral cujos passos individuais podem ser modificados para implementarem técnicas já existentes ou para criar técnicas novas. Entre as técnicas padrão implementadas pelo Astor está o próprio GenProg (essa implementação específica é chamada de jGenProg).

Um framework geral diferente, o RCFix (KHAIREDDINE; MARTINEZ; MILI, 2019), busca corrigir certas inaptidões do Astor, principalmente quando se trata do reparo de programas com múltiplas falhas. Outro framework de geração e validação para Java é o ARJA (YUAN; BANZHAF, 2020). Enquanto o Astor busca ser o mais geral possível para possibilitar a implementação de vários métodos diferentes, o ARJA foca apenas em métodos evolutivos, e tem como objetivo estender a funcionalidade padrão dos mesmos, ao invés de apenas implementá-la. O ARJA também inclui uma versão própria do GenProg.

Para a avaliação dos métodos de reparo, existem uma série de bancos de programas extraídos do mundo real. O Defects4J (JUST; JALALI; ERNST, 2014), o BEARS (MADEIRAL et al., 2019) e o Bugs.jar (SAHA et al., 2018) são exemplos de bancos cujos programas são escritos exclusivamente em Java. O Bugswarm (TOMASSI et al., 2019) e o QuixBugs (LIN et al., 2017) incluem programas em Java junto com programas em outras linguagens.

1.2 PROPOSTA

A proposta desse trabalho é estender o RCFix, de forma a construir uma abordagem nova em cima dos princípios do Astor. Escolhemos o RCFix por acreditarmos que ele será bem sucedido em um conjunto de situações maior do que o Astor, e que será mais adequado a casos reais. Essa hipótese se baseia no fato do RCFix, diferentemente do Astor, fazer uso do conceito de corretude relativa, que será explicado mais adiante.

Mais especificamente, adaptamos o RCFix, que já em sua versão original faz uso de funções e classes do Astor, para usar as versões mais novas dessas funções e classes. Além disso, enquanto o RCFix originalmente implementa apenas um método de reparo (o GenProg), nossa versão implementa todos os métodos pré-definidos do Astor.

Usando nossa versão estendida, realizamos uma série de estudos de caso visando entender as diferenças entre esses dois frameworks, e as vantagens que o RCFix possui. Nossos estudos indicam que ele pode ser capaz de reparar programas com profundidade de falhas arbitrária e programas com múltiplos bugs simultâneos (incluindo casos onde os bugs requerem técnicas diferentes para serem consertados).

1.3 ESTRUTURA

Começamos detalhando, no capítulo 2, o funcionamento geral de algoritmos de geração e validação, usando o GenProg como nosso principal exemplo. Em seguida, apresentamos

o Astor em si.

No capítulo 3, apresentamos uma teoria de *corretude relativa* desenvolvida inicialmente por (DIALLO et al., 2017), e a usamos como base para introduzir o RCFix e sua funcionalidade, destacando alguns pontos que o diferem do Astor.

No capítulo 4, realizamos análises que apontam que o Astor tem dificuldades com situações onde um dado programa apresenta múltiplas falhas e exploramos as formas como o RCFix pode ajudar com essas dificuldades. Usando nossa versão estendida do RCFix, fazemos nossos estudos de caso utilizando o banco de programas Defects4J, da mesma forma que os autores originais dos dois frameworks.

2 REPARO AUTOMÁTICO DE PROGRAMAS ATRAVÉS DE GERAÇÃO E VALIDAÇÃO

Nesse capítulo, começaremos introduzindo o conceito de testes, que será fundamental para o funcionamento da maioria dos métodos de reparo. Em seguida, explicaremos o GenProg, uma técnica específica de geração e validação, para elucidar os conceitos básicos da geração e validação como um todo. Finalmente, com esses fundamentos em mão, iremos introduzir e discutir o framework Astor.

Para realizar o reparo automático, é necessário que o algoritmo usado seja capaz de representar o programa em si, assim como sua especificação. Essas representações podem depender de características que variam de acordo com a linguagem de programação usada no programa original. Portanto, frequentemente os métodos de reparo são voltados para linguagens específicas. No presente trabalho, a maioria das ferramentas discutidas é voltada para a linguagem Java.

2.1 TESTES E AVALIAÇÕES NA LINGUAGEM JAVA

Conforme já discutido, o reparo automático faz uso de uma especificação que define a corretude do programa a ser reparado. Tipicamente, se faz uso de suítes de teste (*test suites*): conjuntos de testes que representam condições que o programa deve obedecer para ser considerado correto.

Em Java, testes geralmente são implementados usando o framework JUnit [1], que é focado em testes de unidade (*unit tests*), que são aplicados a seções individuais de código, como classes ou funções. Um teste é capaz de avaliar o comportamento de uma função ou classe do código dentro de situações particulares, sem a necessidade de executar o código inteiro. Isso é feito principalmente através de *assertions*, que são funções pré-definidas que estabelecem as condições centrais do teste. Cada tipo de *assertion* é capaz de verificar uma característica particular dos parâmetros recebidos: por exemplo, a *assertion* `assertEquals(param1,param2)` verifica que os dois parâmetros recebidos são iguais; se não forem, o teste falha. Os parâmetros podem ser resultados de uma chamada de função feita de dentro do teste, permitindo que o resultado da *assertion* dependa do comportamento da função chamada.

Um exemplo simples de teste, retirado da biblioteca Commons-Math[2], é visto a seguir. Ele testa uma classe de matrizes diagonais, verificando que o número de colunas e linhas é igual à dimensão fornecida.

[1] https://junit.org/junit5/
[2] https://github.com/apache/commons-math

Código 1 – Teste testConstructor1() do Commons-Math

```
1 @Test
2 public void testConstructor1() {
3     final int dim = 3;
4     final DiagonalMatrix m = new DiagonalMatrix(dim);
5     Assert.assertEquals(dim, m.getRowDimension());
6     Assert.assertEquals(dim, m.getColumnDimension());
7 }
```

Testes mais complexos são possíveis. Por exemplo, o JUnit permite criar um teste que falha se o código em seu interior não gerar uma exceção específica. Ainda usando exemplos do Commons-Math, não é possível inverter uma matriz singular, então o comportamento esperado do programa nesse caso é gerar uma exceção. O teste a seguir vai passar apenas se essa exceção for gerada com sucesso.

Código 2 – Teste testInverseError() do Commons-Math

```
1 @Test(expected=SingularMatrixException.class)
2 public void testInverseError() {
3     final double[] data = { 1, 2, 0 };
4     final DiagonalMatrix diag = new DiagonalMatrix(data);
5     diag.inverse();
6 }
```

2.2 GERAÇÃO E VALIDAÇÃO E O MÉTODO GENPROG

Como motivação para o restante do capítulo, vamos primeiro apresentar o método GenProg (GOUES et al., 2012), um exemplo de método de geração e validação através de programação genética.

2.2.1 Programação genética

A programação genética (POLI; LANGDON; MCPHEE, 2008) é uma técnica de aprendizado de máquina (especificamente, um algoritmo evolutivo) onde se começa com programas candidatos (a população inicial) que são aprimorados em gerações sucessivas até se chegar em um programa final que atende requisitos desejados. Os programas de cada geração são avaliados de acordo com algum critério (que irá variar em cada caso), e uma seleção é feita entre os melhores para passar por certas modificações e em seguida servir como base para a geração seguinte. Implementações da programação genética costumam seguir o mesmo algoritmo básico, porém com variações entre si.

Programas são representados como árvores sintáticas abstratas (*abstract syntax trees* ou AST). As árvores são chamadas de abstratas pois focam nas partes da sintaxe do programa que representam de fato seu conteúdo e comportamento, ignorando detalhes como parenteses, espaços e separadores de *statements*. Não existe uma única maneira

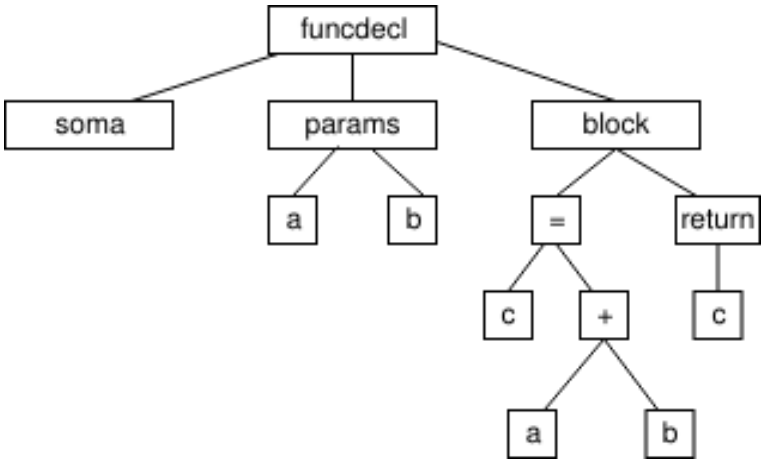
de representar um programa em forma de AST, e regras diferentes podem ser usadas em casos diferentes, adequando-se às necessidades de cada um. Consideremos, por exemplo, o programa a seguir (escrito em pseudocódigo):

Código 3 – Programa exemplo para programação genética

```
1 soma(a, b) {
2     c = a + b;
3     return c;
4 }
```

O programa acima poderia ser representado em forma de árvore da seguinte maneira:

Figura 1 – Representação em árvore do código 3



Os nós que formam as árvores sintáticas se separam em dois conjuntos: o conjunto de funções (*function set*) e o conjunto de terminais (*terminal set*). Terminais são as constantes e identificadores que compõem o programa, e são sempre as folhas da árvore. Funções abrangem todos os demais elementos do programa, como operações aritméticas ou palavras chave, e não são equivalentes ao conceito usual de função usado em programação. Na estrutura de AST acima, usamos funções como **funcdecl** que não existem explicitamente no código para preservar a estrutura e ordem do programa original. Consideramos que essa função é definida tal que ela terá sempre três filhos: um representando seu identificador, um representando seus parâmetros, e um representando o bloco em si que contém o código da função. Cada função do conjunto de funções terá "regras" desse tipo.

Tendo uma maneira de representar um programa em uma estrutura de dados, podemos apresentar a programação genética. Tipicamente, nos algoritmos evolutivos, a população inicial é gerada aleatoriamente. Essa possibilidade também existe na programação genética. Vários métodos são possíveis, mas (POLI; LANGDON; MCPHEE, 2008) destacam alguns específicos. Um deles é o método **full**, que escolhe aleatoriamente elementos do conjunto de funções e vai preenchendo a árvore, começando pela raiz, até atingir uma certa profundidade especificada pelo usuário. Ao atingir essa profundidade, terminais são

escolhidos para servirem como folhas. Esse método pode ser executado várias vezes para gerar uma população inicial de múltiplos indivíduos.

No caso específico da programação genética para reparo de programas, começaremos com uma população inicial baseada apenas no programa com falhas que queremos consertar. Assim sendo, normalmente não será necessário o uso de métodos como o **full**.

Tendo a geração inicial, podemos dar início ao laço principal da programação genética. Para cada geração, os indivíduos são avaliados a partir de uma função de aptidão (*fitness function*), especificada de tal forma que indivíduos de melhor avaliação são considerados mais próximos do programa final que buscamos, e portanto, são candidatos ideais para a geração seguinte. Uma seleção é feita entre os indivíduos com melhor avaliação. A função de aptidão escolhida dependerá fortemente do contexto onde o algoritmo será usado e quais características são desejáveis no programa final. Métodos específicos de programação genética como o GenProg geralmente terão uma função de aptidão pré-definida para seus propósitos, já que são voltados para contextos específicos.

Os indivíduos selecionados são em seguida candidatos a operações de *crossover* e mutação, que irão gerar os indivíduos novos que vão compor a próxima geração. Essas operações não ocorrem de forma uniforme entre todos os programas selecionados, mas sim de forma probabilística, de acordo com taxas pré-definidas (por exemplo, cada indivíduo pode ter uma chance de 99% de sofrer crossover e 1% de sofrer mutação).

No *crossover*, partes de programas diferentes são combinadas para criar programas novos (normalmente havendo apenas um filho resultante para cada operação, mas, em alguns casos, sendo mais adequado gerar dois filhos). O tipo de *crossover* realizado é geralmente o *crossover* entre subárvores, onde um nó é escolhido aleatoriamente em um dos indivíduos, e a subárvore que tem sua raiz nesse nó é substituída por uma subárvore aleatória do outro indivíduo.

Na mutação, uma porção aleatória do programa original é substituída por uma porção nova, que normalmente também é gerada de forma parcialmente aleatória. A mutação é geralmente a mutação de subárvores, onde se substitui uma subárvore do indivíduo por uma nova gerada aleatoriamente (de forma semelhante à geração de árvores usada no método **full** para geração da população inicial).

As operações de mutação e *crossover* são geralmente exclusivas, isso é, um indivíduo só vai ser alvo de uma delas. Se um indivíduo não for selecionado nem para mutação nem para *crossover*, é usada uma operação de reprodução, onde um indivíduo é selecionado e copiado para a próxima geração sem nenhuma modificação.

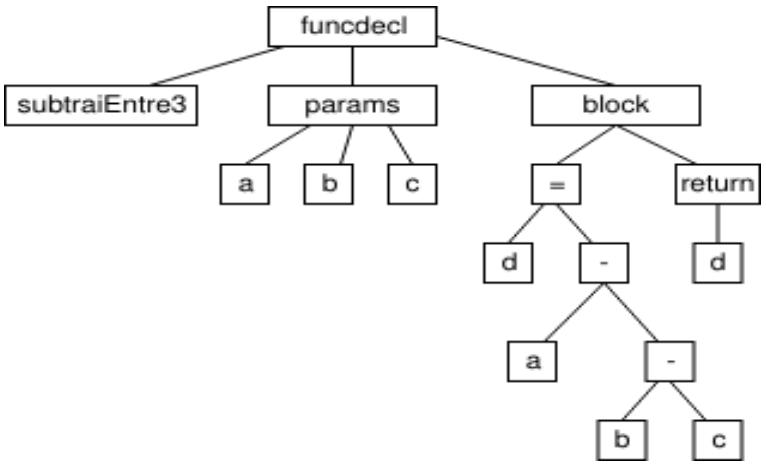
Uma vez executadas as operações de mutação, *crossover* e reprodução, temos uma geração nova definida, e voltamos ao processo de avaliação e seleção. O laço se repete até que se alcance um critério de parada: é comum especificar um número máximo de gerações junto com um critério de sucesso, como a criação de um indivíduo com um certo valor mínimo de aptidão. Independente do critério atingido, após o término do algoritmo,

o melhor indivíduo (ou `n` melhores indivíduos, em certos casos) é geralmente retornado como resultado.

Exemplo 2.1

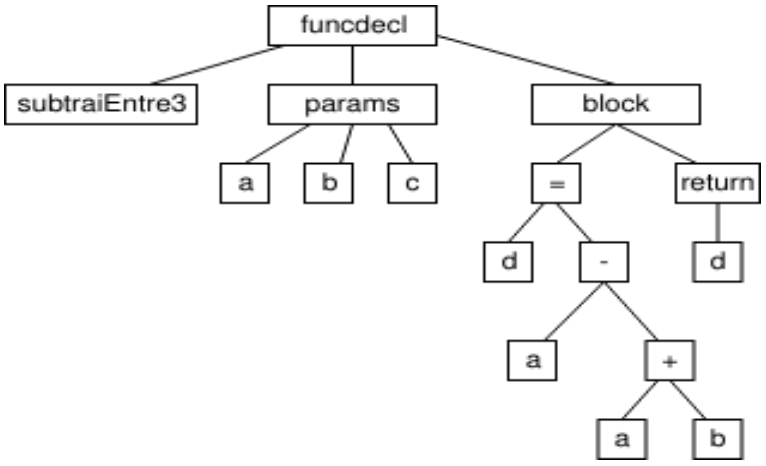
Vamos demonstrar o *crossover* entre o programa de exemplo já apresentado na figura 1 e o programa representado pela seguinte árvore:

Figura 2 – Programa base para crossover



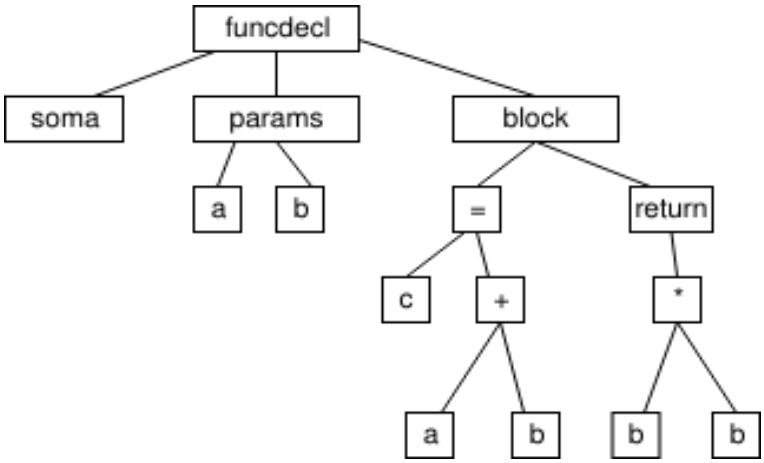
Vamos supor que o nó de subtração mais profundo da árvore da figura 2 terá sua subárvore substituída pela subárvore cuja raiz é "+" da figura 1. A árvore resultante será:

Figura 3 – Resultado do crossover entre as figuras 1 e 2



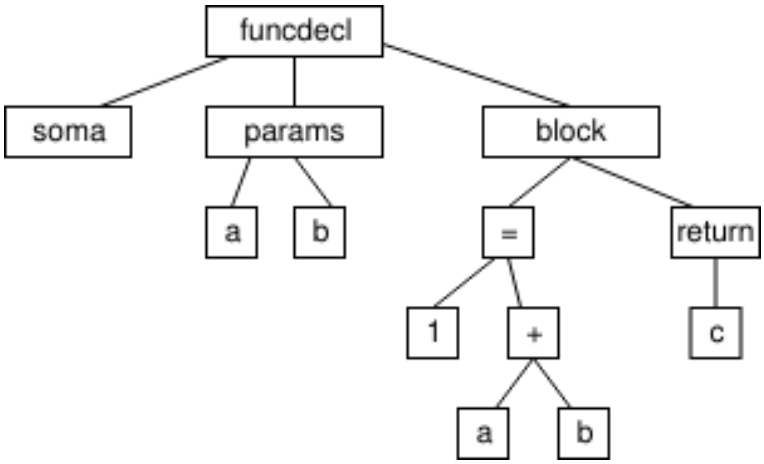
Para demonstrar a mutação, podemos imaginar o filho do nó de retorno na árvore da figura 1 substituído por uma árvore nova tal que o indivíduo resultante é:

Figura 4 – Mutação da figura 1



Notamos que nos casos acima, há substituições possíveis que resultariam em programas sintaticamente inválidos. A operação de mutação na árvore 1, por exemplo, poderia acabar substituindo o valor na esquerda do *assignment* por uma constante numérica:

Figura 5 – Outra mutação possível da figura 1



O programa acima estaria sintaticamente incorreto e não compilaria, pois tentaria atribuir um valor a uma constante numérica. Para evitar tais situações, (POLI; LANGDON; MCPHEE, 2008) recomendam que o conjunto de funções seja formulado de forma que nenhuma substituição de subárvore leve a um programa que gere erros. Entretanto, essa restrição pode limitar em demasia o conjunto de programas que podem ser usados no algoritmo.

Os autores recomendam duas soluções possíveis para esse problema. A primeira é redefinir as funções levando em conta erros possíveis. Por exemplo, é possível definir o operador de igualdade tal que, se o valor na esquerda for uma constante numérica, essa é automaticamente convertida para um nome de variável. Tal alteração afeta apenas a modelagem do programa como árvore, e não requer nenhuma modificação em seu código fonte ou linguagem.

