Ternarius: um operador de mutação para o reparo de software baseado em busca com representação *subpatch*

Vinícius P. L. De Oliveira[1], Eduardo F.D. Souza[1], Altino Dantas[1], Lucas Roque[1], Celso G. Camilo-Junior[1], Jerffeson T. Souza [2]

[1]Universidade Federal de Goiás (UFG)
Instituto de Informática (INF)
Alameda Palmeiras, Quadra D, Câmpus Samambaia. Goiânia – Goiás – Brasil

[2]Universidade Estadual do Ceará
Mestrado Acadêmico em Ciência da Computação (MACC)
Avenida Dr Silas Munguba, 1700, Fortaleza – Ceará – Brasil

Grupo Intelligence for Software (i4Soft)
http://ic1.inf.ufg.br/i4soft
{viniciusdeoliveira,eduardosouza,celso}@ufg.br
{altinoneto,lucasroque}@ufg.br
prof.jerff@gmail.com

Abstract. *Software maintenance is a costly and complex task that has been addressed by various approaches which try to automatically repair buggy programs. In this context, GenProg is a search-based tool that has promising results, however, its original patch representation and genetic operators are still in evolution. Thus, this work proposes a novel mutation operator, based on sub patch representation, capable of performing three types of perturbation in the solutions. The proposal was evaluated with the IntroClass benchmark. The results indicate the approach viability by achieving around 30% more fix than the original mutation and getting 94.69% of qualified fixes.*

Resumo. *A manutenção de software é uma tarefa complexa e custosa que tem sido abordada por várias pesquisas que tentam reparar automaticamente programas defeituosos. Nesse contexto, a GenProg é uma ferramenta baseada em busca que tem resultados promissores, no entanto, sua representação original de patch e seus operadores genéticos continuam sendo evoluídos. Assim, este trabalho propõe um novo operador de mutação, baseado em representação de subpatch, capaz de realizar três tipos de perturbação nas soluções. A proposta foi avaliada com o benchmark IntroClass. Os resultados indicaram a viabilidade da abordagem, que conseguiu cerca de 30% mais de reparos do que a mutação original e obtendo 94,69% de correções com qualidade.*

1. Introdução

A humanidade está cada vez mais habituada à utilização de softwares, em tarefas simples ou complexas. Todavia, nem sempre as aplicações se comportam da forma desejada, e com a crescente utilização das mesmas, as falhas ou *bugs*, representam uma problemática significante. Nesse contexto, a tarefa de manter um software funcionando corretamente é tipicamente complexa e cíclica, envolvendo a busca e recuperação de falhas ou implementação de novas funcionalidades.

A manutenção de software se tornou um processo oneroso. Evidências indicam que a quantidade de *bugs* detectados diariamente é maior que a capacidade dos profissionais em resolvê-los [Le Goues et al. 2012a], atrelado ao fato de que esta fase corresponde a 70% do custo financeiro do ciclo de vida de um software [Pressman 2005].

Neste cenário, o Reparo Automatizado de Software (RAS) surgiu como uma tentativa de diminuir esse desequilíbrio entre a quantidade de *bugs* identificados e a capacidade de solucioná-los, além dos custos relacionados a eles, buscando transformar um comportamento inaceitável de uma execução de um software, em um comportamento aceitável de acordo com uma especificação [Monperrus 2015].

Diversas técnicas de reparo automatizado têm sido propostas. DirectFix [Mechtaev et al. 2015] e Angelix [Mechtaev et al. 2016] são exemplos de metodologias baseadas em semântica, que buscam reparar sistemas através de execução simbólica. Prophet [Long and Rinard 2016] e DeepFix [Gupta et al. 2017], por sua vez, utilizam modelos de aprendizado para determinar a possibilidade de certa alteração reparar o software defeituoso. Finalmente, com técnicas baseadas em busca, podemos citar o SPR [Martinez et al. 2014] e a GenProg [Le Goues et al. 2012b].