A segunda solução seria estender as operações de *crossover* e mutação para que elas explicitamente levem em conta informações como o tipo dos valores (se o valor é constante, se é inteiro, etc.), tal que não seja possível gerar porções de programa incorretas. Por exemplo, a mutação pode descartar qualquer resultado que envolveria um valor constante na esquerda de um *assignment*). Outros erros, como o uso de variáveis fora de escopo ou operações ilegais (como divisão por 0), podem ser resolvidos de forma semelhante.

Na prática, entretanto, usos da programação genética para reparo automático podem optar por aceitar a geração de programas sintaticamente incorretos, usando apenas a função de aptidão para filtrá-los.

2.2.2 GenProg

O GenProg (GOUES et al., 2012) segue o modelo de programação genética apresentado acima, porém com algumas diferenças. Sendo um método voltado para o reparo de programas, a população inicial é baseada em um programa já existente. Além disso, o GenProg define os próprios operadores de mutação, e sua função de aptidão é baseada em uma suíte de testes que serve de especificação para o programa original. Um sistema de pesos também é usado para determinar as porções do programa que melhor servirão como candidatas à mutação. O uso do *crossover* é possível, porém opcional.

Apesar dessas diferenças, o GenProg segue a mesma estrutura da programação genética em geral: dada uma população inicial de programas, modificações pré-definidas são usadas para criar variantes, que são avaliadas de acordo com uma função de aptidão. Os passos são repetidos até que se chegue em um programa que passa em todos os testes, ou até que uma quantidade pré-determinada de recursos seja consumida.

O algoritmo começa com um programa que possui uma falha a ser consertada, representado por uma AST. Também são necessários, como entrada, os casos de teste que vão servir como especificação. Os testes fornecidos podem ser positivos (testes nos quais o programa inicial passa) ou negativos (testes nos quais o programa inicial não passa).

Pesos são associados aos *statements* do programa (categoria que, na definição dos autores do GenProg, abrange declarações de variável, atribuições, chamadas de função, laços, blocos e condicionais) dentro da árvore, dependendo dos casos de teste onde ocorrem. *Statements* que ocorrem em casos de teste negativos mas não em positivos possuem maior peso. Aqueles que ocorrem em ambos os tipos de caso possuem peso médio, e os que ocorrem apenas em casos positivos (ou nunca ocorrem em nenhum caso) possuem peso 0. Subsequentemente, os pesos são usados para identificar seções do código que têm maior chance de estarem atreladas ao problema do programa incorreto.

Os *statements* com peso maior que 0 formam uma sequência de nós que será chamada de caminho com pesos (*weighted path*). O caminho com pesos é chamado de caminho pois representa o "caminho" – isso é, a sequência de *statements* – percorrido pela execução do programa nos casos de teste negativos. Ele não constitui, necessariamente, um caminho

dentro da própria AST do programa. Por isso, vamos representar o *weighted path* como uma árvore adicional composta por pares ordenados `(statement, peso)`, com a mesma estrutura da AST.

A população inicial é construída a partir do programa original, aplicando-se operações de mutação no seu modelo em AST, com o resultado de cada mutação sendo um dos indivíduos dessa população, que terá um tamanho pré-definido.

Os autores definem três operações possíveis de mutação: (i) remover o *statement*, (ii) substituí-lo por um outro, ou (iii) inserir um novo logo em seguida. No caso de substituição ou inserção, os *statements* adicionais são retirados aleatoriamente de outras partes do programa, sem levar em conta o seu peso. Ao serem inseridos, os *statements* novos passam a ter o mesmo peso daquele que está sendo modificado. Qual das três operações será aplicada é decidido de forma completamente aleatória, com igual probabilidade para cada uma.

É importante destacar que, na prática, os *statements* "removidos" são na verdade trocados por *statements* em branco, para que a estrutura da AST seja mantida. Assim como os *statements* trocados, os *statements* em branco também possuem o mesmo peso daqueles que foram removidos. Além disso, na prática, inserções funcionam substituindo um *statement* existente por um bloco que consiste no *statement* antigo seguido do novo. Novamente, isso é feito para preservar a estrutura da AST.

A função de aptidão do GenProg, usada para avaliar os indivíduos de cada geração, é baseada na performance nos casos de teste. Especificamente, é definida como:

$$aptidao(P) = W_{PosT} * PosT + W_{NegT} * NegT \tag{2.1}$$

onde PosT e NegT correspondem, respectivamente, ao número de casos de teste positivos e negativos onde a variante executa com sucesso. Notamos que aqui, casos são classificados como positivos ou negativos baseado nos resultados do programa *original*, isso é, aquele que foi fornecido no início da execução do GenProg, não a variante que está sendo avaliada pela função. Ou seja, NegT é o número de casos que falham no programa original mas passam na variante que está sendo avaliada.

W_{PosT} e W_{NegT} correspondem a pesos pré-definidos que devem ser positivos. Os autores usam valores de 1 e 10, respectivamente, em seus testes, e também os usaremos no nosso exemplo. Variantes que não compilarem sempre terão aptidão igual a 0.

A seleção de quais indivíduos criados irão compor a geração nova é feita com base na função de aptidão. O método em si usado para a seleção pode variar e não é uma regra fixa do GenProg: um exemplo que pode ser usado é a amostragem estocástica universal (*stochastic universal sampling*), onde indivíduos têm uma probabilidade fixa de serem selecionados proporcional ao valor da sua função de aptidão. O número de indivíduos selecionados deve ser metade do tamanho da população.

Após a seleção, é feito o *crossover* entre os selecionados. Só *statements* com peso serão trocados durante o *crossover*. Uma função não especificada escolhe um ponto no *weighted path*; todos os *statements* com peso localizados após esse ponto são trocados entre as duas variantes pais, gerando duas variantes novas. No GenProg, o *crossover* sempre é realizado para todos os indivíduos de uma geração; entretanto, cada indivíduo só pode ser pai de um par de filhos.

Tanto os filhos quanto os pais serão partes da nova geração, trazendo-a para o tamanho de população definido inicialmente. Em seguida, cada indivíduo dessa nova geração sofrerá mutação. As regras de mutação serão as mesmos usadas e já exemplificadas na geração da população inicial. Com as mutações realizadas, teremos uma geração nova sobre a qual realizar novamente o processo de avaliação e seleção.

Os passos acima descrevem o laço básico do GenProg. O algoritmo irá terminar se o programa encontrar uma variante que passa em todos os casos teste fornecidos.

Exemplo 2.2

Vamos considerar a função a seguir, escrita em pseudocódigo.

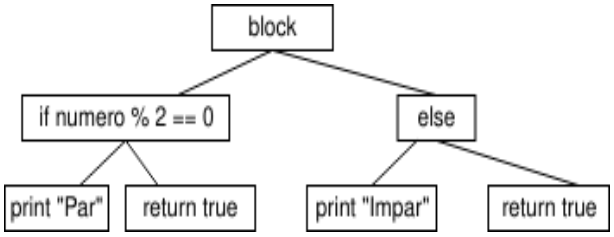
Código 4 – Programa exemplo para o GenProg

```
1 verificaParidade(numero) {
2     if numero % 2 == 0 {
3         print("Par");
4         return true;
5     }
6     else {
7         print("Impar");
8         return true;
9     }
10 }
```

Se considerarmos que a função deve retornar **true** se e somente se o número fornecido como parâmetro for par, ela está claramente incorreta, pois retorna **true** independente do número.

A função pode ser representada pela seguinte AST:

Figura 6 – AST da função verificaParidade()



Diferente das AST vistas na apresentação da programação genética, as AST usadas pelo GenProg são montadas no nível de *statements*, então elementos como declarações de função são desconsiderados.

Agora vamos imaginar que queremos testar a função com o seguinte conjunto de testes:

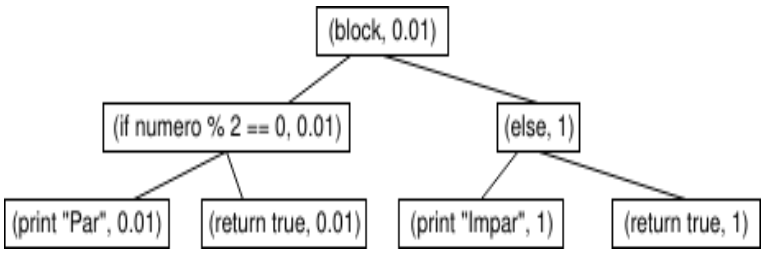
Código 5 – Conjunto de testes para ilustrar o GenProg

```
1 teste1() {
2     Assert.assertEquals(true, verificaParidade(2));
3 }
4
5 teste2() {
6     Assert.assertEquals(false, verificarParidade(3));
7 }
```

Claramente, para o programa original, o teste1 será bem sucedido (positivo) e o teste2 não (negativo). O GenProg rodará ambos os testes, verificando quais *statements* do programa original serão visitados em cada um. No teste1, apenas o primeiro bloco condicional será visitado; no teste2, ambos serão. Qualquer *statement* visitado na execução de um teste negativo tem seu peso inicialmente definido como 1, enquanto *statements* visitados tanto em casos negativos como positivos têm seu peso definido pela constante W_{Path}, que é um dos parâmetros do programa. (GOUES et al., 2012) recomendam um valor de 0.01 para esse parâmetro, que é o que vamos usar nesse exemplo. Como já dito anteriormente, *statements* não visitados em nenhum caso de teste negativo têm peso 0.

Assim, o *weighted path* da árvore será:

Figura 7 – Caminho com pesos para a função verificaParidade()

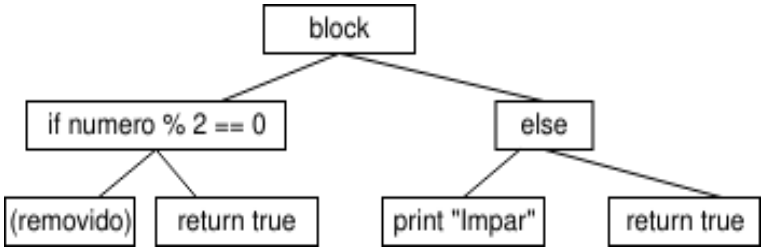


A população inicial será construída aplicando mutações ao programa original. Para simplificar, vamos supor um tamanho de população igual a 4 para o nosso exemplo. Todos os *statements* são candidatos a mutação com probabilidade proporcional a seu peso, porém obedecendo a taxa de mutação pré-definida, que será geralmente baixa ((GOUES et al., 2012) usam valores de 0.03 e 0.06 em seus testes). Na prática, um *statement* será modificado se primeiro um número aleatório em (0,1) for menor que o seu peso, e se em seguida um outro número aleatório em (0,1) for menor que a taxa de mutação. Em casos como o do nosso exemplo, onde temos só alguns *statements*, uma taxa de mutação de 0.03 ou 0.06 seria muito baixa e os programas novos gerados raramente apresentariam qualquer mudança; porém, quase sempre aplicaremos o GenProg em programas muito maiores, onde isso não será um problema.

Entretanto, como no nosso caso as taxas seriam baixas demais para gerar uma população distinta o suficiente, vamos supor que nossa população inicial consiste em quatro

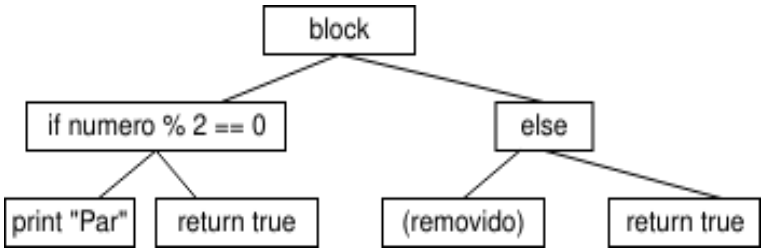
variantes do programa original: duas resultando da remoção do *statement* `print("Par")` e duas da remoção do *statement* `print("Impar")`. A árvore resultante no primeiro caso será:

Figura 8 – Primeira variante do verificaParidade()



E no segundo:

Figura 9 – Segunda variante do verificaParidade()



A geração nova é criada a partir dos resultados da função de aptidão. Variantes que não compilarem sempre terão aptidão igual a 0. No caso do nosso exemplo, os indivíduos da população inicial são iguais ao programa original porém com *statements* de print removidos, o que não vai afetar seu comportamento nos testes; cada um vai passar em um teste positivo e um teste negativo. Assim, todos terão o mesmo valor de aptidão:

$$aptidao(P) = W_{PosT} * PosT + W_{NegT} * NegT = 1 * 1 + 10 * 1 = 11 \tag{2.2}$$

Supomos que estamos usando a amostragem estocástica universal (isso é, a probabilidade do indivíduo ser selecionado é proporcional ao valor da sua função de aptidão). Sendo os valores de aptidão todos iguais, a seleção será completamente aleatória. Vamos supor que a seleção irá consistir em um dos programas que teve o `print("Par")` removido e um que teve o `print("Impar")` removido.

O *crossover* é feito em seguida. No nosso exemplo, os *weighted paths* das árvores que farão parte do *crossover* são:

Figura 10 – Caminho com pesos do primeiro ingrediente de crossover

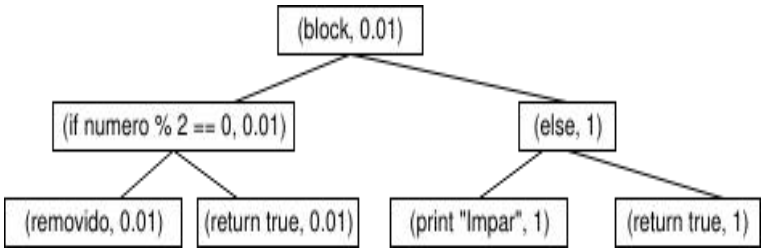
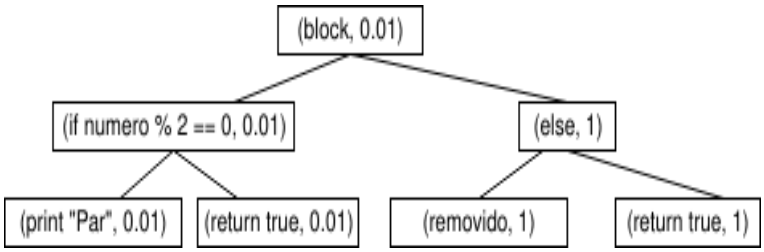


Figura 11 – Caminho com pesos do segundo ingrediente de crossover



Como as árvores ainda não sofreram nenhuma mudança significativa, no caso acima, qualquer troca entre elas irá somente resultar em cópias das duas árvores pai. Isso é, iremos gerar um filho igual ao primeiro pai e um igual ao segundo.

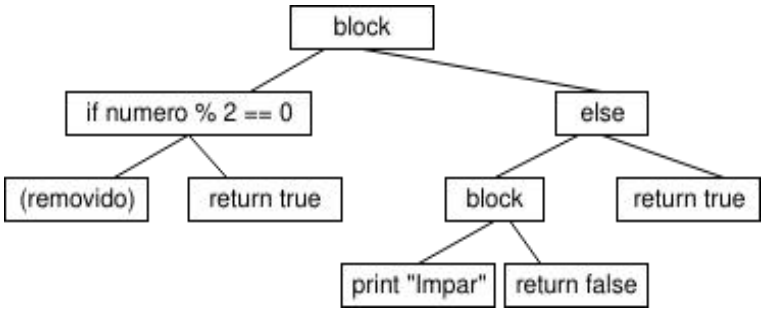
Os dois filhos e dois pais serão partes da nova geração. Cada indivíduo sofrerá mutação novamente, e teremos uma geração nova sobre a qual realizar novamente o processo de avaliação e seleção.

É evidente que, para que o nosso programa passe nos testes, um reparo natural seria substituir o segundo `return true` por um `return false`. Devido ao tamanho do nosso programa de exemplo, não existe nenhum ingrediente no código considerado até agora que resolva esse problema automaticamente. Mas se considerarmos a nossa função como apenas parte de um programa maior (como será em quase todos os casos práticos), é razoável supor a existência de um `return false` em outras partes do código que poderá ser usado.

Tendo esse ingrediente disponível, é possível reparar o programa ou realizando uma operação de substituição no `return true`, ou uma operação de inserção no *statement* anterior. Assim, é suficiente para o reparo do programa que, em uma operação de mutação, o *statement* correto seja selecionado para ser modificado (ou o `return true` ou o anterior); a operação correta seja selecionada em seguida (inserção ou substituição, dependendo do *statement* que será mutado); e finalmente, que o ingrediente correto seja escolhido.

Se considerarmos que uma inserção foi realizada na árvore cujo `print "Impar"` não foi removido, a AST do programa final será:

Figura 12 – AST do programa final



Representando o programa:

Código 6 – Programa final após execução do GenProg

```
1 verificaParidade(numero) {
2     if numero % 2 == 0
3         return true;
4     else {
5         {
6             print("Impar");
7             return false;
8         }
9         return true;
10    }
11 }
```

O programa acima passará no teste2, pois o `return false` será executado antes do `return true` se o bloco `else` for executado. Notamos que todos os testes serão sempre refeitos, pois a correção de um erro pode acabar introduzindo um erro novo. Como o programa também irá passar no teste1, sabemos que ele está correto.

Notamos que, em algumas linguagens, a presença de porções de código inalcançáveis – como o `return true` no bloco `else` acima – pode gerar um erro de compilação; o próprio Java está entre essas linguagens. Como estamos tratando nossos exemplos como pseudocódigo, vamos ignorar essa possibilidade, mas em um caso prático, esse fator pode limitar os caminhos possíveis para o reparo.

O código acima está correto de acordo com a nossa especificação, pois passa em todos os testes. Destacamos, entretanto, que enquanto a função original imprimia `"Par"` caso o número fosse par, essa funcionalidade foi perdida, o que provavelmente não está dentro do comportamento esperado da função, considerando o código inicial. Sem uma especificação adequada, entretanto, o GenProg não é capaz de considerar essa funcionalidade como importante e irá facilmente gerar modificações desnecessárias deste tipo, devido à natureza aleatória do algoritmo. Além disso, a legibilidade do programa pode acabar sendo afetada pelas modificações realizadas.

Para reduzir o impacto de tais modificações, o passo final do GenProg é a chamada

minimização de reparo. Se considerarmos o programa final como resultado de uma série de mudanças granulares, podemos avaliar cada mudança verificando se o comportamento em relação aos testes de caso será afetado com sua remoção ou não. Se não, podemos considerar a mudança como supérflua e excluí-la.

Essa série de testes não é feita para cada combinação possível de mudanças, pois isso seria muito custoso. No lugar disso, (GOUES et al., 2012) usam uma técnica chamada *delta debugging* para realizar esse processo de forma eficiente. O *delta debugging* é um algoritmo que determina, dentre um conjunto de mudanças em um programa, o menor subconjunto delas que resulta em falhas. Uma explicação do algoritmo não seria relevante para o presente trabalho, mas pode ser encontrada em (ZELLER, 1999).

Se bem sucedido, o resultado final do GenProg será um patch em formato **diff** contendo as mudanças necessárias para transformar o programa original em um programa correto, isso é, o último programa gerado pelo algoritmo.

2.2.3 Discussão

A motivação principal do GenProg é adaptar a técnica de programação genética para a área de conserto automático de falhas. O método não é, entretanto, a única forma possível de se usar essa técnica com essa categoria de problemas. O método JAFF (ARCURI, 2011), como ilustração, adapta a função de aptidão para considerar o quão próximo cada teste negativo chega da resposta esperada.

Consideremos, por exemplo, o caso de um teste como assertEquals(sum(1,1),2), que verifica se o resultado do primeiro parâmetro (uma função que simplesmente calcula a soma 1+1) é igual ao do segundo. Uma implementação da função sum() que dá a resposta de sum(1,1) como 3 pode ser considerada melhor do que uma implementação que daria a resposta como 100 (no sentido de estar mais próxima da resposta correta, isso é, 2). Além disso, o JAFF possui operadores de mutação distintos do GenProg, baseados na representação do programa em forma de árvore. As operações possíveis incluem, por exemplo, trocar sub-árvores de lugar, trocar a sub-árvore do pai de um nó pela sub-árvore cuja raiz é o próprio nó, ou inserir um outro nó aleatório entre um nó e seu pai.

Ambos os métodos têm uma característica em comum: inserções ou substituições de código fazem uso de porções de código já existentes no programa a ser consertado. Os autores do GenProg justificam isso com a intuição que, em muitos casos, a solução para um problema é comum o suficiente que a mesma já deve ter sido usada em outro lugar do programa (GOUES et al., 2012). Por exemplo, um erro pode decorrer da falta de um *null check*, que verificaria se uma variável tem valor nulo antes de usá-la; se a variável for usada em outros contextos, a chance de já existir um *null check* no restante do programa é alta.

Entretanto, fica claro que restringir o algoritmo a usar apenas pedaços de código já existentes limita a quantidade e a forma dos reparos possíveis. O GenProg só pode criar

programas que resultem da aplicação sucessiva dos operadores de mutação no programa original fornecido; é possível que todas as soluções para o problema se encontrem fora desse conjunto. Se considerarmos que o espaço de programas possíveis é teoricamente infinito, porém, restrições desse tipo se tornam claramente necessárias para que o reparo seja viável. A programação genética e o uso de funções de aptidão bem modeladas são exemplos de ferramentas que podem auxiliar a navegar o espaço de programas de forma guiada e racional. Entretanto, como tudo o que sabemos sobre a funcionalidade correta do programa é derivado apenas do texto do programa original e dos resultados dos testes, achar automaticamente uma versão correta envolverá uma certa quantidade de tentativa e erro.

Assim, maximizar o número de reparos possíveis também envolve maximizar o espaço a ser navegado (e, por conseguinte, o tempo levado para achar uma solução para o problema). Na prática, um dado método de reparo terá tipos de falha específicos ao qual se adequará melhor. Portanto, como tipos de falha diferentes demandarão métodos diferentes, não existem métodos estritamente melhores que os demais em todas as situações, por mais que apresentem melhor custo-benefício entre eficiência (isso é, a facilidade e rapidez com que se encontra uma solução) e escopo de reparo no geral; a disponibilidade de múltiplos métodos diferentes é, então, desejável.

2.3 VARIAÇÕES POSSÍVEIS E O MODELO ASTOR

Na última seção, consideramos o problema de reparo de programas como uma questão de navegar um *espaço de programas* contendo todas as variantes possíveis de um programa inicial a ser reparado, de forma a chegar, dentro de um tempo viável, em um programa final que passa em todos os casos de teste fornecidos. Essa modelagem é especialmente apta para a categoria de métodos de geração e validação, e pode servir como base para um *framework* genérico de métodos deste tipo.

O modelo Astor (MARTINEZ; MONPERRUS, 2019), voltado para programas em Java, segue e desenvolve essa linha de raciocínio de forma a generalizar o reparo de programas por geração e validação. Trata-se de um algoritmo genérico onde, para certos passos gerais (isso é, passos que os diferentes algoritmos de geração e validação têm em comum), o usuário pode definir uma regra específica a ser seguida.

Por exemplo, todos os algoritmos generalizados pelo Astor possuem um passo de mutação, onde é selecionado um dentre múltiplos operadores possíveis para modificar uma porção específica do programa. Isso de dá mesmo em casos onde o algoritmo não faz uso da programação genética ou de técnicas de computação evolutiva. Já vimos que no GenProg, os operadores são os de inserção, substituição e remoção, e que a seleção entre os mesmos é feita de forma completamente aleatória. O Astor permite que o usuário escolha quais operadores serão possíveis em uma dada execução do algoritmo, e como eles serão

selecionados.

Assim, por exemplo, poderíamos criar um algoritmo semelhante ao GenProg, porém apenas com os operadores de inserção e substituição, e onde a escolha entre eles é aleatória com pesos, tal que o operador de inserção tenha uma chance de 80% de ser escolhido, e o de troca, 20%.

Os passos cujo comportamento específico pode ser definido pelo usuário são os chamados pontos de extensão (*extension points*) do Astor, que possui doze no total. A regra escolhida em cada ponto de extensão define como o Astor navegará o espaço de programas possíveis.

O Astor, por padrão, já providencia algumas regras prontas a serem aplicadas em cada ponto, mas também permite que usuários programem suas próprias, tornando possível que o algoritmo acomode um grande número de métodos. Assim, o próprio GenProg pode ser visto como apenas uma combinação possível de regras do Astor, com algumas diferenças a mais que serão destacadas adiante.

2.3.1 Descrição do algoritmo e seus pontos de extensão

De forma resumida, o algoritmo do Astor consiste nos seguintes passos:

a) Localização de linhas suspeitas no programa original

b) Geração de pontos de modificação (porções de código que podem ser mudadas) a partir das linhas suspeitas

c) Laço de navegação do espaço de programas possíveis, que consiste em:
   - Escolher um ponto de modificação
   - Aplicar uma mutação, possivelmente fazendo uso de ingredientes
   - Avaliar e validar o programa novo gerado

d) Ordenação de programas válidos gerados

Os métodos usados para localizar linhas suspeitas, gerar pontos de modificação, navegar o espaço de programas, etc, são definidos pelos pontos de extensão do Astor. Explicaremos o funcionamento do algoritmo em detalhe a seguir.

Como o GenProg, o Astor recebe como entrada um programa a ser consertado e um conjunto de testes. O primeiro passo do algoritmo é a localização de falha, onde são identificadas linhas de código que potencialmente são responsáveis pelas falhas do programa a ser consertado. Essa identificação é feita a partir dos resultados dos testes fornecidos, associando-se a cada linha um valor de suspeita (*suspiciousness value*). O primeiro ponto de extensão do Astor, EP_FL, define o método usado para gerar tais valores a partir do código original e dos resultados de teste. No GenProg, por exemplo, já vimos que pesos diferentes são atribuídos baseado na presença de um dado *statement* na execução de testes negativos e positivos (é importante destacar que, em contraste, o

Tabela 1 – Resumo dos pontos de extensão do Astor

Sigla	Ponto de extensão	Descrição	Exemplos
EP_FL	Localização de falhas	Define o método usado para identificar linhas de código potencialmente responsáveis pelas falhas	GZoltar
EP_MPG	Granularidade dos pontos de modificação	Define granularidade dos pontos de modificação	Statements, expressões, operadores
EP_OD	Espaço de operadores	Define os operadores de mutação que podem ser aplicados aos programas	Inserção, remoção ou substituição de statements
EP_OS	Seleção dos operadores	Define como será selecionado o operador a ser aplicado na hora da mutação	Aleatório uniforme
EP_IPD	Definição do pool de ingredientes	Define quais ingredientes podem ser usados na mutação, se necessário	Porções de código retirados do mesmo arquivo da porção a ser mutada
EP_IS	Seleção de ingredientes	Define como será selecionado o ingrediente na hora da mutação	Aleatório uniforme
EP_IT	Transformação de ingredientes	Define quais transformações podem ser aplicadas aos ingredientes antes de serem aplicados	Trocar variáveis fora de escopo por variáveis aleatórias dentro do escopo
EP_PV	Validação de programas	Define quais medidas serão usadas para avaliar cada programa gerado	Número de testes em que o programa gerado passa
EP_FF	Função de aptidão	Define a função que avaliará o programa, baseado nas medidas obtidas no EP_PV	Número absoluto de testes aprovados
EP_NS	Estratégia de navegação	Define como será navegado o espaço de programas possíveis	Estratégia evolutiva
EP_MPS	Seleção de ponto de modificação	Define como será selecionado o ponto a ser modificado na hora da mutação	Aleatório uniforme
EP_SP	Priorização de solução	Ordena programas válidos gerados	Programas retornados na ordem cronológica em que foram gerados

Astor realiza a localização de falhas sempre em cima de linhas, e não de *statements*). Os valores de suspeita estão necessariamente entre 0 (a probabilidade mais baixa da linha em questão ser responsável pela falha) e 1 (a probabilidade mais alta).

Baseado nos resultados da localização de falhas, o algoritmo cria pontos de modificação (*modification points*), isso é, define as porções do código que serão candidatas a mutação (destacamos que mutação se refere ao passo onde uma porção do programa é de fato modificada, independente do método usado ser evolutivo ou não). O ponto de extensão EP_MPG decide a *granularidade* de cada ponto de modificação; isso é, qual parte de uma linha do código será de fato alvo de mutação. O algoritmo pode modificar os *statements* contidos na linha (se houver), mas podemos escolher, também, modificar apenas as expressões contidas na linha, ou somente expressões com operadores unários, entre várias outras opções possíveis. A escolha da granularidade é feita apenas uma vez e se aplica para o algoritmo inteiro. Assim, uma dada execução do Astor irá aplicar suas modificações sempre em *statements*, ou sempre em expressões, etc. No GenProg, como já vimos, trabalhamos no nível de *statements*.

As mutações possíveis são definidas através de operadores de reparo (*repair operators*), implementados no ponto de extensão EP_OD. Os operadores possíveis são limitados pela granularidade dos pontos de modificação. Podem representar transformações simples como, por exemplo, trocar um $>$ em uma expressão por um $>=$ (uma transformação diferente das disponíveis no GenProg); por outro lado, podem envolver uso de informações adicionais selecionadas no momento da mutação, chamadas de *ingredientes*. O GenProg é um exemplo de reparo através de ingredientes, onde os ingredientes são porções de código retiradas de outras localizações. Já vimos que os operadores possíveis do GenProg são os de inserção, remoção e substituição.

O processo de decisão entre as operações possíveis na hora de mutação é especificado pelo ponto de extensão EP_OS (no GenProg, o operador escolhido é completamente aleatório).

Quando necessário, o Astor cria um *pool* de ingredientes usando a estratégia definida no ponto de extensão EP_IPD, e no momento de mutação, o ingrediente é selecionado do *pool* usando a estratégia definida no ponto de extensão EP_IS. Tomando o GenProg como exemplo, a estratégia de construção do *pool* envolveria identificar e guardar todos os *statements* do programa, e posteriormente a estratégia de seleção escolheria entre os mesmos de forma completamente aleatória, com probabilidade igual para todos os ingredientes.

O Astor permite, ainda, que transformações sejam realizadas nos ingredientes antes dos mesmos serem usados, através do ponto de extensão EP_IT. Aplicar transformações pode ser útil em casos onde, por exemplo, o código usado como ingrediente faz uso de uma variável que estaria fora de escopo após a inserção no código novo. Uma transformação possível é trocar tais variáveis por variáveis aleatórias que estejam dentro do escopo.

Tendo os pontos de modificação (definidos por EP_FL e EP_MPG) e as mudanças possíveis que podem ser realizadas nesses pontos (definidas por EP_OD, EP_OS, EP_IPD, EP_IS e EP_IT), temos o conjunto de variantes possíveis que o algoritmo pode gerar a partir do programa inicial fornecido. Essas variantes compõem o *espaço de busca* que será navegado pelo Astor. Com esse espaço definido, os demais pontos de extensão descrevem *como* o mesmo será navegado pelo algoritmo.

Uma vez que um código novo é gerado através de mutação, ele passa por um processo de avaliação. Esse processo é especificado pelos pontos de extensão EP_PV e EP_FF. EP_PV define como serão geradas as medidas que em seguida servirão de base para a avaliação do código. Tipicamente, uma variante é avaliada apenas baseada em quantos dos testes fornecidos são bem sucedidos. Porém, métodos alternativos podem envolver a geração de testes adicionais, ou o uso de medidas diferentes no lugar do número de testes sucedidos ou falhados, como, por exemplo, o consumo de recursos durante a execução do programa. EP_FF define a função de aptidão, que vai fazer uso das medidas geradas pelo EP_PV para associar ao programa um valor indicando sua qualidade; um exemplo de função de aptidão já foi visto durante a explicação do GenProg.

O ponto de extensão EP_NS define a estratégia de navegação em si do espaço de busca. Isso é, define como os resultados de EP_PV e EP_FF influenciarão na geração de variantes adicionais, quando essa geração deve ocorrer, quais variantes geradas serão usadas como base para a geração seguinte, e quando o algoritmo deve parar. A programação genética é por si só uma estratégia de navegação, definindo o uso da função de aptidão para seleção de candidatos a mutação. Uma vez que uma variante é escolhida como próxima a sofrer mutação, o ponto de extensão EP_MPS define como escolher o ponto de modificação dentro da mesma que sofrerá o reparo (podendo ser, por exemplo, completamente aleatório, ou proporcional ao valor de suspeita).

Finalmente, chegando-se no fim da navegação, o Astor também providencia um ponto de extensão EP_SP, que permite que os programas válidos gerados (na forma de *patches*) sejam ordenados entre si ao serem retornados no final da execução, já que o Astor pode gerar múltiplos programas válidos. Por padrão, são retornados na ordem cronológica em que foram gerados.

2.3.2 Exemplo de método possível no Astor

Para demonstrar os tipos de métodos adicionais que podem ser implementados com o Astor, vamos ver nessa seção a execução do algoritmo usando regras distintas das do GenProg. Queremos consertar o programa a seguir:

Código 7 – Programa de exemplo a ser consertado pelo Astor

```
1 relacao(a,b) {
2     if a != b:
3         return "Igual";
4     else if a > b:
5         return "Maior";
6     else:
7         return "Menor";
8 }
```

Queremos que o programa acima retorne a string que descreve corretamente a relação entre os números a e b. Assim sendo, a primeira condição está claramente errada. Vamos representar nossa especificação com os seguintes testes:

Código 8 – Testes do programa de exemplo

```
1 teste1() {
2     Assert.assertEquals("Igual", relacao(1,1));
3 }
4
5 teste2() {
6     Assert.assertEquals("Maior", relacao(3,1));
7 }
8
9 teste3 () {
10     Assert.assertEquals("Menor", relacao(3,9));
11 }
```

É fácil ver que todos os testes serão negativos.

Para consertar o programa, definiremos um método novo através de uma seleção específica de parâmetros em cada ponto de modificação.

O primeiro passo do Astor é a localização de falhas, cujo método é definido pelo ponto de extensão EP_FL. A localização de falhas é uma área crucial no reparo de falhas e costuma se usar métodos muito mais complexos do que o definido para o GenProg. Para simplificar nosso exemplo, vamos definir um método *ad hoc* para ser usado nessa seção: atribuímos um valor de suspeita de 1 para todos as linhas de código que sejam visitadas em qualquer teste, sendo positivo ou negativo.

O Astor começará atribuindo valores de suspeita para todas as linhas do programa. No nosso exemplo, o teste1 irá visitar todas as linhas exceto as linhas 3 e 5. teste2 e teste3 irão visitar a linha 3. Assim, todas as linhas terão valor de suspeita igual a 1, exceto a 5, que terá valor 0.

O ponto de extensão EP_MPG irá definir a granularidade dos pontos de modificação. Escolheremos trabalhar no nível de expressões binárias cujos operadores são relacionais (>, >=, ==, !=, <. <=). Assim, após identificar o valor de suspeita de cada linha, o Astor irá criar pontos de modificação a partir de seções do código que tenham a granularidade

Tabela 2 – Pontos de extensão do nosso método personalizado

Ponto	Descrição	Opção escolhida
EP_FL	Localização de falhas	Personalizado (definido abaixo)
EP_MPG	Granularidade dos pontos de modificação	Expressões binárias com operadores relacionais
EP_OD	Espaço de operadores	Troca operador relacional por outro
EP_OS	Seleção dos operadores	Aleatório uniforme
EP_IPD	Definição do pool de ingredientes	Não utilizado
EP_IS	Seleção de ingredientes	Não utilizado
EP_IT	Transformação de ingredientes	Não utilizado
EP_PV	Validação de programas	Testes aprovados e reprovados
EP_FF	Função de aptidão	Número de testes falhados (0 é melhor possível)
EP_NS	Estratégia de navegação	Seletiva
EP_MPS	Seleção de ponto de modificação	Aleatório uniforme
EP_SP	Priorização de solução	Não utilizado

escolhida. No nosso caso, apenas as linhas 2 e 4 possuem tais expressões: `a != b` e `a > b`. Esses serão os únicos pontos de modificação de nosso código.

Finalmente, definimos os operadores possíveis no ponto de extensão EP_OD. Como já dissemos, os operadores dependem da granularidade dos pontos de modificação. Como tratamos apenas de expressões binárias relacionais, faz sentido definir operadores de mutação que se apliquem especificamente a esse tipo de expressão. Cada um de nossos operadores de mutação fará uma troca entre o operador relacional existente na expressão e um outro operador relacional. Ou seja, uma vez escolhido um ponto de modificação (uma expressão binária com operador relacional), qualquer que seja o operador relacional existente na expressão, esse será trocado pelo operador relacional associado ao operador de mutação que será escolhido. Esse será nosso espaço de operadores de mutação.

Com essas informações, o Astor irá começar a navegar o espaço de busca. A estratégia é definida no operador EP_NS. No lugar de uma estratégia evolutiva, usaremos uma estratégia chamada de *seletiva*. Nessa estratégia, executamos um laço onde, em cada iteração, construímos uma lista com alguns dos pontos de modificação e, para cada um deles, aplicaremos um dado operador de mutação.

O ponto de extensão EP_MPS define como essa lista será montada. Vamos escolher um método aleatório uniforme, tal que todos os pontos de modificação têm igual probabilidade de serem escolhidos. O número de pontos selecionados é parâmetro do Astor: vamos supor que escolheremos apenas um. Haverá uma escolha aleatória entre as expressões `a != b` e `a > b`; supomos que a segunda foi escolhida.

Para cada ponto de mutação escolhido, será escolhido um operador de mutação. A forma como a escolha será feita é decidida pelo ponto de extensão EP_OS; supomos que

a escolha é feita de forma uniformemente aleatória, e que no nosso exemplo, o operador escolhido é o que troca o operador atual da expressão pelo operador >= do Java.

Na estratégia seletiva, o Astor cria programas variantes para cada ponto de mutação e seu respectivo operador de mutação, e em seguida verifica se eles são válidos ou não. A validação é feita pela função de aptidão, definida no ponto de extensão EP_FF. Usaremos uma função de aptidão simples que consiste apenas no número de testes que o programa falha, sendo 0 o melhor valor possível. O ponto de extensão EP_PV nos permite implementar medidas mais complexas a partir das quais calcular a função de aptidão, mas optamos por não fazer uso dessa funcionalidade, pois ela não será relevante para nós.

O programa resultante das modificações acima será:

Código 9 – Variante 1 gerada pelo Astor

```
1 relacao(a,b) {
2     if a != b:
3         return "Igual";
4     else if a >= b:
5         return "Maior";
6     else:
7         return "Menor";
8 }
```

O programa acima ainda não vai passar em nenhum dos testes, e o valor da função de aptidão será 3. Logo, a variante será descartada.

Tendo rodado a primeira iteração sem gerar nenhum programa correto, o Astor irá iterar novamente, escolhendo um novo ponto de mutação e um outro operador para aplicar nele, novamente de forma aleatória. Diferente da estratégia evolutiva, a estratégia seletiva não permite "reaproveitar"os programas gerados, partindo sempre do programa original, então a variante já gerada é ignorada.

O programa será consertado quando o ponto de modificação correspondente à expressão `a != b` for selecionado e, em seguida, o operador correspondente `a == b` for aplicado. O programa resultante passará em todos os testes e terá aptidão 0. Por padrão, entretanto, o Astor continuará rodando até esgotar os recursos especificados (por exemplo, até se passar uma certa quantidade de tempo ou um certo número de iterações), e no processo, pode gerar várias variantes corretas.