A GenProg é uma ferramenta que utiliza programação genética para reparar automaticamente softwares defeituosos. A técnica utiliza *patch*, uma sequência de alterações ou operações aplicáveis ao código original, como representação de soluções, isto é, como variantes do código defeituoso. Apesar de apresentar resultados promissores [Le Goues et al. 2012a], alguns de seus aspectos ainda requerem evolução. Por exemplo, visando melhorar a qualidade da busca da GenProg, [Oliveira et al. 2016] propuseram uma representação por *subpatch* - aumentando a granularidade da representação e, assim, permitindo uma melhor troca de material. A nova representação melhorou significativamente o desempenho da ferramenta, quando aplicados os novos e específicos operadores de cruzamento. Contudo, os autores não exploraram novos operadores de mutação.

Assim, o presente trabalho propõe um novo operador de mutação denominado *"Ternarius"* objetivando explorar os benefícios da alta granularidade da representação *subpatch* e aumentar os tipos de pertubações possíveis nas soluções. Sabendo que a atual operação de mutação da GenProg apenas adiciona novas operações aos indivíduos, a principal hipótese desse estudo é que o aumento dos tipos de mutação (mutação com perturbações mais granulares e retirada de operações do indivíduo) melhore a exploração do espaço de busca e, consequentemente, aumente a eficácia da ferramenta.

Para validar a proposta, os experimentos buscam responder as seguintes perguntas de pesquisa:

RQ 1 *O operador Ternarius aplicado à GenProg consegue produzir mais reparos em comparação ao operador canônico?*

RQ 2 *Qual a qualidade dos reparos produzidos pelo operador de mutação Ternarius?*

Com isso, as principais contribuições são:

- Um novo operador de mutação para o reparo de software baseado em busca com representação *subpatch*;
- Uma avaliação de eficácia da proposta e qualidade dos *patches* gerados.

O restante deste trabalho é organizado da seguinte forma. A seção 2 apresenta os princípios da GenProg; Seção 3 descreve o novo operador de mutação proposto; Seção 4 descreve o *design* dos experimentos, além dos resultados e análises; E finalmente, a seção 5 apresenta as considerações finais.

2. Background

2.1. GenProg

A GenProg é uma ferramenta baseada em busca que utiliza programação genética para encontrar uma sequência de operações, chamado *patch*, que quando aplicadas ao software defeituoso, repara-o de acordo com a especificação definida. A ferramenta utiliza um conjunto de casos de teste como especificação para determinar se o sistema foi reparado ou não [Le Goues et al. 2012b].

Para buscar por um reparo, a GenProg recebe um programa e um conjunto de casos de teste com pelo menos um teste negativo, que indica a presença de *bug*. A fim de facilitar a correção da falha, a ferramenta estima onde possivelmente está o erro, com o auxílio dos casos de teste. Posteriormente, são gerados variantes do programa original que são representadas por um *patch*, contendo uma sequencia de edições que será realizada visando a correção do *bug*.

Cada edição é composta de três subespaços, *operation*, *fault* e *fix*, representadas da seguinte forma: *operation(fault,fix)*. *Operation* representa as modificações possíveis dentro de um *patch*, são elas *append* (inserção de um novo *statement*), *swap* (troca de um *statement*) e *delete* (exclusão de um *statement*). *Fault* define o possível ponto de falha, onde a operação será realizada, enquanto *fix* representa o código que será utilizado para modificar o ponto da falha [Le Goues et al. 2012b].

Inicialmente os *patches* são gerados de maneira aleatória, e cada um deles tem seu potencial de correção calculado através dos casos de teste. O objetivo é encontrar um *patch* que ao ser aplicado no código original, gere uma variante capaz de fazer passar todos os casos de teste, chamada de variante plausível. A busca tem continuidade por meio da evolução dos *patches* gerados inicialmente, aplicando sobre eles operadores genéticos de cruzamento e mutação [Le Goues et al. 2012b].

O operador de mutação original da GenProg usa uma probabilidade uniformemente distribuída para realizar uma das três ações: 1) excluir uma operação, 2) inserir operação nova ou 3) substituir uma operação existente. A composição de cada nova operação é realizada a partir de *statements* do programa original.

O operador de cruzamento combina edições dos patches (indivíduos). Por exemplo, define-se um ponto de corte para cada *patch* e, posteriormente, combina-se as partes para gerar novos *patches*. Como cada edição é indivisível na representação *patch*, as

edições possíveis e consideradas na combinação são as geradas aleatoriamente no inicio da busca ou pela mutação [Oliveira et al. 2016].

Além dos operadores, outro aspecto que pode ser melhorado na GenProg é a forma como os *patches* são representados, dada a limitação de combinação dos indivíduos e consequentemente a qualidade da busca.

Visando melhorar a eficiência da ferramenta foi proposta uma nova forma de representação por *patch*, chamada de *subpatch*, para aumentar a granularidade da busca. Nesta, as edições podem ser divisíveis, sendo possível a combinação de subespaços das edições existentes. Essa nova abordagem torna possível que o cruzamento gere operações totalmente novas, o que aumenta as combinações de material genético, consequentemente aumentando a diversidade da população. Ressalta-se que, apesar de aumentar a diversidade, os indivíduos gerados possuem distâncias aceitáveis para o papel de *exploitation* do operador [Oliveira et al. 2016]. A Figura 1 apresenta as duas formas de representação.

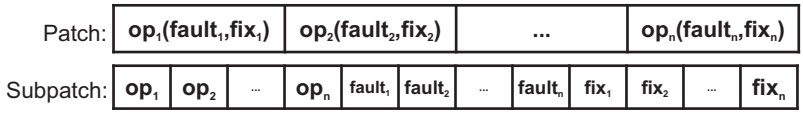

Figura 1. Representação por patch e por subpatch.

2.2. Operador de cruzamento *Unif1Space*

Neste trabalho será utilizado o operador de cruzamento *Unif1Space* com memorização, que apresentou melhor desempenho em trabalhos anteriores quando aplicado à GenProg. Dessa forma, focar-se-á na influência do operador de mutação.

O *Unif1Space* é a aplicação do operador uniforme clássico a um subespaço de uma edição do patch [Oliveira et al. 2016]. Este operador favorece a exploração, pois possui uma capacidade de combinação de material genético superior a um operador de um ponto de corte, por exemplo. Além disso, privilegia operações que necessitam de mudança em apenas um dos subespaços de uma operação do *patch*.

Ainda, o *Unif1Space* pode utilizar um processo de memorização para mitigar as perdas por mapeamento entre representações [Oliveira et al. 2016]. Tal processo consiste em armazenar partes de edições, que foram inutilizadas durante o processo de cruzamento, e reutilizá-las em indivíduos futuros, evitando-se assim perda de material genético. Essa memória genética não é compartilhada entre toda a população, mas sim entre os indivíduos de uma mesma linhagem genética, ou seja, um indivíduo só pode utilizar memória genética proveniente de um antepassado dele. Isso reduz a inserção de valores aleatórios no indivíduo [Oliveira et al. 2016].

3. Operador de mutação *Ternarius*

A granularidade da representação tem grande influência no resultado da operação de mutação. Observa-se que a granularidade da representação original da GenProg impede a alteração de elementos da edição que possuem baixo impacto na busca. A Figura 2 apresenta um exemplo em que se tem um *patch* eleito para a mutação e um *patch* desejado. Observa-se que bastaria a mudança do Fault_1 da primeira operação do *patch* existente para Fault_2 que a operação do *patch* desejado seria obtida. Por isso, uma mutação mais

granulada, no nível de subespaço, poderia, através da mudança em subespaços, reativar operações inativas e aumentar as chances de criação de novas operações.

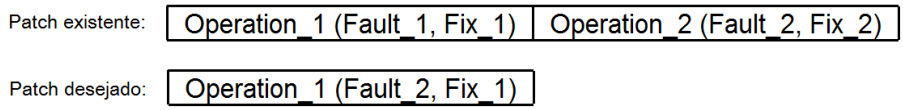

Figura 2. Limitações da representação de *patch* com operações indivisíveis.