No final, o algoritmo retornará todas as variantes válidas e executará o passo de pós-processamento definido na extensão EP_SP. Esse passo poderia, por exemplo, classificar todas as variantes geradas de acordo com alguma medida opcional.

2.3.3 Métodos pré-definidos

Através dos pontos de extensão definidos, o Astor permite que usuários usem qualquer combinação de estratégias no reparo. Apesar disso, ele oferece também combinações pré-

definidas, algumas baseadas em métodos já existentes. No geral, o presente trabalho focará nessas combinações. Elas serão apresentadas a seguir.

a) **jGenProg**: implementa o GenProg, já detalhado anteriormente. Há algumas diferenças entre o algoritmo original e sua implementação no Astor. Primeiramente, o GenProg primeiro divide o programa a ser consertado em *statements*, e em seguida atribui aos mesmos pesos que identificam a probabilidade de estarem atrelados a uma falha. O Astor inicialmente avalia *linhas* do código a ser consertado, e só após atribuir os valores de suspeita são extraídos os *statements* que vão servir como pontos de modificação. Essa diferença se expressa em casos em que, por exemplo, uma única linha contem múltiplos *statements*; o Astor considerará a mutação igualmente provável para todos os *statements* da linha, enquanto o GenProg original pode acabar, teoricamente, atribuindo pesos diferentes aos mesmos.

   Além disso, o jGenProg usa uma técnica de localização de falha consideravelmente mais sofisticada do que a do GenProg original, chamada de GZoltar (CAMPOS et al., 2012). Isso permite que o jGenProg use valores de suspeita intermediários além dos 0, 1 e W_{Path} definidos no GenProg.

   Finalmente, na especificação original do GenProg, cada *statement* de um indivíduo é teoricamente candidato a mutação, embora com probabilidade baixa, de forma que poucos *statements* serão mutados na prática em cada geração. O Astor substitui esse processo por uma seleção aleatória de um número fixo de pontos de modificação para sofrerem mutação. Essa seleção pode ser uniformemente aleatória ou proporcional aos valores de suspeita. O Astor também substitui a função de aptidão original do GenProg por uma simplificada, onde a aptidão é apenas o número de testes que falham, com 0 sendo a melhor avaliação.

b) **jKali**: implementa o método Kali (QI et al., 2015). O Kali foi desenvolvido baseado na observação de que muitos dos patches gerados por métodos como o GenProg funcionam através da remoção de funcionalidade: ou removendo porções de código sem substituí-las, ou substituindo condições em blocos if por *true* ou *false* (fazendo com que ou o bloco em si ou o bloco *else* nunca sejam executados), ou introduzindo expressões de retorno ou saída que fazem com que partes do código não sejam executadas. Seguindo essa intuição, essas são as três mudanças possíveis que o Kali aplica em um programa.

   Os autores notam que, embora muitos dos programas que resultam dessas mudanças sejam capazes de passar em todos os testes, as mudanças em si podem introduzir problemas adicionais, como vulnerabilidades de segurança ou mesmo comportamentos incorretos que não são avaliados pela especificação. Portanto, o Kali também pode ser usado especificamente para identificar fraquezas nas suítes de testes, em alguns casos.

c) **jMutRepair**: baseado em (DEBROY; WONG, 2010). Realiza a troca de operadores relacionais ou lógicos por outros de mesma classe; por exemplo, trocando $>$ por $>=$ ou AND por OR. O espaço de busca pode ser navegado de forma exaustiva (todos os programas possíveis são gerados) ou seletiva (a cada passo, gera-se um único programa novo, selecionando um ponto de mutação usando EP_MPS e uma mutação usando EP_OS).

d) **DeepRepair**: adapta o método apresentado em (WHITE et al., 2017). O DeepRepair cria uma rede neural em cima do código do programa a ser consertado. O algoritmo em si funciona de forma semelhante ao GenProg; porém, ingredientes não são selecionados aleatoriamente, e sim ordenados de acordo com a semelhança entre os métodos ou classes de onde são retirados com os métodos ou classes onde vão ser inseridos. Essa semelhança é quantificada com base na rede neural. Os ingredientes escolhidos para uso são os de melhor classificação.

   O DeepRepair também faz uso de transformações para substituir variáveis (ou outros componentes) que passariam a estar fora de escopo após a inserção do ingrediente em seu contexto novo. A rede neural também é usada nas transformações, para avaliar os componentes candidatos (que substituirão as componentes fora de escopo).

e) **Cardumen**: detalhado em (MARTINEZ; MONPERRUS, 2018). O Cardumen extrai *templates* dos programas a serem consertados. Um *template* é uma porção de código contendo *placeholders* que podem ser substituídos por elementos apropriados disponíveis na localização de reparo. Por exemplo, uma chamada a uma função específica pode ser considerada um *template*, com seus parâmetros servindo de *placeholders*; patches possíveis são criados substituindo os *placeholders* por nomes de variáveis ou funções. Os patches candidatos são classificados com base em um modelo probabilístico construído em cima do programa: por exemplo, se um patch usar variáveis que frequentemente aparecem juntas dentro do código, esse será melhor classificado do que aqueles onde esse não é o caso.

f) **TIBRA**: uma variante do GenProg própria do Astor. Diferente do GenProg, o TIBRA aplica transformações nos ingredientes substituindo variáveis fora de escopo por variáveis aleatórias que estejam dentro do escopo.

2.4 EXEMPLO DE USO DO ASTOR: MATH-5

Em termos de verificação empírica, o desempenho de um método de reparo será extremamente dependente dos programas usados para teste. Em particular, programas criados especificamente para serem usados para avaliação de ferramentas de reparo podem acabar dessemelhantes a programas usados no mundo real (fora do contexto experimental), de

forma que os resultados obtidos testando em cima desses programas não venham a refletir os resultados que seriam obtidos em situações práticas.

(JUST; JALALI; ERNST, 2014) introduzem o Defects4J, um banco de dados que consiste em bugs coletados de programas reais (todos usando linguagem Java), disponibilizados para teste de métodos de reparo. O banco de dados é acompanhado de um framework que permite extrair uma versão de um programa contendo uma única falha a ser reparada e uma suíte de testes adequada para resolver essa falha. Além disso, cada falha incluída no banco possui um reparo já realizado manualmente no mundo real, ao qual os resultados do método automático podem ser comparados.

Para ter uma ideia concreta de como o Astor opera em uma falha real, vamos usá-lo para consertar um dos bugs mais simples do Defects4J, o Math-5 (retirado da biblioteca Commons-Math, já mencionada anteriormente), usando o método JGenProg.

O bug ocorre em uma classe (`Complex.java`) que representa números complexos. Uma das funções da classe, `reciprocal()`, retorna o recíproco do número, definido como:

$$\frac{a}{a^2 + b^2} - \frac{b}{a^2 + b^2}i \tag{2.3}$$

para um número complexo da forma a + bi.

A função leva em conta casos especiais, como quando o número representado pela classe é 0. A função inteira é apresentada a seguir. `real` e `imaginary` são propriedades da classe que representam as partes real e imaginária do número complexo. A classe também define os valores `NaN`, `ZERO` e `INF`.

Código 10 – Função reciprocal() do Complex.java

```
1 public Complex reciprocal() {
2     if (isNaN) {
3         return NaN;
4     }
5
6     if (real == 0.0 && imaginary == 0.0) {
7         return NaN;
8     }
9
10    if (isInfinite) {
11        return ZERO;
12    }
13
14    if (FastMath.abs(real) < FastMath.abs(imaginary)) {
15        double q = real / imaginary;
16        double scale = 1. / (real * q + imaginary);
17        return createComplex(scale * q, -scale);
18    } else {
19        double q = imaginary / real;
20        double scale = 1. / (imaginary * q + real);
21        return createComplex(scale, -scale * q);
22    }
23 }
```

O teste que falha no código acima é apresentado a seguir.

Código 11 – Teste testReciprocalZero()

```
1 @Test
2 public void testReciprocalZero() {
3     Assert.assertEquals(Complex.ZERO.reciprocal(), Complex.INF);
4 }
```

O teste acima chama a função `reciprocal()` para o valor zero, e irá passar se o valor retornado for `INF`. Claramente, a função está incorreta, e pode ser consertada com uma simples substituição da linha `return NaN` no segundo bloco condicional por `return INF`.

É necessário definir alguns parâmetros para a execução do Astor. O algoritmo irá parar ao atingir o máximo pré-definido de gerações ou de tempo; como o tempo de execução depende dos atributos do sistema no qual o algoritmo é executado, preferimos usar um limite arbitrariamente alto (de 1000 minutos) e tomar o número de gerações como o limite "verdadeiro". Um limite de 5000 gerações será mais do que o suficiente para a maioria dos casos, mas dependerá em parte do tamanho do programa a consertar e do método de reparo escolhido.

É necessário também definir um limiar (*threshold*) de suspeita, a partir do qual os pontos de modificação serão considerados. Já vimos que o uso do GZoltar permitirá

Tabela 3 – Parâmetros do experimento

Limite de tempo	1000 minutos
Limite de gerações	5000
Limiar de suspeita	0.01
Tamanho da população	1

Tabela 4 – Pontos de extensão do jGenProg

Ponto	Descrição	Opção escolhida
EP_FL	Localização de falhas	GZoltar
EP_MPG	Granularidade dos pontos de modificação	Statements
EP_OD	Espaço de operadores	Inserção, remoção e substituição de statements
EP_OS	Seleção dos operadores	Aleatória uniforme
EP_IPD	Definição do pool de ingredientes	Porções de código retiradas do mesmo pacote do código a ser modificado
EP_IS	Seleção de ingredientes	Aleatória uniforme
EP_IT	Transformação de ingredientes	Não utilizada
EP_PV	Validação de programas	Testes aprovados e reprovados
EP_FF	Função de aptidão	Número de testes falhados (0 é melhor possível)
EP_NS	Estratégia de navegação	Evolutiva
EP_MPS	Seleção de ponto de modificação	Aleatória uniforme
EP_SP	Priorização de solução	Não utilizada (paramos ao achar a primeira solução)

valores de suspeita além de 0, 1 e W_{Path}. Na maioria dos casos, esperamos que os pontos de modificação relevantes para o reparo tenham valores relativamente altos, mas há exceções; um limiar de por volta de 0.01 é por vezes necessário para que o programa leve em consideração as seções importantes do código.

Veremos em seguida que, no caso do Math-5, o ponto de modificação mais importante terá um valor de suspeita de 1, mas em situações naturais (onde não sabemos a modificação que resolve de fato a falha), essa suposição poderia impedir que achássemos a solução. Por isso, vamos usar um limiar de 0.01.

Finalmente, precisamos escolher o tamanho da população. Como só precisamos de uma única modificação para o conserto do código, não será útil fazer uso do *crossover*. Optamos por usar uma população de apenas um indivíduo, para poder acompanhar claramente o comportamento do Astor.

O primeiro passo será a localização de falhas. Como explicado anteriormente, o jGen-

Prog faz uso do método GZoltar para atribuir valores de suspeita a cada linha do código, proporcionais à probabilidade estimada da linha ser responsável pela falha no programa. Para cada linha acima do valor de *threshold*, são criados pontos de modificação baseado na granularidade escolhida (no caso, o jGenProg trabalha no nível de *statements*, então cada *statement* em cada linha será um ponto de modificação). A saída abaixo relata a criação de todos os pontos de modificação.

Código 12 – Pontos de modificação criados para o Math-5

```
1 Creating variant 1
2 --ModifPoint:CtReturnImpl, suspValue 1.0, line 305, file Complex.java
3 --ModifPoint:CtReturnImpl, suspValue 1.0, line 1228, file Complex.java
4 --ModifPoint:CtIfImpl, suspValue 0.4472135954999579, line 304, file
     Complex.java
5 --ModifPoint:CtIfImpl, suspValue 0.4082482904638631, line 300, file
     Complex.java
6 --ModifPoint:CtReturnImpl, suspValue 0.1543033499620919, line 348, file
     Complex.java
7 --ModifPoint:CtLocalVariableImpl, suspValue 0.14907119849998599, line
     344, file Complex.java
8 --ModifPoint:CtIfImpl, suspValue 0.14907119849998599, line 345, file
     Complex.java
9 --ModifPoint:CtIfImpl, suspValue 0.14586499149789456, line 343, file
     Complex.java
10 --ModifPoint:CtIfImpl, suspValue 0.1259881576697424, line 340, file
     Complex.java
11 --ModifPoint:CtAssignmentImpl, suspValue 0.07142857142857142, line 99,
     file Complex.java
12 --ModifPoint:CtAssignmentImpl, suspValue 0.07142857142857142, line 100,
     file Complex.java
13 --ModifPoint:CtAssignmentImpl, suspValue 0.07142857142857142, line 102,
     file Complex.java
14 --ModifPoint:CtAssignmentImpl, suspValue 0.07142857142857142, line 103,
     file Complex.java
```

Os valores como `CtReturnImpl` ou `CtIfImpl` são as classificações dos *statements* dentro do compilador Spoon[3], que é usado pelo Astor para interagir com o código. Além disso são relatados os valores de suspeita e a linha e arquivo do código em questão. O primeiro ponto de modificação criado acima, na linha 305 do arquivo `Complex.java`, é justamente o `return NaN` que queremos modificar, e tem valor de suspeita 1.

Com os pontos de modificação criados, o laço principal do jGenProg começa. Em cada geração, um ponto de modificação será escolhido para sofrer uma mutação aleatória. A primeira mutação que precisar de algum ingrediente criará o *pool* de ingredientes que será usada no restante da execução. Na nossa execução, na geração 20, a mutação escolhida

[3] https://spoon.gforge.inria.fr/

resolveu o problema, como relatada na seção a seguir da saída.

Código 13 – Geração 20 da execução

```
***** Generation 20 : 0
[...]
location: Complex.java305
[...]
---OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return org.apache.
    commons.math3.complex.Complex.NaN ‘ -topatch--> ‘return org.apache.
    commons.math3.complex.Complex.INF ‘ (spoon.support.reflect.code.
    CtReturnImpl)
[...]
TR: Success: true, failTest= 0, was successful: true, cases executed:
    123] ,[]
```

O Astor escolheu o ponto da linha 305 (`return NaN`) para modificação, o operador de substituição, e o ingrediente `return INF`, retirado de outra porção do código. O código resultante passará em todos os testes fornecidos e será considerado uma solução. A execução pode continuar em seguida, mas optamos por terminar o algoritmo uma vez que a primeira solução é encontrada.

Devido aos fatores aleatórios, outras execuções podem prosseguir de forma um pouco diferente da relatada acima.

Tendo visto com detalhe como o Astor opera em um bug específico, discutiremos em seguida seu desempenho geral. O Defects4J serve como um conjunto de bugs em comum que pode ser usado não só para avaliar métodos individuais, mas também para compará-los entre si. (MARTINEZ; MONPERRUS, 2019) o usam para verificar o desempenho de todos os métodos disponibilizados pelo Astor.

No final, os autores relatam que 98 dos 357 bugs disponibilizados pelo Defects4J na época são reparados por pelo menos um dos métodos implementados, representando uma taxa de sucesso de 27.4%. O método individual do Astor com melhor sucesso foi o Cardumen (77 bugs consertados, 21.5%), seguido do DeepRepair (51 bugs, 14.2%), jGenProg (49 bugs, 13.7%), TIBRA (35 bugs, 9.8%), jKali (29 bugs, 8.1%) e jMutRepair (23 bugs, 6.4%). Comparando esse desempenho com oito outros sistemas de reparo contemporâneos, concluem que o Astor é superior a sete deles, e inferior a apenas um.

O Astor é, portanto, um dos exemplos preeminentes de reparo de programas atuais. Entretanto, ainda apresenta espaço para melhorias possíveis. No capítulo a seguir, vamos apresentar os fundamentos para uma abordagem nova, que será construída em cima dos conceitos básicos do Astor, visando aprimorar os resultados já obtidos até aqui.

3 INTRODUZINDO CORRETUDE RELATIVA AO MODELO DE GERAÇÃO E VALIDAÇÃO

Nesse capítulo, vamos introduzir um conceito novo ao nosso modelo de reparo, e depois discutir as melhoras possíveis que esperamos ver como consequência.

3.1 TEORIA DE FALHAS

Até o momento, analisamos alguns modelos diferentes de reparo de programas sem definir rigorosamente o conceito de *falha*. Embora seja possível alcançar resultados úteis sem esse rigor, uma teoria sistemática se mostrará útil para conceitualizar métodos novos.

(DIALLO et al., 2017) desenvolvem uma teoria formal de falhas e reparos que será usada no restante desse trabalho. Seus fundamentos serão apresentados a seguir.

3.1.1 Corretude

Consideramos primeiro um *espaço de estados* S: um conjunto que contém os valores possíveis que as variáveis de um programa podem assumir. Um programa P pode ser definido como um conjunto de relações *(x,y)*, tal que x e y pertencem a S, onde o programa retorna o valor y como saída se receber x como entrada. É importante destacar que estaremos lidando apenas com programas determinísticos; programas não determinísticos vão além do escopo desse trabalho. Assim sendo, para cada entrada possível, um programa terá apenas uma saída.

Já foi mencionado que algoritmos de reparo automático costumam usar suítes de teste como especificação. Uma maneira alternativa de definir uma especificação é como um conjunto R de relações *(x,y)* dentro do espaço S. Diferente de um programa, uma especificação pode incluir múltiplas saídas possíveis para uma mesma entrada. Ou seja, (1,2) e (1,3) podem fazer parte da mesma especificação.

O domínio de competência (*competence domain*) de um programa P em relação a uma especificação R é o conjunto de valores x tal que *(x,y)* pertence tanto a P quanto a R. Ou seja, é o conjunto de entradas x a partir das quais P se comporta da maneira especificada por R, retornando a saída correta y.

Considerando dois programas P e P' com domínios de competência CD e CD', respectivamente, P' é considerado *mais correto* que P se e somente se $CD \subseteq CD'$. P' é *estritamente mais correto* que P se e somente se $CD \subset CD'$. A noção de que um programa pode estar mais correto que outro, que chamaremos de corretude relativa (*relative correctness*), define uma ordenação parcial entre programas diferentes. Destacamos que um programa pode ser mais correto que outro sem imitar exatamente seu comportamento, como ilustramos no exemplo a seguir.

Exemplo 3.1

Seja S um espaço de estados, R uma especificação, e P, P' e P'' programas cujos domínios de competência são CD, CD' e CD'', respectivamente.

S = {0,1,2}.
R = {(0,1),(1,1),(1,2),(2,0)}.
P = {(1,1)}.
P' = {(1,2)}.
P" = {(0,1),(1,2)}.
CD = {1}.
CD' = {1}.
CD" = {0,1}.

P não é estritamente mais correto que P', e vice versa. P'' é estritamente mais correto que ambos, apesar de demonstrar um comportamento diferente com a entrada 1 do que aquele demonstrado pelo programa P. Isso porque, apesar de $P \not\subset P''$, $CD \subset CD''$.

Também destacamos que o número de elementos dos domínios de competência não necessariamente nos diz algo sobre a corretude relativa, como visto no exemplo a seguir.

S = {0,1,2}.
R = {(0,1),(1,1),(1,2),(2,0)}.
P = {(0,1)}.
P' = {(1,1),(2,0)}.
CD = {0}
CD' = {1,2}

Apesar do programa P' estar correto (em relação a R) para um número maior de entradas, $CD \not\subseteq CD'$. Portanto, não diremos que P' é mais correto que P.

Um programa P no espaço de estados S será considerado *absolutamente correto* ou apenas *correto* de acordo com uma especificação R se e somente se P for mais correto que qualquer outro programa possível em S. Essa definição é importante pois significa que podemos alcançar a corretude absoluta gerando programas cada vez mais corretos que os anteriores.