O operador de mutação tem grande impacto e relevância na busca da GenProg. Conforme dito anteriormente, este operador é responsável pela inserção de uma edição totalmente nova no *patch*, resultando em mais material para a busca. Apesar da importância do aumento dos indivíduos, estudos demonstram que *patches* gerados por seres humanos são, comumente, curtos [Martinez and Monperrus 2015].

Além disso o crescimento acentuado dos *patches* pode causar problemas para a busca, pois *patches* muito grandes tendem a mudar completamente um código, que eventualmente seria reparado com uma alteração pontual. Este fato também impacta no processo de busca, pois quanto maior o individuo menor é a influência exercida pela mutação. Sendo assim, a capacidade de redução do tamanho do indivíduo pode ser benéfica para a busca. A Figura 2 demonstra também que a deleção da segunda operação do *patch* existente aproximaria ele do *patch* desejado.

A fim de explorar melhor as características da representação de maior granularidade e incluir uma estratégia de controle de crescimento dos indivíduos, foi proposto um novo operador de mutação, intitulado de ”*Ternarius*”. Este operador é capaz de modificar uma variante de até três formas:

- Deletando operações existentes selecionadas aleatoriamente;
- Alterando um valor de um subespaço, selecionado aleatoriamente, de uma operação aleatoriamente escolhida;
- Acrescentando novas operações de edições a um indivíduo.

Cada uma dessas alterações possuem uma probabilidade de serem aplicadas ao indivíduo, conforme a ordem descrita acima. Esta ordem é necessária para evitar que uma modificação anule o efeito de outra que ocorreu anteriormente. A figura 3 apresenta um exemplo do novo operador de mutação, demonstrando as três etapas.

A Figura 3 apresenta um exemplo onde o novo operador de mutação é aplicado em um *patch* com três operações. Inicialmente, o *patch* é submetido à etapa de deleção em que é escolhido uma operação aleatória do *patch* e é feita a deleção desta operação, neste caso foi escolhido a terceira operação “r(5,6)”. O resultado da deleção é submetido à etapa de mudança de valores de subespaço, em que uma operação é escolhida aleatoriamente e um subespaço desta operação é selecionado aleatoriamente, nesta caso o subespaço *operator* foi escolhido, é então sorteado um valor dentro do subespaço escolhido (*operator*), sendo assim o valor selecionado “d” (*delete*) é substituído por “a” (*append*). Finalmente, é feito a mutação gerando assim uma nova operação de edição que será incluída no fim do *patch*, neste caso a operação resultante da mutação foi “s(1,3)”.

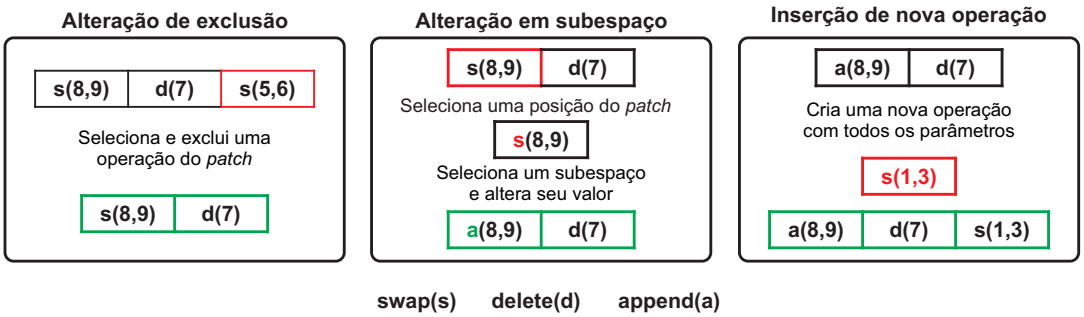

Figura 3. Exemplo da aplicação do operador *Ternarius*. Destaques em vermelho indicam itens selecionados ou criados, e em verde indicam os resultados em cada fase.

4. Avaliação da proposta

Para responder as questões de pesquisa, o operador de mutação *Ternarius* foi implementado na ferramenta GenProg. Assim, ambos os operadores de mutação foram avaliados a partir do *Benchmark* denominado *IntroClass* [Goues et al. 2015]. Este *dataset* publicado online e que vem sendo utilizado em pesquisas relacionadas a reparo automatizado de software, é composto por versões de programas construídos por estudantes de programação. Nele podem ser encontrados programas corretos, versões defeituosas e casos de teste capazes de relevar os defeitos para cada programa. A Tabela 1 apresenta informações do *IntroClass* que foram utilizadas neste experimento. A coluna "Reparo" indica a quantidade de casos de teste que foram utilizados durante o processo de geração de soluções para cada programa, enquanto a coluna "Validação" apresenta a quantidade de casos de teste que foram utilizados para avaliar a qualidade dos reparos produzidos.

Tabela 1. Quantidade de programas, versões e casos de teste do *Benckmark* utilizado (*IntroClass*) para avaliação do operador de mutação proposto.

		Casos de teste	
Programa	**Versões**	**Reparo**	**Validação**
checksum	8	6	10
digits	18	6	10
median	14	7	6
smallest	14	7	8
syllables	14	6	10

Tendo os dados provenientes do *benckmark*, a ferramenta GenProg foi executada 30 vezes para cada uma das versões dos programas. Os parâmetros utilizados foram: 30 gerações, população de 30 indivíduos, elitismo de três indivíduos, taxa de cruzamento de 50%, taxa de mutação de 100% e seleção por torneio binário [Goues et al. 2015]. A alta taxa de mutação comumente utilizada é justificada pelas características e papel já relatadas do operador de mutação na GenProg. O critério de parada foi alcançar 30 gerações ou encontrar um reparo para o programa. Tanto no operador de mutação canônico, quanto na implementação com o novo operador de mutação, o tipo de cruzamento utilizado foi o Unif1Space com memorização, proposto em [Oliveira et al. 2016].

Conforme indicado na seção 3, o operador de mutação *Ternarius*, diferente do

canônico, pode aplicar três ações diferentes durante a perturbação. Por isso, foram avaliadas 5 variações deste operador, sendo que cada variação corresponde a combinações das probabilidades de ocorrência de cada operação. A convenção "delete_insert_update" indica as taxas de: a) deleção de uma operação do *patch*, b) inserção de uma nova operação no *patch* e c) manipulação de material genético dentro de um subespaço, respectivamente. É importante pontuar que não foram apresentadas variações da probabilidade de ocorrer operação de inserção na mutação porque, em testes preliminares, foi observado que alterações neste parâmetro sempre produzia resultados substancialmente inferiores.

4.1. Resultados e Análises

Inicialmente serão apresentados os dados relativos à eficácia de cada uma das estratégias de mutação. Para esta análise, foram condensados todos os resultados de todos os programas por cada uma das variações de operador de mutação. A Tabela 2 apresenta o percentual médio de reparos obtidos em relação ao total de defeitos existentes em todos os programas, considerando as 30 execuções para cada versão e operador de mutação.

Tabela 2. Percentual médio de reparos encontrados entre todos os programas, para cada variação da mutação proposta e da mutação canônica, considerando 30 execuções. ▲ indica superioridade e ▼ inferioridade em relação ao resultado da mutação canônica com o respectivo cruzamento.

	ID	delete_insert_update (%)	Reparos
Mutação canônica	MC	0_100_0	27,962%
Ternarius	T1	0_100_100	27,654% ▼
	T2	0_100_10	**36,358%** ▲
	T3	10_100_0	29,382% ▲
	T4	80_100_0	34,876% ▲
	T5	80_100_10	26,600% ▼