3.1.2 Falhas

Intuitivamente, consertar um programa incorreto envolve a criação de um programa novo que é estritamente mais correto que o anterior. Os dois programas obviamente serão diferentes em termos de seu código fonte. Chamaremos de *falha* toda e qualquer parte de um programa incorreto que pode ser substituída de forma a gerar um programa estritamente mais correto. Essa substituição é chamada de remoção de falha (*fault removal*). O conceito de substituição usado aqui é abrangente: remover uma porção de código

sem inserir código novo pode ser considerado uma substituição, assim como inserir sem remover.

Rigorosamente, falhas não precisam ter a mesma granularidade. Um único operador de uma expressão pode ser considerado uma falha e, ao mesmo tempo, um bloco inteiro pode ser considerado uma falha dentro do mesmo programa. Entretanto, ao tratar de reparo automático de programas, costuma-se lidar com partes do código de mesma granularidade. (KHAIREDDINE; MARTINEZ; MILI, 2019) apontam que, por exemplo, podemos considerar o programa todo como uma falha (se estiver incorreto), mas isso não nos será útil se quisermos identificar as partes incorretas do código ou as substituições que devem ser feitas. É razoável, então, supor que em casos reais, "falhas" se referem a partes do código que possuem a mesma granularidade. Entretanto, isso é uma consideração prática que pode variar de caso a caso.

Também não é estritamente necessário que uma falha seja uma porção contígua de código. Ou seja, duas partes distintas podem ser consideradas como uma única falha contanto que a substituição simultânea de ambas possa gerar um programa estritamente mais correto. Nesse caso, entretanto, também demandamos que nenhuma das partes distintas seja, por si só, uma falha. Para esclarecer esse conceito, chamaremos de *falha elementar* toda falha que ou a) abrange apenas um elemento do código, ou b) abrange vários elementos (contíguos ou não), nenhum dos quais é uma falha por si só. Como no parágrafo acima, porém, essa definição também está sujeita a considerações práticas. Em alguns casos, pode ser razoável limitar falhas para apenas porções contíguas.

A densidade de falhas (*fault density*) de um programa é a quantidade de falhas elementares presentes no mesmo. Essa medida pode ser enganadora: se um programa tem cinco falhas, por exemplo, remover uma delas não necessariamente resultará em um programa com quatro falhas: é possível que a remoção de uma porção do código faça com que outras parem de funcionar, ou passem a funcionar corretamente.

Para amenizar esse problema, (DIALLO et al., 2017) introduzem o conceito de profundidade de falhas (*fault depth*). A profundidade de falhas de um programa P é o menor número de remoções de falha necessárias para obter um programa absolutamente correto P' a partir de P. Para ilustrar essa diferença, junto com o conceito de falhas em geral, consideramos o programa de exemplo abaixo, escrito em pseudocódigo.

Exemplo 3.1

Código 14 – Exemplo para ilustrar profundidade e densidade de falhas

```
1 boolean exemplo (int b) {
2     int a;
3     a = 2;
4
5     if (a == 5) {
6         return false;
7     }
8
9     if (a == 2) {
10        b = 3;
11        return false;
12    }
13
14    if (b != 3) {
15        return false;
16    }
17
18    return true;
19 }
```

Consideramos que o programa acima será correto somente se ele retornar **true** para todas as entradas (valores de `b`) possíveis. Assim, ele está claramente incorreto em todos os casos. Como a variável *a* é sempre igual a 2, o segundo bloco condicional sempre irá executar e retornar **false**.

Como já dissemos anteriormente, é razoável considerar que as falhas do programa terão uma mesma granularidade. Nesse caso, consideramos que a granularidade do programa consiste em apenas instruções simples (não incluindo blocos) e expressões. Tendo isso em mente, veremos quais substituições podem ser feitas para consertar o programa acima.

Considerando a declaração inicial "int a", na linha 2, podemos substituí-la por uma declaração combinada com uma inicialização, como "int a = 3". Porém, isso não afetará a lógica do programa, pois *a* receberá uma atribuição diferente logo em seguida. Consideraremos também que não podemos substituir a declaração por qualquer código que não envolva uma declaração de *a*, pois em muitas linguagens de programação, a primeira condicional ("if (a == 5)") resultaria em um erro de compilação, usando uma variável não declarada como l-value. Assim, não consideramos "int a" como uma falha.

O primeiro bloco condicional (na linha 5) não afeta o comportamento do programa, pois sua condição nunca é satisfeita. Substituir sua expressão condicional ou a instrução em seu interior não podem tornar o programa correto.

A expressão "a = 2" é claramente uma falha. Podemos substituí-la por qualquer outra atribuição que não tenha valor 5 (como "a = 3") e o programa resultante retornará **true** para a entrada b = 3, pois as três condicionais não serão válidas. Além disso, podemos

substituí-la por "return **true**", o que garantirá a corretude do programa para todas as entradas possíveis. Em ambos os casos, o programa novo será estritamente mais correto que o programa original.

De forma semelhante, a expressão "a == 2" dentro da segunda condicional (linha 9) também é uma falha. Se substituirmos por uma expressão que sempre será incorreta (como "false"), o programa retornará **true** para b = 3. Esse novo programa também é estritamente mais correto que o original.

A expressão "b = 3" dentro do segundo bloco condicional é uma falha, pois pode ser substituída por "return **true**" e tornar o programa resultante estritamente mais correto que o original para todas as entradas possíveis (e, na verdade, absolutamente correto). O "return false" do mesmo bloco também pode ser substituído por um "return true" e obter o mesmo resultado. As porções subsequentes do programa não são falhas, pois substituí-las não impede que o "return false" do segundo condicional seja executado.

Assim, contamos uma densidade de falhas igual a 4 no programa apresentado. Entretanto, sua profundidade de falhas é apenas 1: substituir qualquer uma das declarações dentro do segundo bloco condicional por "return true" faz com que o programa se comporte corretamente em todos os casos.

Também a partir do programa acima, podemos ilustrar alguns princípios adicionais destacáveis. Primeiramente, uma mesma falha permite múltiplas substituições que podem resultar em níveis diferentes de corretude: substituir "a = 2" por "b = 2" fará com que o programa continue incorreto para todos os casos; substituir por "a = 3" nos dará um programa mais correto; substituir por "return true" nos dará um programa *absolutamente* correto.

Além disso, uma remoção de falhas pode gerar falhas novas. Substituir "a = 2" por "a = 5" faz com que o primeiro bloco condicional seja executado, falhando o programa em todos os casos. Ao mesmo tempo, outras falhas podem deixar de existir: todo o código após o primeiro bloco condicional deixa de contar como falha, pois sua substituição não afetará em nada o comportamento do programa, já que nunca será executado.

3.2 CORRETUDE RELATIVA E O RCFIX

Como discutimos anteriormente, a teoria de falhas de (DIALLO et al., 2017) permite alcançar a corretude absoluta gerando programas cada vez mais corretos. (KHAIREDDINE; MARTINEZ; MILI, 2019) constroem um algoritmo de reparo de programas, chamado de RCFix, baseado nesse princípio. O RCFix é um algoritmo geral, que não especifica os próprios operadores de mutação, mas sim faz uso dos operadores de métodos já existentes (como o GenProg). Ele será visto em seguida, de forma simplificada.

Teremos como entrada um programa incorreto P, uma especificação R a partir da qual julgar a corretude, e um limite MaxM (chamado de limite de multiplicidade) que define o

tamanho máximo permitido para falhas elementares compostas, isso é, falhas elementares que abrangem múltiplas porções de código. Além disso, o programa supõe um gerador de patches não especificado, que define os operadores de mutação que poderão ser usados e gera programas novos a partir dos mesmos. O método particular usado para gerar patches é flexível e não é crucial para os princípios do algoritmo.

Código 15 – Pseudocódigo do RCFix

```
func rcfix(P,R,MaxM,gerador):
  corretudeAbsoluta = falso
  multiplicidade = 1

  enquanto corretudeAbsoluta = falso e multiplicidade < maxM:
    multiplicidade = 1
    corretudeRelativa = falso

    enquanto corretudeRelativa = falso e multiplicidade < maxM:
      programasModificados = gerarPatches(gerador,P,multiplicidade)
      para cada programaNovo em programasModificados:
        corretudeRelativa = testarCorretudeRelativa(programaNovo,R)
        se corretudeRelativa = verdadeiro:
          corretudeAbsoluta = testarCorretudeAbsoluta(programaNovo,R)
          P = programaNovo
      multiplicidade = multiplicidade + 1

  retornar P
```

O algoritmo começa tentando alcançar um aumento unitário na corretude relativa de P, isso é, tentando achar um programa P' estritamente mais correto que P. Para isso, começando com uma multiplicidade de 1, o programa testa todos os patches possíveis retornados pelo gerador até que não existam mais patches para testar com essa multiplicidade, ou até que um programa estritamente mais correto P' seja gerado. No primeiro caso, a multiplicidade é incrementada e a fase de testes é recomeçada com essa nova multiplicidade. No segundo caso, o processo inteiro é repetido mas tomando P' como programa de entrada.

Um programa novo P' é considerado estritamente mais correto se ele passar nos mesmos testes que o programa original P e pelo menos um teste novo em que o programa original não passava. Ele é considerado absolutamente correto se passar em todos os testes.

Há três condições sob as quais o algoritmo inteiro irá parar. Em uma delas, um programa absolutamente correto é encontrado. Nesse caso, o algoritmo executou com sucesso. Por outro lado, se a multiplicidade exceder MaxM sem que nenhum dos patches gerados até então incremente a corretude relativa do programa, consideramos que o algoritmo falhou. Nesse caso, porém, ainda teremos como saída o programa com maior corretude

relativa encontrado até o momento. Finalmente, o algoritmo irá parar em qualquer momento se o tempo máximo for atingido. Nesse caso, também consideramos que o algoritmo falhou, e ainda retornaremos como saída o programa com maior corretude relativa.

3.3 ADAPTANDO MÉTODOS EXISTENTES USANDO CORRETUDE RELATIVA

(KHAIREDDINE; MARTINEZ; MILI, 2019) adaptam o método jGenProg para usar a corretude relativa seguindo o algoritmo do RCFix. Os mesmos princípios adotados por eles podem ser usados para os demais métodos. O uso de um método já existente afeta principalmente o passo de geração de patches do RCFix.

Para um dado método do Astor, vamos fazer uso de sua granularidade (EP_MPG), técnica de localização de falha (EP_FL), seus operadores de mutação (EP_OD) e seus métodos de criação de ingredientes (EP_IPD e EP_IT); os demais pontos de extensão são ignorados.

Patches possíveis são gerados primeiro identificado pontos de modificação (usando a localização de falhas e granularidade) e em seguida criando, para cada ponto, uma lista com todos os operadores possíveis que poderiam ser aplicados. Haverá um programa novo criado para cada patch, e cada programa será testado um de cada vez, como visto na seção anterior. Uma consequência dessa abordagem é que a localização de falhas é rodada sempre que se geram os patches, ao invés de uma única vez no início do algoritmo (como é feito no Astor).

Todos os programas variantes gerados são guardados em uma lista, independente do seu resultado. Uma vez incrementada a multiplicidade e chegando novamente na geração de patches, o processo descrito no parágrafo anterior é repetido, porém usando cada programa dessa lista como base, no lugar do programa original P. Como os programas da lista já possuem um certo número de modificações (dependendo da multiplicidade anterior), uma mutação adicional irá criar um programa com a multiplicidade incrementada.

3.4 DISCUSSÃO

Podemos considerar que tanto o Astor quanto o RCFix focam em navegar um espaço de soluções possíveis visando chegar em um programa mais correto.

Nas estratégias padrões do Astor, esse espaço é navegado usando uma mudança unitária (isso é, que afeta apenas um elemento do código) por vez, e geralmente usando fatores aleatórios (seja na seleção de operadores de mutação, na seleção de pontos de modificação, etc.). Essa navegação será bem sucedida apenas se terminar em um programa absolutamente correto.

Por outro lado, o RCFix permite mudanças simultâneas em múltiplas partes do código, explora necessariamente todas as mutações possíveis (obedecendo um limite de multiplicidade), e em um dado instante visa apenas obter um programa estritamente mais correto,

com a corretude absoluta sendo apenas um produto da execução sucessiva do algoritmo em programas cada vez mais corretos. Ele é, teoricamente, capaz de realizar qualquer reparo, contanto que o valor de MaxM seja alto o suficiente e que o gerador de patches usado seja capaz de realizar as transformações necessárias.

Essas diferenças acarretam em diversas consequências práticas. Entre elas, podemos destacar:

a) O RCFix permite consertar falhas elementares que abrangem vários elementos de código. As estratégias exaustiva e seletiva do Astor só possibilitam mudanças únicas que tornam o programa absolutamente correto, enquanto a estratégia evolutiva permite mudanças cumulativas, mas apenas se cada uma delas resultar em uma melhora no valor da função de aptidão. Assim, no geral, a estratégia evolutiva só será capaz de reparar falhas elementares unitárias.

   Por outro lado, o RCFix é capaz de navegar mudanças múltiplas de forma controlada, tentando primeiro todas as mudanças com uma certa multiplicidade, depois todas as mudanças com a multiplicidade incrementada, e assim por diante.

b) O RCFix é capaz de reparar programas com profundidade de falhas arbitrária, tanto pelo uso de multiplicidade quanto pela consideração da corretude relativa.

   Pelo item anterior, já fica claro que as estratégias exaustiva e seletiva não são capazes de fazer o mesmo. A estratégia evolutiva pode, em alguns casos, reparar um programa com profundidade de falhas maior que 1, mas também falhará em outros casos específicos. Podemos destacar casos onde uma mudança única é necessária para resolver algumas falhas do programa, mas acaba também criando outras. Na estratégia evolutiva, isso pode resultar em uma função de aptidão pior, tal que a mudança não será considerada em gerações seguintes. O uso de multiplicidade permite que essa mudança seja incorporada junto com mudanças adicionais que resolvem as falhas criadas, resultando em um programa estritamente mais correto.

c) O fato do RCFix tratar cada programa estritamente mais correto como uma entrada nova significa que a localização de falhas será executada novamente sempre que alcançarmos um programa mais correto (o que não é feito na estratégia evolutiva), permitindo fazer uso de informações novas para encontrar falhas que antes não teriam sido detectadas.

d) O RCFix pode entregar um programa estritamente mais correto sem necessariamente chegar na corretude absoluta, possibilitando até misturar métodos diferentes em um processo de reparo, através de execuções consecutivas.

e) Em certos casos, uma busca exaustiva pode ser mais veloz que uma busca seletiva ou evolutiva, já que esses métodos envolvem uso de estratégias aleatórias. O RCFix pode tomar vantagem disso sem sofrer das limitações que a busca exaustiva

apresenta no contexto do Astor (que só conserta programas com profundidade de falhas igual a 1).

Além disso, o RCFix é, na teoria, capaz de resolver qualquer falha que já possa ser solucionada por um método do Astor, contanto que as regras usadas para a geração de patches sejam as mesmas (isso é, contanto que usem os mesmos operadores, localização de falha, e assim por diante). Isso se dá pois o RCFix irá explorar todos os patches possíveis gerados pelo respectivo método do Astor.

Por esses motivos, queremos adaptar os métodos principais do Astor para usarem corretude relativa e multiplicidade através do RCFix. Considerando os fatores destacados acima, esperamos que no final, os métodos adaptados sejam capazes de consertar programas que antes não conseguiam.

4 IMPLEMENTAÇÃO, EXPERIMENTOS E RESULTADOS

No capítulo a seguir, apresentamos uma análise dos resultados do Astor que aponta para alguma das inadequações presentes no framework. Em seguida, descrevemos nossa implementação do RCFix e realizamos alguns estudos de caso em luz das inadequações apontadas, visando entender as diferenças entre o RCFix e o Astor.

4.1 ANÁLISE DOS RESULTADOS DO ASTOR

Apesar da taxa de sucesso geral do Astor, a quantidade de bugs do Defects4J que não podem ser reparados por nenhum dos métodos disponíveis ainda aponta para alguns pontos cegos que todos eles têm em comum. É de interesse analisar os bugs não consertados de forma a descobrir características que possam compartilhar entre si.

(SOBREIRA et al., 2018) realizaram um levantamento (chamado de Defects4J Dissection) das características das falhas disponíveis no Defects4J, assim como dos patches manuais que as consertam (isso é, aqueles feitos pelos desenvolvedores dos programas originais, e não gerados por métodos automáticos). Entre os dados coletados estão as ações tomadas pelo patch (por exemplo, adicionar uma chamada a um método ou modificar uma declaração), exceções geradas pelo programa original, ou o número de linhas adicionadas ou removidas pelo patch. Além disso, os autores registram quais métodos conhecidos podem reparar as falhas, dentre uma seleção que inclui os métodos do Astor, entre outros. No total, entre os 395 bugs incluídos no Dissection (o artigo faz uso de uma versão do Defects4J posterior a da usada na avaliação do Astor, incluindo mais bugs), 189 possuem algum reparo automático (63.6%).

Baseado no levantamento desses autores, nós fizemos uma análise da frequência com que as diferentes características recorrem entre as diferentes falhas, e com que frequência falhas com tais características são reparadas pelos métodos levados em conta pelo Dissection. Por exemplo, uma das ações possíveis tomadas pelos patches é a adição de um bloco condicional que contém um **return**. O Dissection relata que há 77 bugs cujo reparo manual envolve essa adição. De acordo com a nossa análise, 35 desses bugs são reparados pelos métodos em questão. Usando tais medidas, podemos descobrir pontos fracos nos métodos existentes.

Ao realizar tal análise, precisamos levar em conta a frequência da ocorrência da característica no banco de dados. Uma característica que só ocorre em um número pequeno de bugs do Defects4J não nos é interessante, pois pode não ser uma amostra fiel da população geral de bugs semelhantes.

Outro fator que deve ser considerado é que a presença de uma certa característica em uma falha não resolvida não significa, necessariamente, que a característica em si é o que

apresenta um obstáculo para os métodos. Como exemplo, 149 dos bugs envolvem a adição de um bloco if em seu reparo manual; desses, apenas 57 são capazes de ser consertados por métodos automáticos. Não podemos necessariamente deduzir, entretanto, que isso indica uma deficiência na capacidade dos métodos de introduzir blocos if adequados. A falha no conserto automático pode resultar da incapacidade de realizar alguma outra ação crucial para o reparo manual. Portanto, devemos exercer cautela ao analisar os resultados.

Realizando nossa análise, verificamos que, no geral, não parece haver nenhuma exceção presente nos bugs ou ação tomada pelos patches manuais que está totalmente além do escopo dos métodos. Dentre as características presentes apenas em bugs que não são reparados por nenhum método, nenhuma ocorre em mais do que dez dos bugs do Defects4J, então consideramos que não há uma amostragem adequada para chegarmos a conclusões sobre o desempenho no geral. Não podemos concluir, portanto, que as falhas por parte dos métodos de reparo se dão devido a um tipo de erro ou ação em particular; o contexto em que ocorrem, ou combinações específicas de características, parecem ser mais importantes.

Tendo isso em mente, uma característica interessante de analisar é a capacidade dos métodos existentes de executar reparos que requerem modificação em múltiplas linhas, comparado a reparos que requerem modificação em apenas uma. O Dissection relata 98 bugs que podem ser consertados modificando apenas uma linha; nossa análise aponta que, dentre esses, 71 (72.4%) são reparados por métodos automáticos (notamos que estamos incluindo métodos fora da implementação padrão do Astor). Dos 297 restantes, apenas 118 são reparados (39.7%). Embora esse não seja um desempenho exatamente ruim, a diferença nos dois casos é grande o suficiente para indicar que a complexidade dos reparos necessários é um fator importante na possibilidade dos métodos automáticos consertarem uma dada falha.

Intuitivamente, como todo método de reparo trabalha com um espaço de busca limitado por fatores como as mutações possíveis ou a granularidade do ponto de modificação, a necessidade de um maior número de mudanças para reparar um programa acarretará em uma maior probabilidade do programa correto (resultado de todas as mudanças) estar fora do espaço de busca. Mesmo assim, veremos mais a frente que a necessidade de realizar múltiplas modificações pode gerar dificuldades mesmo em casos aparentemente simples.

É razoável supor que o conceito de corretude relativa, e sua aplicação no RCFix, pode ajudar a remediar esse problema. Se um bug só é consertado modificando mais de uma seção do código, ele provavelmente terá ou uma profundidade de falhas maior do que 1, ou falhas elementares que abrangem vários elementos de código. Já vimos que o RCFix é adequado para ambas as situações. Acreditamos, então, que o RCFix será útil para cobrir os pontos cegos apontados por nossa análise do Defects4J Dissection.

4.2 O RCFIX ESTENDIDO

(KHAIREDDINE; MARTINEZ; MILI, 2019) realizaram experimentos iniciais para o RCFix usando um protótipo do algoritmo construído em cima do programa do Astor. Entretanto, o GenProg foi o único método adaptado, e o algoritmo foi usado apenas para demonstrar a possibilidade de reparo em casos onde múltiplos bugs do Defects4J estavam presentes em um único programa. Assim, o impacto da implementação dos autores foi limitado, comparado ao potencial que discutimos no capítulo anterior.

Em nosso trabalho, criamos uma versão do RCFix que implementa todos os métodos nativos do Astor, e buscamos testar ele em um conjunto mais abrangente de situações. O nosso programa faz uso das funções e classes já implementadas pelo Astor, mas no contexto de um algoritmo novo. Ele é baseado na implementação usada por (KHAIREDDINE; MARTINEZ; MILI, 2019), que foi adaptada por nós para ser compatível com a versão mais nova do Astor e capaz de usar todos os métodos apresentados por (MARTINEZ; MONPERRUS, 2019).

Assim, esperamos não só ampliar a utilidade do RCFix, como explorar seu potencial de forma mais profunda. Nos experimentos que seguem, buscamos confirmar as melhorias que o RCFix traz para o Astor, comparando o funcionamento e desempenho dos dois algoritmos em alguns casos escolhidos.

Continuaremos usando os parâmetros definidos na seção 2.4, isso é, um tempo máximo arbitrariamente alto de 1000 minutos, um número máximo de gerações igual a 5000 e um limiar de 0.01 para valores de suspeita.

4.3 PROFUNDIDADE DE FALHAS ARBITRÁRIA - MÉTODO EVOLUTIVO

Vamos analisar um bug da biblioteca Commons-Math e o comportamento padrão do jGenProg no mesmo. Especificamente, esse é o bug indexado como Math-22 do Defects4J.

O bug envolve duas classes que simulam distribuições probabilísticas: `FDistribution` e `UniformRealDistribution`. Ambas as classes possuem métodos que retornam informações sobre o *suporte* das suas respectivas distribuições. O suporte de uma função matemática é o subconjunto do seu domínio que exclui elementos onde a função é igual a 0 (SAUVIGNY, 2006). Os métodos `isSupportLowerBoundInclusive` e `isSupportUpperBound_Inclusive`, presente em ambas as classes, informam se os extremos inferiores e superiores dos seus suportes estão inclusos no conjunto ou não (retornando `true` ou `false`).

Pela especificação da biblioteca, o extremo vai estar incluso no próprio suporte se e somente se ele for finito e a função de densidade da distribuição for definida para aquele ponto. Essas condições são verificadas pelos casos de teste abaixo:

Código 16 – Casos de teste para o Math 22

```
1 @Test
2 public void testIsSupportLowerBoundInclusive() {
3     final double lowerBound = distribution.getSupportLowerBound();
4     double result = Double.NaN;
5     result = distribution.density(lowerBound);
6     Assert.assertEquals(!Double.isInfinite(lowerBound) &&
7         !Double.isNaN(result) &&
8         !Double.isInfinite(result),
9         distribution.isSupportLowerBoundInclusive());
10 }
11
12 @Test
13 public void testIsSupportUpperBoundInclusive() {
14     final double upperBound = distribution.getSupportUpperBound();
15     double result = Double.NaN;
16     result = distribution.density(upperBound);
17     Assert.assertEquals(!Double.isInfinite(upperBound) &&
18         !Double.isNaN(result) &&
19         !Double.isInfinite(result),
20         BoundInclusive());
21 }
```

Os casos de teste acima serão executados tanto para `FDistribution` quanto para `UniformRealDistribution`. Consideramos que sabemos de antemão que ambos os métodos devem retornar `false` para `FDistribution` e `true` para `UniformRealDistribution` (pois refletem as propriedades verdadeiras dessas distribuições na matemática).

Nas classes em questão, o código para as funções `isSupportLowerBoundInclusive` e `isSupportUpperBoundInclusive` não envolve nenhum cálculo, apenas retornando a resposta como um booleano. O código para essas funções no Math-22 é igual em ambas as classes, incorretamente identificando que `isSupportLowerBoundInclusive` é verdadeiro e `isSupportUpperBoundInclusive` falso para ambas as distribuições, como no trecho a seguir:

Código 17 – Trecho do código para FDistribution.java e UniformRealDistribution.java

```
1 public boolean isSupportLowerBoundInclusive() {
2     return true;
3 }
4
5 public boolean isSupportUpperBoundInclusive() {
6     return false;
7 }
```

A solução para esse bug é extremamente simples, envolvendo apenas a troca dos valores booleanos atuais pelos corretos. Como o código já possui os *statements* `return true` e

`return false`, o jGenProg deveria ser capaz de consertar o programa com uma simples substituição em cada um dos arquivos. Entretanto, (MARTINEZ; MONPERRUS, 2019) relatam que o método não consegue resolver o problema. A seguir, veremos com detalhe a execução do jGenProg em cima desse bug. Para comparação, usaremos os mesmos parâmetros que usamos para o reparo do Math-5 (seção 2.4).

O primeiro passo será a localização de falhas, atribuindo valores de suspeita a cada linha do código, seguida da criação de pontos de modificação, em ordem decrescente de valor de suspeita.

Código 18 – Criação dos pontos de modificação para o Math-22

```
1 Creating variant 1
2 --ModifPoint:CtReturnImpl, suspValue 0.7071067811865475, line 184, file
     UniformRealDistribution.java
3 --ModifPoint:CtReturnImpl, suspValue 0.7071067811865475, line 275, file
     FDistribution.java
4 --ModifPoint:CtIfImpl, suspValue 0.35355339059327373, line 109, file
     UniformRealDistribution.java
5 --ModifPoint:CtReturnImpl, suspValue 0.35355339059327373, line 112, file
      UniformRealDistribution.java
6 --ModifPoint:CtLocalVariableImpl, suspValue 0.35355339059327373, line
     129, file FDistribution.java
7 [...]
```

A lista acima foi abreviada; o programa cria 226 pontos de modificação no total. A linha 184 do `UniformRealDistribution.java` e a linha 275 do `FDistribution.java` de fato correspondem aos trechos que devem ser modificados nos respectivos códigos, então o trecho acima nos diz que foram criados pontos de modificação para as regiões cruciais do código, e que eles possuem os maiores valores de suspeita.

Após a criação dos pontos de modificação, o processo evolutivo começa de fato. Na nossa primeira execução, nas gerações iniciais, nenhuma das modificações conseguiu um aprimoramento na função de aptidão. A primeira geração a escolher um dos dois pontos de modificação que representam de fato falhas no código foi a geração de número 7.

Código 19 – Geração 7 da execução

```
***** Generation 7 : 0
[...]
location: UniformRealDistribution.java184
[...]
---OP_INSTANCE:
InsertBeforeOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘
    -topatch--> ‘final double p = getProbabilityOfSuccess()‘ (spoon.
   support.reflect.code.CtLocalVariableImpl)
[...]
-The child does NOT compile: 22, errors: [UniformRealDistribution.java
   :185: error: cannot find symbol
[...]
```

O Astor escolheu a linha 184 do `UniformRealDistribution.java` para modificação, escolheu o operador que insere um ingrediente antes da linha atual (`InsertBeforeOp`). Finalmente, escolheu um ingrediente, `final double p = getProbabilityOfSuccess()`, retirado de outra parte do código. Como o ingrediente envolve uma função que está fora do escopo, o código não irá compilar.

O Astor finalmente conseguiu um aprimoramento na função de aptidão na geração 276, passando no teste para `FDistribution`, substituindo o `return true` pelo `return false` correto.

Apesar dessa sorte inicial, o Astor só escolheu esporadicamente a linha que faltava para o reparo, e nunca escolheu o operador correto seguido do ingrediente correto. O programa chega a 5000 gerações sem que a função vá além de 1. No total, testamos o método três vezes (para tentar compensar os fatores aleatórios, como no artigo original), sem obter êxito. O teste é feito três vezes diferentes apenas para obtermos a segurança que os fatores aleatórios do método evolutivo não irão influenciar nossa conclusão; ainda queremos que o programa seja capaz de achar a solução dentro de 5000 gerações, pois é um limite que seria considerado prático em uma situação real.

Esse resultado contrasta fortemente com o reparo do Math-5 (seção 2.4), que se deu em apenas 20 gerações. Como já vimos, no caso de bugs que necessitam de modificações em mais de uma linha, os métodos do Astor têm uma taxa de sucesso menor; mesmo usando a navegação evolutiva do GenProg. A necessidade de realizar múltiplas modificações ainda apresenta uma dificuldade adicional e acarreta numa demora maior para o conserto.

Outras diferenças claras entre o Math-5 e o Math-22 são o número de pontos de modificação e número de ingredientes. O primeiro ponto já foi discutido. O segundo se dá porque o Astor, por padrão, busca ingredientes que pertencem ao mesmo pacote (em Java) do arquivo a ser consertado. No caso do Math-22, o pacote dos dois arquivos com problema é consideravelmente maior que no caso do Math-5, resultando num espaço de busca maior, o que torna mais difícil que o programa encontre a resposta certa dentro dos

limites de recursos estipulados.

Concluímos que, apesar da estratégia de navegação evolutiva utilizada pelo GenProg permitir o aprimoramento gradual de programas, também requer que o reparo dependa de fatores aleatórios que podem ser extremamente limitantes em escalas maiores. A eficácia do jGenProg em outros bugs é comprovada pelos experimentos de (MARTINEZ; MONPERRUS, 2019), mas casos como esse apontam pontos cegos nas estratégias do programa.

Ainda assim, sabemos que o GenProg é teoricamente capaz de realizar as modificações necessárias, e podemos deduzir que dado tempo e número de gerações ilimitados, o programa consertaria o bug (tomando, porém, uma quantidade de tempo que não seria prática). Isso não se aplica a todos os métodos padrões do Astor: métodos que utilizam as estratégias exaustiva e seletiva só conseguem fazer uma única modificação, o que impede que eles resolvam o problema independente dos recursos disponíveis.

Também vale a pena destacar que o exemplo acima não representa, necessariamente, o resultado típico de uma execução do Astor. Lembramos que os testes de (MARTINEZ; MONPERRUS, 2019) obtiveram uma taxa de sucesso de 27.4%. Buscamos apenas chamar atenção a uma fraqueza existente no jGenProg, que pode ser aprimorada pela corretude relativa. Tais fraquezas são de se esperar, dado que o reparo automático de software ainda é uma área relativamente nova, que ainda está alcançando a viabilidade prática. Ainda assim, o exemplo serve para indicar quais são os obstáculos existentes para essa viabilidade.

Tendo essas falhas em mente, veremos, agora, o desempenho do RCFix no mesmo bug, verificando se o RCFix é capaz de cobrir algumas das inaptidões do Astor. Vamos usar os mesmos operadores que no jGenProg (substituição, remoção e inserção de seções do código). Buscamos mostrar que o algoritmo é capaz de reparar programas com profundidade de falhas arbitrária, contanto que tenha os operadores adequados em mãos.

Começamos pela localização de falhas. Os pontos de modificação criados são exatamente os mesmos da execução do Astor. Uma vez criados, o algoritmo irá navegar todos eles de forma exaustiva, em ordem decrescente do valor de suspeita. Para cada ponto, ele irá criar uma lista com todas as modificações possíveis e aplicá-las em ordem, descartando aquelas que não resultam em uma melhora.

Código 20 – Operadores no ponto de modificação 1

```
 MP (1/226) location to modify: MP=org.apache.commons.math3.distribution
    .UniformRealDistribution line: 184, pointed element: CtReturnImpl
[...]
Number modifications to apply: 111
--- List of operators (111) : [OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return inverseCumulativeProbability(random.nextDouble())‘
    (spoon.support.reflect.code.CtReturnImpl) , OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return 0‘ (spoon.support.reflect.code.CtReturnImpl) ,
   OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return java.lang.Double.NEGATIVE_INFINITY‘ (spoon.support
   .reflect.code.CtReturnImpl) , OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return java.lang.Double.POSITIVE_INFINITY‘ (spoon.support
   .reflect.code.CtReturnImpl) , OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return false‘ (spoon.support.reflect.code.CtReturnImpl) ,
    OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return solverAbsoluteAccuracy‘ (spoon.support.reflect.
   code.CtReturnImpl) , OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return 0.0‘ (spoon.support.reflect.code.CtReturnImpl) ,
   OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return 1‘ (spoon.support.reflect.code.CtReturnImpl) ,
   OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return false ‘ -
   topatch--> ‘return true‘ (spoon.support.reflect.code.CtReturnImpl) ,
   OP_INSTANCE:
[...]
```

A lista acima teve que ser apresentada parcialmente. No total, são criados 111 operadores, abrangendo as três operações possíveis do GenProg. Pelo que sabemos sobre o programa, o último operador apresentado será o suficiente para o reparo do primeiro arquivo. Os operadores acima serão aplicados em ordem, então o programa vai conseguir um aprimoramento ainda na décima geração, substituindo o `return false` em `UniformRealDistribution` por `return true`.

Ao realizar essa modificação, o RCFix irá rodar os testes negativos para verificar que houve uma melhora, e com essa verificação feita, irá rodar os testes positivos para confirmar que nenhum teste antigo passou a falhar no código novo. Ao obter essa confirmação, temos que o conjunto de testes em que o código antigo passa é um subconjunto estrito

do conjunto de testes em que o código novo passa. Isso quer dizer que o código novo é estritamente mais correto que o antigo, como definido anteriormente.

Uma vez que o RCFix obtém um programa estritamente mais correto, a localização de falhas é executada novamente e temos uma nova rodada de modificações, o que difere o RCFix da estratégia exaustiva do Astor. O programa novo irá criar 216 pontos de modificação.

Código 21 – Pontos de modificação novos no Math-22

```
Creating variant 1
--ModifPoint:CtReturnImpl, suspValue 1.0, line 275, file FDistribution.
    java
--ModifPoint:CtLocalVariableImpl, suspValue 0.5, line 129, file
    FDistribution.java
--ModifPoint:CtLocalVariableImpl, suspValue 0.5, line 130, file
    FDistribution.java
--ModifPoint:CtLocalVariableImpl, suspValue 0.5, line 131, file
    FDistribution.java
--ModifPoint:CtLocalVariableImpl, suspValue 0.5, line 132, file
    FDistribution.java
--ModifPoint:CtLocalVariableImpl, suspValue 0.5, line 133, file
    FDistribution.java
--ModifPoint:CtLocalVariableImpl, suspValue 0.5, line 134, file
    FDistribution.java
[...]
```

A linha 275 do `FDistribution.java`, como já vimos, é uma das localizações que precisa ser modificada para consertar o código, e o valor de suspeita passou a ser 1. A partir dos novos pontos de modificação, iremos novamente aplicar todos os operadores de forma exaustiva.

Código 22 – Operadores do primeiro ponto de modificação

```
 MP (1/216) location to modify: MP=org.apache.commons.math3.distribution
    .FDistribution line: 275, pointed element: CtReturnImpl
[...]
Number modifications to apply: 129
--- List of operators (129) : [OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return true ‘ -
   topatch--> ‘return inverseCumulativeProbability(random.nextDouble())‘
    (spoon.support.reflect.code.CtReturnImpl) , OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return true ‘ -
   topatch--> ‘return 0‘ (spoon.support.reflect.code.CtReturnImpl) ,
   OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return true ‘ -
   topatch--> ‘return java.lang.Double.NEGATIVE_INFINITY‘ (spoon.support
   .reflect.code.CtReturnImpl) , OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return true ‘ -
   topatch--> ‘return java.lang.Double.POSITIVE_INFINITY‘ (spoon.support
   .reflect.code.CtReturnImpl) , OP_INSTANCE:
ReplaceOp:(spoon.support.reflect.code.CtReturnImpl) ‘return true ‘ -
   topatch--> ‘return false‘ (spoon.support.reflect.code.CtReturnImpl) ,
    OP_INSTANCE:
[...]
```

O último operador na lista acima irá finalmente consertar o código. O programa resultante será absolutamente correto. O reparo leva 15 gerações, no total.

No caso acima, o RCFix toma vantagem do fato da localização de falhas atribuir corretamente às porções do código que precisam ser consertadas os maiores valores de suspeita, permitindo que a navegação exaustiva corrija as falhas rapidamente, diferente da estratégia evolutiva adotada pelo jGenProg. Isso é, enquanto a estratégia evolutiva escolhe pontos de modificação de forma aleatória, a exaustiva considera todas as modificações possíveis para todos os pontos, ordenados de acordo com seu valor de suspeita. Na implementação do Astor, que não leva em consideração o conceito de corretude relativa, essa navegação só pode gerar uma modificação por programa, e portanto está limitada a programas com profundidade de falhas igual a 1. O RCFix corrige este problema e permite que ela conserte o Math-22.

Como consequência, um bug que antes era irreparável (pois a quantidade de recursos que teria que ser consumida extrapolaria os limites definidos na execução) passa a ser consertado de forma eficiente.

4.4 PROFUNDIDADE DE FALHAS ARBITRÁRIA - MÉTODO EXAUSTIVO

Já vimos como o RCFix pode cobrir as inaptidões dos métodos evolutivos na hora de consertar bugs com profundidade de falhas arbitrária. Veremos agora uma situação

parecida, mas se tratando de um método exaustivo. O bug é retirado do compilador Closure[1], um compilador para JavaScript focado em melhora de código.

O Closure faz uso de várias funções para checagem de tipos na hora de compilação. O JavaScript é uma linguagem com tipagem dinâmica, onde informações sobre tipos são determinadas, em sua maioria, apenas na hora da execução. Assim, o próprio compilador se encarrega de implementar essas funções. Isso torna possível, por exemplo, verificar incompatibilidades de tipo antes da execução do programa.

O bug em questão está relacionado a duas funções do código `TypeValidator.java`. Ambas as funções têm um propósito semelhante: verificar que ambos os lados de uma operação de atribuição são compatíveis entre si. Isso é, verificar que o valor do lado direito da atribuição pode ser atribuído ao símbolo do lado esquerdo. As funções são chamadas de `expectCanAssignTo` e `expectCanAssignToPropertyOf`; a primeira serve para casos gerais, enquanto a segunda serve especificamente para casos onde o lado esquerdo da atribuição é propriedade de um objeto.

Código 23 – expectCanAssignTo()

```
1 boolean expectCanAssignTo(NodeTraversal t, Node n, JSType rightType,
      JSType leftType, String msg) {
2     if (!rightType.canAssignTo(leftType)) {
3         if ((leftType.isConstructor() || leftType.isEnumType()) && (
              rightType.isConstructor() || rightType.isEnumType())) {
4             registerMismatch(rightType, leftType, null);
5         } else {
6             mismatch(t, n, msg, rightType, leftType);
7         }
8         return false;
9     }
10    return true;
11 }
```

A função acima faz uso de uma outra função, `canAssignTo`, que testa a compatibilidade entre os tipos dos dois lados, usada na expressão condicional da linha 2. Se os tipos foram compatíveis, a linha 10 é executada e a função retorna `true`. As funções `registerMismatch` e `mismatch` ambas servem para registrar os momentos em que a incompatibilidade ocorre. O bloco das linhas 3-8 serve apenas para distinguir como esse registro deve acontecer. A diferença principal é que, caso algum dos tipos seja um construtor ou um tipo enumerável, a função `registerMismatch` era originalmente capaz de identificar não só a linha onde a incompatibilidade ocorreu, mas também as linhas onde o construtor ou tipo enumerável foram definidos inicialmente. Entretanto, outras diferenças internas na implementação das duas funções de registro também acarretam em diferenças na forma como o registro é feito, tal que `mismatch` considera a incompatibilidade como um

[1] https://github.com/google/closure-compiler

erro de compilação e `registerMismatch` não. A função `expectCanAssignToPropertyOf` faz exatamente a mesma checagem que `expectCanAssignTo` faz nas linhas 3-8.

O bug surgiu quando os testes relevantes do Closure passaram a exigir o retorno desse erro de compilação, sem que as funções fossem devidamente atualizadas.

Código 24 – Teste 1 do Closure-6

```
1 public void testTypeRedefinition() throws Exception {
2     testClosureTypesMultipleWarnings(
3         "a={};/**@enum {string}*/ a.A = {ZOR:'b'};"
4         + "/** @constructor */ a.A = function() {}",
5         Lists.newArrayList(
6             "variable a.A redefined with type function (new:a.A):
                 undefined, " +
7             "original definition at [testcode]:1 with type enum{a.A}",
8             "assignment to property A of a\n" +
9             "found   : function (new:a.A): undefined\n" +
10            "required: enum{a.A}"));
11 }
```

O teste acima faz uso de uma função `testClosureTypesMultipleWarnings` (criada especialmente para testes) que funciona da seguinte forma: o primeiro parâmetro é uma porção de código em JavaScript a ser analisada pelo compilador, enquanto o segundo é uma lista contendo dois erros que o teste espera receber. A forma atual do programa só retorna o primeiro erro. Esse teste está presente, de forma idêntica, em dois arquivos de teste do programa, e por isso é contado como duas falhas diferentes. O teste restante funciona de forma semelhante, usando uma função diferente que também recebe uma porção de código e um erro esperado.

Código 25 – Teste 2 do Closure-6

```
1 public void testIssue635b() throws Exception {
2     testTypes(
3         "/** @constructor */" +
4         "function F() {}" +
5         "/** @constructor */" +
6         "function G() {}" +
7         "/** @type {function(new:G)} */ var x = F;",
8         "initializing variable\n" +
9         "found   : function (new:F): undefined\n" +
10        "required: function (new:G): ?");
11 }
```

Como já dito, o `registerMismatch` não retorna esses erros, então os testes irão falhar, pois fazem uso de tipos enumerados e construtores. O reparo oficial, feito pelos desenvolvedores do compilador, envolve remover a checagem da linha 3 da `expectCanAssignTo` para que todos os casos façam uso do `mismatch`. Os desenvolvedores optaram por desconsiderar a funcionalidade especial do `registerMismatch`. A remoção também é feita na

função `expectCanAssignToPropertyOf`, então o código precisa ser modificado em duas seções para passar nos três testes.

Código 26 – expectCanAssignTo() consertado

```
1 boolean expectCanAssignTo(NodeTraversal t, Node n, JSType rightType,
      JSType leftType, String msg) {
2     if (!rightType.canAssignTo(leftType)) {
3         mismatch(t, n, msg, rightType, leftType);
4         return false;
5     }
6     return true;
7 }
```

O método jKali é capaz de trocar expressões condicionais por `false`; se aplicado na linha 3 do `expectCanAssignTo` original, o código irá funcionar da mesma maneira do código consertado oficial. Portanto, ele deve ser capaz de consertar esse bug.

Vamos primeiro rodar o Astor no programa explicado acima. A localização de falha cria 794 pontos de modificação. Embora esse número seja alto, o número de operações permitidas pelo jKali é bem menor do que numa execução típica do jGenProg. Portanto, ainda teremos uma quantidade razoável de variantes geradas.

O jKali é um método exaustivo, então ele irá passar por todos os pontos de modificação e testar todos os operadores possíveis, um de cada vez. Notamos que, diferente de métodos evolutivos como o jGenProg, métodos exaustivos testam todos os operadores sempre em ordem, então todas as execuções irão proceder da mesma maneira, sem influência de fatores aleatórios. Cada variante gerada terá uma única diferença do programa original, então deduzimos que o método não será capaz de realizar as duas modificações necessárias simultaneamente. De fato, verificamos que ele chega ao final da execução sem encontrar uma solução.

Vamos em seguida testar o RCFix com os operadores do jKali. Como na execução do Astor, 794 pontos de modificação são criados. Apenas no ponto de número 21 temos uma das expressões que precisa ser modificada:

Código 27 – Conserto do Closure-6 parte 1

```
ReplaceIfBooleanOp:(spoon.support.reflect.code.CtIfImpl) `if ((leftType.
    isConstructor() || leftType.isEnumType()) && (rightType.isConstructor
    () || rightType.i[...] ` -topatch--> `if (false) { registerMismatch(
    rightType, leftType, null); } else { [...]` (spoon.support.reflect.
    code.CtIfImpl)
[...]
- SRC Found Solution, child variant number 64
```

Essa solução irá consertar a função `expectCanAssignToPropertyOf`, gerando um programa estritamente mais correto. Esse programa novo passará a ser usado como uma base nova, e a localização de falhas será executada novamente. São gerados 836 pontos de mo-

dificação no programa novo. Eventualmente a modificação equivalente será feita na função `expectCanAssignTo`, gerando um programa absolutamente correto.

Código 28 – Conserto do Closure-6 parte 2

```
ReplaceIfBooleanOp:(spoon.support.reflect.code.CtIfImpl) ‘if ((leftType.
    isConstructor() || leftType.isEnumType()) && (rightType.isConstructor
    () || rightType.i[...] ‘ -topatch--> ‘if (false) { registerMismatch(
    rightType, leftType, null); } else { mismatch(t, n, msg, rightType,
    [...]‘ (spoon.support.reflect.code.CtIfImpl)
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

Como havíamos deduzido, o fato do método original ser exaustivo impede que o bug seja consertado pelo Astor, pois no Astor, a navegação exaustiva só permite que uma modificação seja feita por vez e não leva em conta resultados intermediários. O RCFix com corretude relativa é capaz de identificar uma variante como estritamente mais correta, a partir da qual modificações adicionais podem ser feitas. Assim, conseguimos usar os operadores do jKali para obter um resultado que não havia sido possível na abordagem do Astor.

4.5 MATH-80, MATH-81 E MATH-84: MÚLTIPLOS BUGS SIMULTÂNEOS

Já mostramos a capacidade do RCFix em lidar com uma profundidade de falhas arbitrária. Usamos uma única versão do Commons-Math disponível no Defects4J para uma dessas demonstrações. No geral, as versões disponibilizadas pelo Defects4J podem ter profundidade ou densidade de falhas maior que 1, mas cada uma ainda pode ser considerada como um único "bug", pois as falhas estão correlacionadas e foram resolvidas (durante o desenvolvimento real do programa) com um único patch. Para garantir que esse fator não resulte em um experimento enviesado, é de interesse testar a capacidade do RCFix em lidar com múltiplos bugs (isso é, múltiplas versões, cada uma das quais pode ter uma ou mais falhas) ao mesmo tempo (misturados em uma única instância do programa).

Para esse experimento, vamos usar o método jMutRepair, que troca operadores lógicos e relacionais existentes em um programa por outros da mesma categoria (ou seja, operadores lógicos são trocados por outros operadores lógicos, e relacionais por relacionais), navegando o espaço de busca de forma exaustiva. Usaremos os bugs Math-80, Math-81 e Math-84, cada um dos quais pode ser reparado (individualmente) pelo jMutRepair. O programa final que queremos consertar contém todas as falhas de cada uma dessas versões.

Como já sabemos de antemão que os bugs são resolvidos com sucesso, não vamos nos preocupar em explicar o funcionamento e resolução de cada um com detalhe. O que é relevante para nós é a interação que eles têm entre si, ao fazerem parte de uma mesma versão do programa. Assim, vamos olhar brevemente a mudança feita para consertar cada

um.

No resolução do Math-80, o jMutRepair irá modificar o operador relacional de uma condição no arquivo `EigenDecompositionImpl.java`, trocando o `<` na expressão `(1.5 * work[pingPong]) < work[(4 * (n - 1)) + pingPong]` por um `==`, obtendo um programa correto como solução.

Código 29 – Solução do Math-80

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) ‘(1.5
    * work[pingPong]) < work[(4 * (n - 1)) + pingPong] ‘ -topatch-->
   ‘(1.5 * work[pingPong]) == work[(4 * (n - 1)) + pingPong]‘ (spoon.
   support.reflect.code.CtBinaryOperatorImpl)
[...]
-The child compiles: id 394
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

No Math-81, irá adicionar um operador lógico ! (not) à condição `(dMin1 == dN1) && (dMin2 == dN2)`, presente no mesmo arquivo que foi modificado no Math-80.

Código 30 – Solução do Math-81

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) ‘(
   dMin1 == dN1) && (dMin2 == dN2) ‘ -topatch--> ‘!((dMin1 == dN1) && (
   dMin2 == dN2))‘ (spoon.support.reflect.code.CtUnaryOperatorImpl)
[...]
-The child compiles: id 57
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

No Math-84, troca o operador relacional `<` na condição `comparator.compare(contracted, best) < 0` do arquivo `MultiDirectional.java` por um `==`.

Código 31 – Solução do Math-84

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) ‘
   comparator.compare(contracted, best) < 0 ‘ -topatch--> ‘comparator.
   compare(contracted, best) == 0‘ (spoon.support.reflect.code.
   CtBinaryOperatorImpl)
[...]
-The child compiles: id 8
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

Embora resolva individualmente cada um dos bugs acima, o jMutRepair falha na resolução da versão mista. Primeiramente, na criação dos pontos de modificação, apenas dois dos três pontos necessários são criados.

Código 32 – Pontos de modificação na versão mista

```
Creating variant 1
--ModifPoint:CtIfImpl, suspValue 0.5, line 74, file MultiDirectional.
   java
--ModifPoint:CtIfImpl, suspValue 0.5, line 90, file MultiDirectional.
   java
[...]
--ModifPoint:CtIfImpl, suspValue 0.25, line 1587, file
   EigenDecompositionImpl.java
[...]
```

A linha 90 do arquivo `MultiDirection.java` é a que é modificada para consertar o Math-84, enquanto a linha 1587 do `EigenDecompositionImpl.java` é a que conserta o Math-81. A linha 1134, que deveria ser modificada para resolver o bug Math-80, não tem um ponto de modificação criado nela.

Uma explicação possível é que as falhas do Math-81 impedem com que o código relevante do Math-80 seja executado. Isso é, os testes do Math-80, na versão individual, executam uma dada sequência de instruções, mas as falhas introduzidas pelo Math-81 impedem (pela nossa hipótese) que todas essas instruções sejam executadas na versão mista. Os testes do Math-80 ainda falham, mas sem passar pelas mesmas porções de código de antes. Como a atribuição de valores de suspeita se baseia na frequência com que uma linha de código é executada durante os testes negativos, a linha de código correta não tem um valor alto o suficiente (dentro do limiar) para que um ponto de modificação seja criado. Se essa hipótese for verdadeira, verificaremos que o RCFix será capaz de criar tal ponto de modificação apenas após o conserto das falhas do Math-81.

Logo no início da execução, o jMutRepair irá realizar a modificação que consertaria o Math-84. Como essa modificação por si só não resulta em um programa que passa em todos os testes, ela é descartada e o algoritmo passa a tentar as outras modificações possíveis. O mesmo irá acontecer om a modificação que consertaria o Math-81, como visto abaixo.

Código 33 – Execução do jMutRepair

```
mod_point MP=org.apache.commons.math.linear.EigenDecompositionImpl line:
    1587, pointed element: CtIfImpl
-->op: OP_INSTANCE:
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `(
   dMin1 == dN1) && (dMin2 == dN2) ` -topatch--> `!((dMin1 == dN1) && (
   dMin2 == dN2))` (spoon.support.reflect.code.CtUnaryOperatorImpl)
[...]
TR: Success: false, failTest= 3, was successful: false, cases executed:
   23] ,[]
```

Ambas as modificações resultam em um programa estritamente mais correto, mas como o Astor não leva esse conceito em consideração, esse aprimoramento não é aprovei-

tado. Isso ocorre como consequência da navegação exaustiva, que no Astor, testará cada modificação uma única vez, não sendo capaz de criar um programa com múltiplas modificações. Após 773 gerações, não há mais modificações para serem testadas e o algoritmo falha.

Agora vamos testar o RCFix, fazendo uso dos operadores do jMutRepair. Os pontos de modificação criados são os mesmos do Astor, e também não incluem a linha 1134 do `EigenDecompositionImpl.java` inicialmente. Já na oitava geração, uma versão estritamente mais correta do programa é criada, resolvendo as falhas do bug Math-84.

Código 34 – Execução do jMutRepair com corretude relativa

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) ‘
    comparator.compare(contracted, best) < 0 ‘ -topatch--> ‘comparator.
    compare(contracted, best) == 0‘ (spoon.support.reflect.code.
    CtBinaryOperatorImpl)
-The child compiles: id 8
[...]
- SRC Found Solution, child variant number 8
```

O RCFix passará a usar essa nova versão como base e irá executar a localização de falhas novamente. Os pontos de modificação novos ainda não incluirão a linha 1134, mas a linha 1587 (que resolve o Math-81) passará a ter um valor de suspeita maior, pois o fato de um dos bugs ter sido consertado quer dizer que há menos testes negativos, e portanto, a linha aparece em uma proporção maior dos testes negativos. As falhas do Math-81 são consertadas em seguida sem dificuldades.

Código 35 – Execução do jMutRepair com corretude relativa parte 2

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) ‘(
    dMin1 == dN1) && (dMin2 == dN2) ‘ -topatch--> ‘!((dMin1 == dN1) && (
    dMin2 == dN2))‘ (spoon.support.reflect.code.CtUnaryOperatorImpl)
-The child compiles: id 57
[...]
- SRC Found Solution, child variant number 57
```

Novamente, a versão resultante servirá como a nova base e a localização de falhas será executada novamente. Dessa vez, como conjecturamos, a linha que conserta o Math-80 (1134 do `EigenDecompositionImpl.java`) finalmente tem um ponto de modificação associado.

Código 36 – Execução do jMutRepair com corretude relativa parte 3

```
Creating variant 1
--ModifPoint:CtIfImpl, suspValue 1.0, line 943, file
    EigenDecompositionImpl.java
--ModifPoint:CtReturnImpl, suspValue 1.0, line 1093, file
    EigenDecompositionImpl.java
--ModifPoint:CtIfImpl, suspValue 1.0, line 1343, file
    EigenDecompositionImpl.java
--ModifPoint:CtIfImpl, suspValue 0.24253562503633297, line 661, file
    EigenDecompositionImpl.java
--ModifPoint:CtIfImpl, suspValue 0.24253562503633297, line 834, file
    EigenDecompositionImpl.java
--ModifPoint:CtReturnImpl, suspValue 0.24253562503633297, line 836, file
     EigenDecompositionImpl.java
--ModifPoint:CtIfImpl, suspValue 0.24253562503633297, line 870, file
    EigenDecompositionImpl.java
--ModifPoint:CtIfImpl, suspValue 0.24253562503633297, line 874, file
    EigenDecompositionImpl.java
--ModifPoint:CtIfImpl, suspValue 0.24253562503633297, line 956, file
    EigenDecompositionImpl.java
--ModifPoint:CtIfImpl, suspValue 0.24253562503633297, line 1020, file
    EigenDecompositionImpl.java
--ModifPoint:CtReturnImpl, suspValue 0.24253562503633297, line 1084,
    file EigenDecompositionImpl.java
--ModifPoint:CtReturnImpl, suspValue 0.24253562503633297, line 1121,
    file EigenDecompositionImpl.java
--ModifPoint:CtIfImpl, suspValue 0.24253562503633297, line 1134, file
    EigenDecompositionImpl.java
[...]
```

Assim, o algoritmo será capaz de realizar o reparo final.

Código 37 – Execução do jMutRepair com corretude relativa parte 4

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `(1.5
     * work[pingPong]) < work[(4 * (n - 1)) + pingPong] ` -topatch-->
    `(1.5 * work[pingPong]) == work[(4 * (n - 1)) + pingPong]` (spoon.
    support.reflect.code.CtBinaryOperatorImpl)
-The child compiles: id 73
[...]
-Found Solution, child variant #73
```

Com a última modificação, chegamos no programa absolutamente correto.

O experimento acima ressalta algumas características importantes do RCFix. Primeiramente, reforça a possibilidade de aplicar um método que originalmente usava navegação exaustiva (dessa vez o jMutRepair) em um programa com profundidade de falhas maior do que 1. Além disso, apresenta especificamente a possibilidade de consertar um programa

com múltiplos bugs distintos e não apenas falhas correlacionadas. Finalmente, mostra a importância da localização de falhas ser executada sempre que se obtém um aprimoramento na corretude relativa do programa. Se o algoritmo usasse os mesmos pontos de modificação definidos inicialmente em toda a execução, não seria possível consertar as falhas do Math-80, pois a linha relevante não seria identificada como um ponto de modificação possível.

Esse último ponto quer dizer que mesmo um método evolutivo como o jGenProg não seria capaz de consertar o programa misto. Mesmo com o método sendo capaz de consertar um programa com profundidade de falhas arbitrária, o fato da localização de falhas ser executada apenas uma vez quer dizer que as falhas do Math-80 não seriam consideradas pelo algoritmo. Apenas o RCFix é capaz de reparar a versão mista.

4.6 COMBINAÇÃO DE MÉTODOS

Para esse experimento, buscamos verificar que a corretude relativa permite consertar programas cujos diferentes bugs não são resolvidos pelo mesmo método. Uma única execução do RCFix só é capaz de usar um único método, então estritamente falando, não seria possível consertar um programa desse tipo de uma vez só. Mas, diferente do Astor, o RCFix ainda será capaz de identificar e retornar um programa estritamente mais correto, a partir do qual outros reparos podem ser feitos, eventualmente chegando em um programa absolutamente correto.

O bug Math-58 é consertado pelo jMutRepair, mas não pelo jGenProg, enquanto o bug Math-53 é consertado pelo jGenProg mas não pelo jMutRepair. O reparo do Math-58 feito pelo jMutRepair consiste na troca de um `<=` em uma operação de comparação por um `==`.

Código 38 – Conserto do Math-58

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) ‘
   param[2] <= 0 ‘ -topatch--> ‘param[2] == 0‘ (spoon.support.reflect.
   code.CtBinaryOperatorImpl)
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

O reparo do Math-53 feito pelo jGenProg envolve inserir uma expressão condicional antes de uma operação de retorno existente.

Código 39 – Conserto do Math-53

```
InsertBeforeOp:(spoon.support.reflect.code.CtReturnImpl) ‘return
   createComplex(real + rhs.getReal(),  imaginary + rhs.getImaginary())
   ‘ -topatch--> ‘if (isNaN || rhs.isNaN) {  return org.apache.commons.
   math.complex.Complex.NaN; }‘ (spoon.support.reflect.code.CtIfImpl)
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

Entretanto, ambos os métodos falham ao serem executados com a versão combinada dos dois bugs, retornando apenas uma mensagem dizendo que a solução não foi encontrada.

Ao rodar o RCFix com os operadores do jMutRepair, porém, conseguimos um programa estritamente mais correto, mas falhamos posteriormente sem achar uma solução absolutamente correta.

Código 40 – Conserto parcial feito pelo jMutRepair

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) ‘
   param[2] <= 0 ‘ -topatch--> ‘param[2] == 0‘ (spoon.support.reflect.
   code.CtBinaryOperatorImpl)
[...]
- SRC Found Solution , child variant number 113
```

Podemos, entretanto, usar o programa estritamente mais correto retornado pelo RCFix como uma base nova para reparo. Aplicando o RCFix com os operadores do jGenProg, conseguimos finalmente o programa absolutamente correto.

Código 41 – Conserto final feito pelo jGenProg

```
InsertBeforeOp:(spoon.support.reflect.code.CtReturnImpl) ‘return
   createComplex(real + rhs.getReal(),  imaginary + rhs.getImaginary())
   ‘ -topatch--> ‘if (isNaN || rhs.isNaN) {  return org.apache.commons.
   math.complex.Complex.NaN; }‘ (spoon.support.reflect.code.CtIfImpl)
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

Esse experimento é importante pois, no mundo real, é improvável que todos os bugs presentes em um mesmo programa possam ser consertados pelo mesmo método. Assim, mesmo com as capacidades vistas nas seções anteriores, uma única execução do RCFix ainda pode falhar em achar uma solução absolutamente correta, mas a capacidade de ainda retornar um programa estritamente mais correto permite o reparo eventual de todos os bugs do programa.

4.7 MÚLTIPLAS MODIFICAÇÕES PARA UM MESMO TESTE

Para nosso último experimento, vamos usar dois bugs da biblioteca Commons-Lang[2]. Ambos os bugs já são consertados pelo método DeepRepair do Astor, que é um método evolutivo. Ambos os bugs também falham no mesmo teste. Se ambos os bugs estiverem presentes na mesma versão do programa, isso apresentará um problema para o método usado até aqui para avaliar a corretude, pois consertar apenas um dos bugs não irá fazer com que o programa passe no teste. Mesmo os métodos evolutivos do Astor só podem fazer uma modificação por vez e testar cada uma individualmente para verificar se há uma mudança na função de aptidão. Como o programa só se tornaria estritamente mais correto ao reparar ambos os bugs, pela nossa definição de falhas, os dois bugs contam como uma única falha.

Consertar uma combinação dos bugs Lang-7 e Lang-27 requer que o programa seja capaz de gerar uma versão com múltiplas modificações sem precisar que nenhum teste seja feito no meio. O RCFix é capaz de fazer isso devido ao conceito de multiplicidade.

Como há mais de um bug presente no programa, faz sentido aumentar o número máximo de gerações, já que o número que usamos até aqui foi baseado no tempo que se costuma levar para consertar um único bug. Para ter certeza de que o DeepRepair vai ter chances o suficiente para consertar o programa, dadas essas circunstâncias, aumentamos o número de gerações para 10000. Ainda assim, ao rodar o DeepRepair na versão combinada do programa, chegamos ao final sem conseguir uma solução.

Rodamos, então, o RCFix com os operadores do DeepRepair, com o número modificado de gerações. No final da localização de falhas, são criados 219 pontos de modificação. O RCFix, na teoria, irá rodar todos os operadores em todos os pontos de modificação antes de aumentar a multiplicidade. Quando isso acontecer, o programa irá retornar ao primeiro ponto de modificação, e passar a tentar modificações em múltiplas localizações.

Entretanto, com uma quantidade tão alta de pontos de modificação, o número de operadores que deve ser testado antes de se aumentar a multiplicidade já é extremamente alto. Após aumentar a multiplicidade, temos que navegar de forma exaustiva o espaço de todas as mudanças duplas possíveis. Isso é, precisamos testar todas as mudanças simultâneas em duas seções de código até achar uma solução. Isso é extremamente custoso, e o RCFix também falha nessa tarefa, esgotando o limite de gerações. Não parece ser possível consertar a combinação do Lang-7 e Lang-27 dentro de um tempo razoável. Além disso, como os pontos de modificação necessários para o conserto não estão nas linhas de maior suspeita, diminuir o limiar de valor de suspeita na esperança de reduzir o número de pontos de modificação também não ajuda na viabilidade do reparo.

O Defects4J não providencia nenhum outro candidato claro para testar essa funcionalidade do RCFix, então não seremos capazes de verificar a viabilidade em um caso real.

[2] https://github.com/apache/commons-lang

Apenas para demonstrar que a funcionalidade existe, optamos por usar um caso artificial baseado no bug Math-50. Esse bug é consertado pelo jMutRepair do Astor. Ele modifica uma condição no arquivo `BaseSecantSolver.java`, trocando de `x == x1` para `x > x1`. Isso faz que o bloco condicional não seja executado durante o teste.

Código 42 – Conserto do Math-50

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) ‘x ==
    x1 ‘ -topatch--> ‘x > x1‘ (spoon.support.reflect.code.
   CtBinaryOperatorImpl)
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

Fazemos uma modificação no programa tal que o mesmo teste irá falhar por causa de duas seções de código diferentes:

Código 43 – Versão do Math-50 para o experimento

```
1 [...]
2 case REGULA_FALSI:
3     // Nothing.
4     if (x == x1) {
5         x0 = 0.5 * (x0 + x1 - FastMath.max(rtol * FastMath.abs(x1), atol
              ));
6         f0 = computeObjectiveValue(x0);
7     }
8     if (x == x1) {
9         x0 = 0.5 * (x0 + x1 - FastMath.max(rtol * FastMath.abs(x1), atol
              ));
10        f0 = computeObjectiveValue(x0);
11    }
12    break;
13 [...]
```

Originalmente o `case` acima só continha as linhas 4-7 e 12. Nós replicamos as linhas 4-7. Isso não afetará em quais testes o programa irá falhar, pois o resultado do `case` será o mesmo. Mas isso tornará necessário realizar duas modificações para passar no teste. Esse programa só pode ser consertado com o uso da multiplicidade. Mesmo usando um limiar baixo de 0.01, teremos uma quantidade razoavelmente pequena de pontos de modificação: um total de 30. Os dois pontos necessários são os dois no topo da lista, com valor de suspeita de 0,3779644730092272.

Código 44 – Pontos de modificação do Math-50 modificado

```
Creating variant 1
--ModifPoint:CtIfImpl, suspValue 0.3779644730092272, line 187, file
    BaseSecantSolver.java
--ModifPoint:CtIfImpl, suspValue 0.3779644730092272, line 191, file
    BaseSecantSolver.java
--ModifPoint:CtReturnImpl, suspValue 0.3779644730092272, line 243, file
    BaseSecantSolver.java
--ModifPoint:CtReturnImpl, suspValue 0.31622776601683794, line 124, file
     BaseSecantSolver.java
--ModifPoint:CtIfImpl, suspValue 0.21320071635561041, line 171, file
    BaseSecantSolver.java
[...]
```

A implementação padrão do jMutRepair, no Astor, não conserta esse programa modificado. O RCFix, usando os operadores do jMutRepair, irá esgotar todas as modificações possíveis para uma multiplicidade de valor 1, mas em seguida irá testar modificações de multiplicidade 2. Para cada variante gerada durante o laço de multiplicidade 1, o algoritmo irá testar todos os operadores novamente, gerando variantes com duas modificações de diferença do programa original. Tanto pelo número de pontos de modificação quanto pelo número pequeno de modificações que o jMutRepair é capaz de fazer, a quantidade de variantes será bem menor do que no nosso experimento com o Lang-7 e Lang-27, o que torna possível que o programa teste as modificações de multiplicidade 2 dentro de uma quantidade razoável de tempo.

Ao testar modificações de multiplicidade 2 para a variante que modifica corretamente a primeira expressão condicional (linha 4 no programa acima), o algoritmo irá, eventualmente, testar o operador que modifica a segunda expressão condicional (linha 8). Isso resultará em uma variante correta.

Código 45 – Resultado do Math-50 modificado

```
IfExpresionMutOp:(spoon.support.reflect.code.CtBinaryOperatorImpl) `x ==
    x1 ` -topatch--> `x > x1` (spoon.support.reflect.code.
    CtBinaryOperatorImpl)
[...]
----SUMMARY_EXECUTION---
End Repair Search: Found solution
```

O experimento acima mostra a capacidade do RCFix de consertar falhas elementares espalhadas por múltiplas seções de código. Entretanto, não conseguimos testar essa capacidade em um caso do mundo real. Fica claro que a viabilidade desse tipo de reparo será limitada pelo número de pontos de modificação e de operadores possíveis, pelo menos em sua forma atual.

4.8 RESUMO DOS APRIMORAMENTOS DO RCFIX

Nos experimentos acima, foram realizados apenas estudos de caso, devido à falta do tempo e recursos computacionais necessários para fazer testes exaustivos em todos os bugs do Defects4J. Os testes com métodos evolutivos precisam de um número alto de gerações (pelo menos 5000) para verificar a sua inviabilidade, e precisam ser feitos múltiplas vezes para que se possa compensar os fatores aleatórios. Testes com métodos exaustivos podem alcançar números semelhantes de gerações (dependendo do método e do programa a ser reparado), ou até maiores. O tempo levado por cada geração depende dos operadores de mutação, do programa a ser reparado e da capacidade da máquina usada para teste.

Com os recursos computacionais disponíveis para nós, alguns testes poderiam demorar mais de quatro horas. Considerando o número de bugs disponíveis no Defects4J (395), a necessidade de testar todos os métodos para cada bug, e a necessidade de realizar múltiplos testes para os métodos evolutivos, o tempo total para um experimento exaustivo iria extrapolar o tempo que nos foi disponibilizado.

Buscamos explorar capacidades do RCFix que não estão presentes no Astor. Entre elas, destacamos:

a) Reparar programas com profundidade de falhas arbitrária, independente do método

b) Reparar múltiplos bugs distintos não correlacionados presentes em um único programa

c) Encontrar falhas novas ao achar um programa estritamente mais correto, que antes não teriam sido detectadas pela localização de falhas

d) Usar a noção de corretude relativa para misturar métodos distintos

e) Consertar falhas elementares que estejam espalhadas por múltiplas seções de código, apesar de apenas em uma situação artificial

Algumas dessas capacidades são importantes para reparar bugs singulares de datasets como o Defects4J, enquanto outras são importantes para situações reais onde é natural que um único programa apresente múltiplas falhas. Como só conseguimos explorar essas capacidades em estudos de caso, entretanto, é natural examinar com que facilidade os resultados podem ser generalizados para outros casos. Não há garantias que o Defects4J irá apresentar outras situações semelhantes àquelas estudadas acima.

Porém, todos os casos estudados são razoavelmente amplos. Por exemplo, a melhoria aplicada na seção 4.4 será relevante para qualquer caso que apresente múltiplas falhas que só podem ser consertadas por um método não evolutivo (como o jMutRepair). Nossa análise do Defects4J Dissection nos leva a crer que tais situações serão relativamente comuns, já que em 297 dos 395 bugs, o reparo manual precisou modificar mais de uma linha. Embora o reparo gerado por um método de reparo automático não precise corresponder exatamente ao reparo manual (só é necessário que o programa resultante passe nos testes),

na prática os reparos automáticos costumam ser semelhantes aos manuais. Uma lógica semelhante pode ser aplicada à melhoria da seção 4.3. Os demais estudos de caso podem ser justificados com intuições sobre como bugs aparecem no mundo real. No caso da seção 4.5, por exemplo, esperamos que um programa grande o suficiente apresentará múltiplos bugs distintos em um dado instante, o que tornaria inviável o uso de métodos exaustivos do Astor.

Assim, há motivos para acreditar que os resultados dos estudos de caso poderão ser generalizados tanto para casos de teste de datasets como o Defects4J, quanto para casos reais mais complexos.

5 CONCLUSÃO

A área de reparo automático de programas é recente comparada a demais áreas da computação, com publicações só se tornando frequentes a partir da década de 2010 (GAZZOLA; MICUCCI; MARIANI, 2019). O desenvolvimento de métodos elaborados de geração e validação como o GenProg permitiu o conserto de bugs reais como os presentes no Defects4J, enquanto o Astor ajudou a ampliar o alcance de métodos como esse através de um framework geral personalizável.

Entretanto, como buscamos apontar, esse framework ainda possui fraquezas. Podemos enxergar o Astor como um passo intermediário no caminho até um reparo de bugs viável no mundo real, reunindo muitas das descobertas importantes feitas na área até o seu desenvolvimento, mas ainda enfrentando obstáculos.

Acreditamos que os pontos cegos explorados no capítulo anterior podem contribuir para a inviabilidade do Astor em certos contextos práticos. Porém, nossos estudos de caso indicam que o conceito de corretude relativa pode ser capaz de remediar a maioria deles. Assim, existe a possibilidade do RCFix servir como não só uma melhora direta a um framework existente, mas também com um novo paradigma para os métodos de geração e validação. Esperamos que os nossos resultados, construídos em cima dos fundamentos fornecidos por (KHAIREDDINE; MARTINEZ; MILI, 2019), indiquem a importância da corretude relativa nesse sentido, apontando para inovações adicionais possíveis.

5.1 PROBLEMAS ENFRENTADOS E TRABALHOS FUTUROS

O maior obstáculo foi a inviabilidade de realizar experimentos exaustivos em todos os bugs do Defects4J, como feito no artigo original do Astor (MARTINEZ; MONPERRUS, 2019), devido aos custos computacionais envolvidos. Tais experimentos seriam a melhor maneira de avaliar a importância da corretude relativa. Optamos por apenas mostrar possíveis efeitos, sem sermos capazes de avaliar o impacto geral na prática. Essa é, talvez, a possibilidade mais importante a ser explorada por um trabalho futuro.

O RCFix, especificamente, é construído em cima do Astor e faz uso de alguns de seus conceitos principais. Métodos que não fazem parte do framework do Astor também poderiam apresentar melhores resultados com o uso da corretude relativa. Seria especialmente interessante aplicar essa ideia em métodos com alta taxa de sucesso como o NOPOL (XUAN et al., 2017).

Não foi possível verificar a utilidade da multiplicidade em casos reais, e o experimento que fizemos nos leva a crer que seu uso será inviável em muitos casos reais. Assim, sugerimos não só testes mais aprofundados dessa característica do RCFix, como tentativas posteriores de melhorar a sua viabilidade. O uso de heurísticas para reduzir o número de

candidatos pode ajudar nessa questão.

REFERÊNCIAS

ACKLING, T.; ALEXANDER, B.; GRUNERT, I. Evolving patches for software repair. In: KRASNOGOR, N.; LANZI, P. L. (Ed.). **GECCO**. ACM, 2011. p. 1427–1434. ISBN 978-1-4503-0557-0. Disponível em: http://dblp.uni-trier.de/db/conf/gecco/gecco2011.html#AcklingAG11.

ARCURI, A. Evolutionary repair of faulty software. **Applied Soft Computing**, v. 11, n. 4, p. 3494–3514, 2011.

CAMPOS, J. et al. Gzoltar: an eclipse plug-in for testing and debugging. In: **Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering**. [S.l.]: ACM, 2012. p. 378–381. ISBN 978-1-4503-1204-2.

DEBROY, V.; WONG, W. E. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In: **Proceedings of the Third International Conference on Software Testing, Verification and Validation**. [S.l.]: IEEE Computer Society, 2010. p. 65–74. ISBN 978-0-7695-3990-4.

DIALLO, N. et al. What is a fault? and why does it matter? **Innov. Syst. Softw. Eng.**, v. 13, n. 2-3, p. 219–239, 2017. Disponível em: https://doi.org/10.1007/s11334-017-0300-7.

EYOLFSON, J.; TAN, L.; LAM, P. Do time of day and developer experience affect commit bugginess. In: **Proceedings of the Eighth International Working Conference on Mining Software Repositories**. [S.l.]: ACM, 2011. p. 153–162. ISBN 978-1-4503-0574-7.

GAZZOLA, L.; MICUCCI, D.; MARIANI, L. Automatic software repair: a survey. **IEEE Trans. Software Eng.**, v. 45, n. 1, p. 34–67, 2019.

GOUES, C. L. et al. Genprog: A generic method for automatic software repair. **IEEE Trans. Software Eng.**, v. 38, n. 1, p. 54–72, 2012. Disponível em: http://dblp.uni-trier.de/db/journals/tse/tse38.html#GouesNFW12.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: **Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)**. San Jose, CA, USA: [s.n.], 2014. p. 437–440.

KHAIREDDINE, B.; MARTINEZ, M.; MILI, A. Program repair at arbitrary fault depth. In: **2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)**. [S.l.: s.n.], 2019. p. 465–472.

KOU, R.; HIGO, Y.; KUSUMOTO, S. A capable crossover technique on automatic program repair. In: **2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)**. [S.l.: s.n.], 2016. p. 45–50.

LATOZA, T. D.; VENOLIA, G.; DELINE, R. Maintaining mental models: a study of developer work habits. In: **ICSE '06: Proceedings of the 28th international**

conference on Software engineering. New York, NY, USA: ACM, 2006. p. 492–501. ISBN 1-59593-375-1. Disponível em: http://portal.acm.org/citation.cfm?id=1134355.

LIN, D. et al. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In: **Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017**. [s.n.], 2017. p. 55–56. Disponível em: https://doi.org/10.1145/3135932.3135941.

MADEIRAL, F. et al. Bears: An extensible java bug benchmark for automatic program repair studies. In: **2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2019. p. 468–478.

MARTINEZ, M.; MONPERRUS, M. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: **SSBSE**. [S.l.]: Springer, 2018. (Lecture Notes in Computer Science, v. 11036), p. 65–86.

MARTINEZ, M.; MONPERRUS, M. Astor: Exploring the design space of generate-and-validate program repair beyond genprog. **J. Syst. Softw.**, v. 151, p. 65–80, 2019.

NGUYEN, H. D. T. et al. Semfix: Program repair via semantic analysis. In: **2013 35th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2013. p. 772–781.

POLI, R.; LANGDON, W. B.; MCPHEE, N. F. **A field guide to genetic programming**. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza). Disponível em: http://www.gp-field-guide.org.uk.

QI, Z. et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: **Proceedings of the 24th International Symposium on Software Testing and Analysis**. [S.l.]: ACM, 2015. p. 24–36. ISBN 978-1-4503-3620-8.

SAHA, R. et al. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In: **2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2018. p. 10–13.

SAUVIGNY, F. **Partial differential equations**. [S.l.]: Springer-Verlag London, Ltd., London, 2006. 451 p.

SOBREIRA, V. et al. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In: **Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering**. [S.l.]: IEEE Computer Society, 2018. p. 130–140. ISBN 978-1-5386-4969-5.

TOMASSI, D. A. et al. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In: **2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2019. p. 339–349.

WHITE, M. et al. Sorting and transforming program repair ingredients via deep learning code similarities. **CoRR**, abs/1707.04742, 2017. Disponível em: http://arxiv.org/abs/1707.04742.

XUAN, J. et al. Nopol: Automatic repair of conditional statement bugs in java programs. **IEEE Transactions on Software Engineering**, v. 43, n. 1, p. 34–55, 2017.

YUAN, Y.; BANZHAF, W. Arja: Automated repair of java programs via multi-objective genetic programming. **IEEE Transactions on Software Engineering**, v. 46, n. 10, p. 1040–1067, 2020.

ZELLER, A. Yesterday, my program worked. today, it does not. why? In: NIERSTRASZ, O.; LEMOINE, M. (Ed.). **Software Engineering — ESEC/FSE '99**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. p. 253–267. ISBN 978-3-540-48166-9.