Como pode ser visto, na segunda linha constata-se que 27,9% de todos os defeitos foram reparados quando se utilizou o operador de mutação canônico. Da linha 3 à linha 7, são apresentados os resultados das cinco diferentes configurações do operador *Ternarius*. É possível observar que as variações T1 e T5 obtiveram resultado pior do que a mutação canônica, embora a diferença não chegue a 0,5%. No caso do T1, o fato de haver 100% de chance de mutação em um dos subespaços pode estar causando mais perturbações do que o razoável para manter material genético de qualidade, impedido assim um melhor desempenho para esta configuração. Já a T5 apresenta uma probabilidade de 10% de realizar alterações em subespaços, porém, com 80% de chance de deletar operações do *patch*. O desempenho desta combinação pode não ter sido melhor pela diminuição de material genético imposto pelo um alto percentual de deleção em combinação com as perturbações nos subespaços, pois dessa forma a mutação em tais subespaços passa a ter um grande impacto na composição dos indivíduos. Nesse sentido, o resultado de 34,8% obtido pela T4, o segundo melhor, deve ser destacado uma vez que também possui alta taxa de deleção de operações. Entretanto, neste caso não há chance de mudança nos subespaços de modo que, tal resultado é obtido graças ao operador de cruzamento utilizado, o qual permite a troca de material entre subespaços.

Finalmente, a configuração T2 foi a que obteve o melhor resultado, chegando a reparar em média 36,3% do defeitos no *benchmark*. Como se vê, esta configuração

(0_100_10) não realiza deleção de operações, sempre insere novas operações e realiza poucas perturbações em subespaços; Dessa forma há sempre um boa quantidade de material genético disponível para a combinação e a atuação pontual nos subespaços promove a melhoria de eventuais materiais de baixa qualidade.

Para uma observação mais detalhada da configuração *Ternarius* que obteve os melhores resultados considerando todos os programas, a Tabela 3 apresenta os percentuais médios de reparos alcançados pela configuração T2 em cada um dos cindo programas.

Tabela 3. Percentual médio de reparos encontrados para cada um dos programas considerando 30 execuções da mutação canônica e da proposta T2. ▲, ▼ e – indicam respectivamente que a proposta foi superior, inferior ou igual em relação à mutação canônica com o mesmo cruzamento aplicadas ao mesmo programa.

Programas	Mutação canônica	T2
checksum	12,92%	17,92% ▲
digits	5,56%	5,56% –
median	32,38%	46,19% ▲
smallest	49,76%	70,95% ▲
syllables	5,71%	5,71% –

Como pode ser visto, em três dos cinco programas, a configuração T2 conseguiu produzir, em média, mais reparos do que a mutação canônica da ferramenta GenProg. Observa-se que no caso do programa "checksum" saiu-se de 12,92% para 17,92%, um ganho real de 38,7% ao se utilizar o *Ternarius* configurado como 0_100_10. Para os casos dos programas "median" e "samllest" o ganho foi ainda maior, chegando a 42,6% em ambos. Entretanto, nota-se que para os programas "digits" e "syllables", mesmo sendo a melhor configuração, a mutação T2 não foi suficiente para melhorar os resultados obtidos pela a mutação original. Não obstante, deve ser observado que para tais programas o percentual de reparos encontrados é consideravelmente pequeno (5,56% e 5,71%) indicando que os defeitos neles contidos podem apresentar alguma característica particularmente complexa para a GenProg como um todo, não sendo contornável apenas com mudança no mecanismo de mutação. Como o *benchmark IntroClasse* não fornece detalhes sobre os tipo de defeito em cada programa, seria necessária uma inspeção manual em cada uma das versões para se obter eventuais *insights* sobre como contornar esse problema, porém, esta tarefa foi deixado para evolução do presente estudo.

Diante do que fora exposto, pode-se concluir que, no cenário estudado, o operador de mutação *Ternarius* conseguiu apresentar, em média, mais reparos do que o operador de mutação atualmente presente na ferramenta GenProg considerando todos programas juntos, e, na pior das hipóteses, obteve os mesmos resultados quando observado cada um dos programas individualmente. Respondendo-se assim a **RQ1**.

Visando analisar a qualidade dos reparos produzidos com cada um dos operados de mutação, todos os reparos encontrados foram validados utilizando-se os casos de teste de validação disponíveis no *benchmark*. Na Tabela 4, verifica-se a acurácia dos operadores de mutação através da medida da relação entre a quantidade de reparos válidos e o total de reparos obtidos pela ferramenta. A tabela reporta ainda os quantitativos de casos de teste de validação que foram executados, quantos passaram (positivos) e quantos falharam

(negativos). A linha MC é relativa ao operado canônico e as linhas abaixo desta trazem os dados para as variações do *Subsapce Mutation*.

Tabela 4. Dados de validação dos reparos obtidos pelo operador de mutação canônico e pelas variações do Ternarius.

	Quantidade de casos de teste			Quantidade de Reparos			
Mutação	Executados	Positivos	Negativos	Encontrados	Falsos	Válidos	Acurácia
MC	3534	3498	36	453	13	440	**97,13%**
T1	3508	3420	88	448	29	419	93,53%
T2	4518	4404	114	589	45	**544**	92,36%
T3	3686	3587	99	476	35	**441**	92,65%
T4	4358	4280	78	565	30	**535**	**94,69%**
T5	3372	3296	76	430	24	406	94,42%

Observando os valores de acurácia, constata-se que o operador canônico obteve o valor mais elevado, 97,13%, uma vez que dos 453 reparos encontrados 440 foram validados. Isso significa que é alta a chance de um reparo produzido por tal operador ser de fato um reparo válido e não um falso positivo. Nesse aspecto, a variação do *Ternarius* que mais se aproximou do MC foi a T4 com 535 reparos válidos de 565 reportados, performando, assim, 94,64% de acurácia. A configuração T2, que encontrou a maior quantidade de reparos (589), apresentou 92,36% sendo o pior valor nesta métrica.

As observações anteriores indicam um *trade-off* entre a eficácia em encontrar reparos e a garantia do quão válidos são os mesmos. Por esse raciocínio, ainda que se escolham as configurações T2, T3 e T4 ter-se-ão mais reparos válidos do que a quantidade de reparos válidos provenientes da mutação canônica. Portanto, em resposta à **RQ2** pode-se afirmar que dentre as melhores configurações do operador *Ternarius*, 94,69% é o indicativo de qualidade dos reparos obtidos pela configuração T4.

5. Considerações Finais

A manutenção de sistemas se tornou um processo ainda mais dispendioso com a popularização dos softwares. Por esse motivo, várias propostas de Reparo Automatizado de Software têm sido desenvolvidas, porém ainda há espaço para melhorias.

Este trabalho propôs um novo operador de mutação para a GenProg, uma consolidada ferramenta de reparo automatizado. O operador nomeado *Ternarius*, além de inserir novas operações aos indivíduos (*patches*), como é feito na GenProg tradicional, pode também excluí-las ou realizar perturbações nos subespaços das operações.

Os experimentos indicaram que o princípio do operador proposto é promissor ao aumentar o percentual médio de reparos de 27,9% para 36% (cerca de 30% de aumento) considerando todos os defeitos do *benchmark* utilizado. Além disso, constatou-se um *trade-off* entre encontrar soluções para os defeitos e a taxa de validade destas.

Como evolução desta pesquisa, pretende-se verificar se eventuais características do *dataset* utilizado não privilegiam alguma variação do operador proposto, ampliar o conjunto de programas avaliados para outros *benchmarks* e verificar o comportamento do operador de mutação *Ternarius* em combinação com outros operadores de cruzamento e fora do contexto da GenProg.

Referências

Goues, C. L., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., and Weimer, W. (2015). The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351.

Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012a). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE.

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012b). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Long, F. and Rinard, M. (2016). Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, volume 51, pages 298–312. ACM.

Martinez, M. and Monperrus, M. (2015). Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205.

Martinez, M., Weimer, W., and Monperrus, M. (2014). Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 492–495. ACM.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2015). Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 448–458. IEEE Press.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 691–701. IEEE.

Monperrus, M. (2015). Automatic Software Repair: a Bibliography. Technical Report hal-01206501, University of Lille.

Oliveira, V. P. L., Souza, E. F. D., Le Goues, C., and Camilo-Junior, C. G. (2016). Improved crossover operators for genetic programming for program repair. In Sarro, F. and Deb, K., editors, *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 112–127, Cham. Springer International Publishing.

Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan.