Avaliação do Feedback Gerado Por Um Corretor Automático de Algoritmos

André L. A. Raabe[1,2], Elieser A. de Jesus[1], Andrei Hodecker[3], Fillipi Pelz[1]

[1]Mestrado em Computação Aplicada
[2]Programa de Pós Graduação em Educação
[3]Bacharelado em Ciência da Computação
Universidade do Vale do Itajaí - UNIVALI
{raabe, elieser, andreihod, fillipi}@univali.br

Resumo: Este artigo apresenta a avaliação sobre como o feedback fornecido por um corretor automático de algoritmos impactou nas atitudes subsequentes dos estudantes. O artigo detalha o corretor construído que combina a análise estática e dinâmica para fornecer auxilio aos estudantes encontrarem seus erros. A avaliação realizada com 23 estudantes de uma disciplina introdutória de programação indicou que o corretor teve um impacto positivo na dinâmica de resolução e depuração das soluções pelos estudantes, mas apontou a necessidade de ampliação no número de testes estáticos a serem realizados.

Abstract. This paper presents an impact evaluation of the feedback provided by an automatic program correction system in student's attitudes. The article details the system architecture that combines static and dynamic analysis to provide assistance to students to find their mistakes. The assessment carried out with 23 students of an introductory programming course indicated that system had a positive impact on the dynamics of resolution and debugging solutions by students, but pointed out the need to expand the number of static tests to be performed.

1. Introdução

A correção automática de programas escritos pelos estudantes é foco de frequentes pesquisas. Dentre as crenças que motivam a realização destes trabalhos está a possibidade de atender melhor ao aprendiz de programação fornecendo *feedback* automático.

Conforme Ala-Mutka (2005), as técnicas mais utilizadas para correção automática de programas são a análise dinâmica e análise estática. Na análise dinâmica o algoritmo é executado com entradas pré-definidas e suas saídas são comparadas com as saídas esperadas. Na análise estática o algoritmo não é executado. É realizada uma análise de sua estrutura possibilitando avaliar o estilo de codificação, a presença ou ausência de itens léxicos (tokens), a detecção de plágio e etc.

Segundo Pelz e Raabe (2013), muitos sistemas de correção automática de programas limitam-se a fornecer um parecer sobre o programa indicando se ele completa com sucesso um determinado número de casos de teste, tal como fazem os ambientes de competição e juízes on-line. Do ponto de vista da técnica utilizada, estes corretores realizam apenas a análise dinâmica. A utilização da análise estática é relatada em poucos trabalhos, e nestes, geralmente voltada à identificação de métricas de

DOI: 10.5753/cbie.sbie.2015.358

qualidade do código e estilo de programação, não atuando diretamente para auxiliar o estudante a detectar seus erros.

Este trabalho apresenta uma abordagem para construção de um gerador de dicas sobre os erros cometidos pelos estudantes, usando como base a combinação das análises estática e dinâmica para correção dos programas. Desta forma, além de usar casos de teste da análise dinâmica incluímos a detecção de padrões de erros comuns realizados por estudantes realizando para isso uma analise estrutural do programa (Análise Estática).

Os erros comuns foram inferidos semi-automaticamente a partir de uma base com 1429 problemas resolvidos anteriormente pelos estudantes. Foram desenvolvidos algoritmos denominados de *Tree Walkers* que detectam a presença destes erros ao percorrer a estrutura do programa do estudante. O gerador de dicas foi integrado a ferramenta Portugol Studio (NOSCHANG et al, 2014) e foi realizado um experimento com uma turma de 23 estudantes para avaliação do impacto das dicas fornecidas nas ações subsequentes durante a resolução de problemas.

Na primeira parte deste artigo discutimos as técnicas que foram utilizadas para gerar as dicas. Na sequencia discutimos o projeto do gerador de dicas e como este foi implementado. Por fim, apresentamos os detalhes do experimento que foi realizado com os estudantes, os resultados obtidos e as discussões finais.

2. Trabalhos Similares

Três revisões da literatura sobre correção automática de algoritmos foram publicadas desde 2005. Douce (2006) faz um levantamento histórico das ferramentas de correção automática de algoritmos desde 1960 até o presente. Ala-Mutka (2005) direcionou sua revisão para apresentar os métodos e técnicas utilizados pelos sistemas de correção. Por fim Ihantola *et al*. (2010) atualizaram a revisão de Ala-Mutka. Pelz e Raabe (2013) complementaram a revisão de Ilhantola com foco em analisar o tipo de feedback fornecido pelos corretores.

Marceau *et al.* (2011) avaliaram a efetividade das mensagens de erros exibidas por uma ferramenta denominada DrScheme para entender como os estudantes reagiam às mensagens e determinar porque algumas eram mais efetivas do que outras. Os autores realizaram uma análise de diversas edições do código em resposta as mensagens de erros. Entrevistaram os estudantes sobre as interpretações destas mensagens e os questionaram sobre o vocabulário utilizado nas mensagens. A metodologia utilizada para a avaliação da eficiência das mensagens de erro partiu da seguinte premissa: uma mensagem de erro é considerada eficiente quando um estudante lê o texto da mensagem, consegue entender o seu significado e realiza alguma ação útil para a resolução de um problema. O objetivo dos autores foi investigar quanto os estudantes conseguiram progredir depois de receberem uma mensagem de erro. Foram assinaladas rubricas para cada execução de código após a exibição de uma mensagem de erro. As rubricas utilizadas são ilustradas no quadro 1.

Quadro 1. Rubricas para avaliação das ações posteriores a uma mensagem de erro.

Rubrica	**Descrição**
[DEL]	O estudante apagou o código problemático.
[UNR]	O estudante faz algo que não tem relação ao erro e também não ajuda.
[DIFF]	O estudante faz algo que não tem relação ao erro, mas corrige outro erro.

[PART]	O estudante entendeu o erro e está no caminho de encontrar a solução.
[FIX]	O estudante corrigiu o erro.

Fonte: Marceau *et al.* (2011)

Em nosso trabalho utilizamos uma abordagem similar a de Marceau (2011) para avaliar a efetividade das dicas que foram geradas no experimento realizado com os estudantes.

3. Gerador de Dicas

A geração de dicas é baseada na detecção de erros nos algoritmos dos estudantes, que por sua vez exige um mecanismo de correção automática. O mecanismo utilizado neste trabalho utiliza tanto a análise estática quanto a dinâmica para corrigir o algoritmo do estudante, encontrar possíveis erros, e caso eles existam gerar dicas que possam ser úteis para a aprendizagem.

O gerador de dicas apresentado neste artigo é fruto de uma pesquisa conduzida em duas etapas. A primeira etapa definiu e validou um framework para correção automática e geração de dicas que combina a análise estática e dinâmica. A segunda etapa construiu algoritmos de análise estática, denominados *treewalkers,* para geração de dicas e avaliou o impacto destas. A seguir detalhamos estas duas etapas.

3.1. Framework do Corretor Automático

A correção ocorre sempre após a submissão da resposta de um problema pelo estudante. Após a submissão o estudante recebe como *feedback* duas listas. A primeira delas indica os casos de testes que foram realizados com sucesso e os que não obtiveram sucesso, resultado da análise dinâmica. A segunda lista apresenta as dicas que resultam da execução dos *treewalkers* ligados à análise estática. A seguir são descritos os principais componentes do *framework* ilustrado na Figura 1.

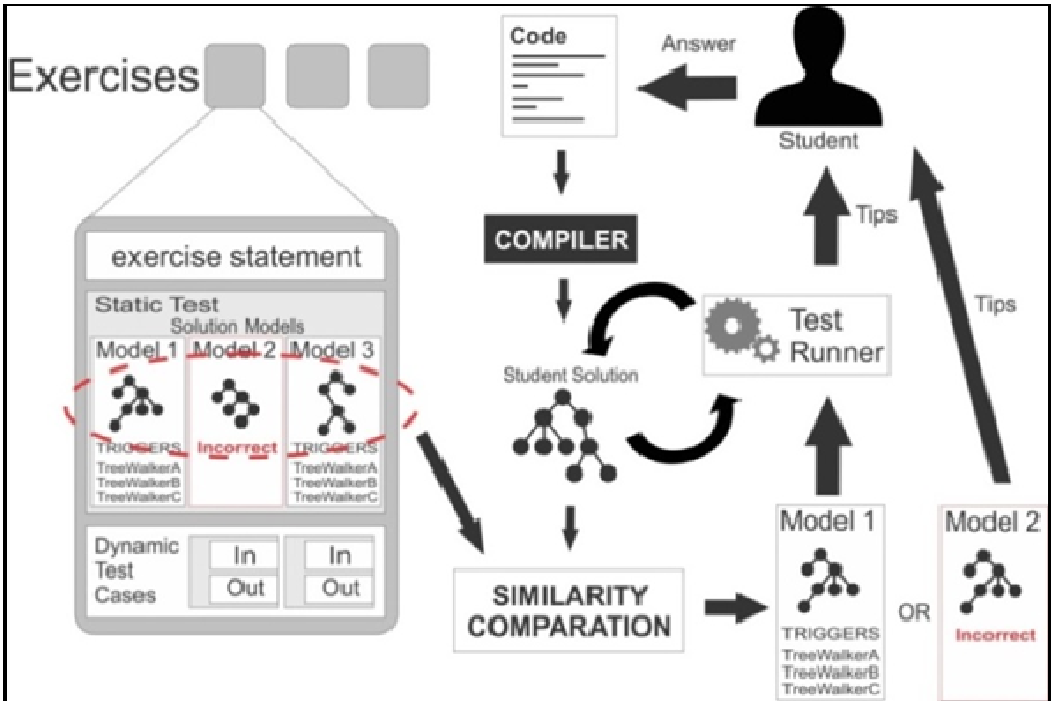

Figura 1- Framework de correção automática

Para que um exercício possa ser corrigido ele deve possuir os seguintes metadados:

(i) Enunciado (*exercise statement*): contém a descrição textual do algoritmo a ser resolvido pelo estudante;

(ii) Soluções modelo (*solution models*): são árvores sintáticas abstratas serializadas contendo possíveis soluções para o algoritmo, ou ainda soluções reconhecidamente erradas. Cada solução modelo possui um conjunto de *treewalkers* associados a ela. Os *treewalkers* por sua vez possuem parâmetros e a mensagem que eles deverão disparar como feedback ao estudante.

(iii) Casos de Teste dinâmicos (*dynamics test cases*): Contém uma série de casos de teste com valores de entrada e de saída que possibilitam simular a execução do algoritmo e avaliar sua corretude. Foi definida uma gramática para especificação dos casos de teste para facilitar a definição de testes com muitos valores (ex. problemas com vetores contendo 100 elementos).

O processo inicia com o estudante submetendo um código para o corretor. O compilador analisa o código e gera uma árvore sintática abstrata da solução do estudante. Em seguida ocorre a verificação da similaridade entre a solução do estudante e uma das soluções modelo para o problema. Esta verificação é feita com as árvores serializadas e utiliza o algoritmo da distância de Levenhstein (LEVENSHTEIN, 1966). A solução modelo mais similar a solução do estudante é selecionada e com ela a lista de *treewalkers* associados que serão executados.

O executor do teste (*test runner*) tem a incumbência de executar a solução do aluno utilizando os casos de testes dinâmicos e disparar os *treewalkers* para percorrer a estrutura da solução do estudante a fim de encontrar padrões de erro pré-definidos. Como resultado ele gera as duas listas contendo os casos de teste e as dicas relacionadas à análise estática.

No caso da solução modelo ser uma solução errada não haverá *treewalkers* associados apenas uma dica é retornada diretamente ao estudante. Esta estratégia busca tratar de forma mais direta problemas em que muitos estudantes se equivocam. Por exemplo, o algoritmo que solicita a troca do valor entre duas variáveis a e b e recebe como resposta a=b; b=a.

Cada exercício é armazenado em um arquivo XML (*Extensible Markup Language*). Foram construídos 60 exercícios com base na lista definida em Pelz (2011), para isso foi construído um editor que auxilia na definição dos casos de teste dinâmicos, na inserção das soluções modelo, na associação dos treewalkers as soluções modelo e definição do texto das dicas.

3.2. Definição e Construção dos *Treewalkers*

Os *tree walkers* foram inferidos a partir de uma lista de erros frequentemente cometidos por estudantes de programação. Para esta inferência utilizou-se uma base de respostas para os 60 exercícios mencionados na seção anterior, totalizando 1429 respostas/programas que foram realizados ao longo de seis semestres por aproximadamente 250 estudantes. Cada resposta possuía uma nota associada (dada pelo monitor da disciplina), os tópicos envolvidos, e os comentários descritos pelo monitor indicando os erros.

Consultas a esta base de dados permitiram definir uma lista de 12 questões que mais continham erros (não tinham nota 10) e que cobriam diferentes tópicos como loops, desvios, e vetores. Os erros mais comuns nestas 12 questões foram classificados manualmente formando padrões de erros comuns. Para cada um foram criados *tree*

walkers capazes de detectar estes erros nos programas. Para tal os tree walker necessitam ter parâmetros que se adaptam a cada tipo de problema a que estão associados. Uma vez que um erro comum é encontrado é possível gerar uma dica para ajudar o estudante a encontrar a solução correta para o problema detectado.

O quadro 2 descreve todos os *tree walkers* criados e suas respectivas funções.

Quadro 2. *Tree walkers* e suas descrições

Nome do *Tree Walker*	**Descrição**
MandatoryInstructions	Procura a existência de uma ou mais estruturas dentro do código do estudante.
ProhibitedInstructions	Informa quando existem instruções proibidas no código do estudante.
ReadAfterOperation	Informa quando o estudante faz uma entrada de dados após ter manipulado as variáveis do programa.
ReadWriteOrder	Detecta se a ordem das variáveis passadas por parâmetro no comando de entrada de dados está em uma ordem diferente durante a saída de dados.
UsingAux	Verifica se o uso de variável auxiliar para a troca de valores entre variáveis foi utilizada corretamente.
VarIsNotUsed	Detecta se alguma variável que foi declarada e não utilizada durante o programa.
PossibleInfiniteLoop	Identifica possíveis laços infinitos.
NumberOfConditions	Verifica se existe uma quantidade errada de desvios no código do estudante
ConstantIndex	Detecta se o estudante não incrementou o índice do vetor ou matriz.
EmptyBlocks	Detecta blocos de código vazios.
NumberOfInstructions	Detecta se o estudante está utilizando uma instrução mais vezes do que o necessário.
MandatoryArray	Detecta se existe vetor ou matriz na solução do estudante.

4. Avaliação do Impacto das Dicas

O corretor e as questões selecionadas foram integradas a IDE Portugol Studio (Noschang, et al. 2014) usando o suporte a *plug-ins* existente na ferramenta. Foi realizado um experimento com 23 estudantes da primeira fase em um curso de Ciência da Computação. O experimento durou 90 minutos e os estudantes resolveram os 12 exercícios de algoritmos para o qual foram criados *treewalkers* (mencionados na seção anterior).

O objetivo deste experimento foi avaliar a influência das dicas emitidas pelo corretor nas ações tomadas pelos estudantes para corrigir o erro diagnosticado, bem como avaliar se o corretor automático ajuda o estudante a resolver os exercícios propostos.

No experimento foi considerada como uma “dica efetiva” aquela que forneceu informações úteis para o estudante corrigir o problema e progredir na resolução do

exercício. A utilidade da dica do ponto de vista do estudante foi mensurada através da observação da ação tomada pelo estudante logo após a exibição da dica, classificando-a como relevante ou não para a resolução do problema.

Para capturar a ação realizada pelo estudante logo após a exibição de uma dica foi desenvolvido um plug-in para o Portugol Studio. Este plug-in foi projetado para gravar um registro toda vez que o estudante solicitava uma correção de um exercício, seja através da interface gráfica do corretor (o plug-in) ou da tela onde foram exibidas as dicas do corretor estático e os casos de teste do corretor dinâmico. Desta maneira foi possível registrar não só as solicitações de correção automática, mas também se o estudante navegou entre os casos de testes ou selecionou uma dica emitida pelo corretor estático.

Para inferir se a mensagem de um *tree walker* foi eficiente utilizou-se uma abordagem similar ao trabalho de Marceau *et al.* (2011). Para cada *tree walker* e código-fonte analisado foi anotado uma rubrica (ver quadro 3) representando a alteração que o estudante realizou no código para resolver o problema.

Quadrdo 3. Rubricas anotadas em cada alteração de código-fonte

Rubrica	**Explicação**
[FIX]	Mnemônico da palavra *fixed* (corrigido). Usado quando o estudante corrigiu um erro apontado pela dica.
[ATT]	Mnemônico da palavra *attention* (atenção), usado quando se constatou que o estudante estava tentando resolver o problema, mas sua solução ainda não era correta. Esta constatação foi feita com a detecção de cliques em uma das dicas geradas e através da monitoração de alguma alteração no código-fonte para tentar solucionar o problema.
[UNR]	Mnemônico da palavra *unrelated* (sem relação), usado quando a alteração feira no código não tinha relação com a dica gerada.

O processo de anotar as rubricas para cada correção se deu em 3 etapas: primeiramente se anotava como "FIX" às dicas que existiam em uma correção mas não na correção seguinte do estudante. As dicas que persistiam ou não tinham continuidade eram anotadas com "UNR".

Em uma segunda etapa foram analisadas as rubricas do tipo "UNR". Nos momentos em que o estudante deu foco na dica gerada pelo corretor (posicionando o mouse na área da tela onde a dica era apresentada) assumiu-se que o estudante estava prestando atenção, e então assinalou-se a rubrica "ATT". Por fim, foram analisadas todas as correções com rubricas "UNR", agora em menor quantidade, e o código-fonte do estudante na correção seguinte. Se o estudante realizou alguma alteração relacionada com a dica apresentada, então se assinalou a rubrica "ATT".

A partir das rubricas anotadas foi calculada uma *taxa de eficiência* para cada *tree walker* através da equação a seguir, onde a variável *t* representa o *tree walker* analisado.

$$e_t = \frac{FIX}{UNR + ATT + FIX} * 100$$

A eficiência é portanto uma razão entre a frequencia de ações FIX e o total de ações ($UNR + ATT + FIX$).

5. Resultados

A seguir, são apresentados os resultados obtidos no experimento sob dois pontos de vista, um observando os estudantes separadamente e trançando uma média dentre os exercícios respondidos, e outro observando cada exercício fazendo a média entre os estudantes que o responderam.

Na tabela 1 são apresentados os resultados obtidos para cada um dos doze exercícios. As colunas da tabela representam respectivamente: estudantes que iniciaram o exercício, nota média obtida pelos casos de teste (a razão entre o número de casos certos e o total de casos), média de tempo que cada estudante se dedicou ao problema e número médio de correções solicitadas para o problema.

Tabela 1. Dados relativos aos exercícios respondidos.

Exercício	Estudantes que Iniciaram	Nota Média dos casos de teste	Média de Tempo no Exercício (em minutos)	Número médio de Correções Solicitadas
Exercício 1	19	4,5263	9,6316	2,6316
Exercício 2	16	9,75	7,3750	2,3750
Exercício 3	7	5,7142	1,7143	2,2857
Exercício 4	5	6,2	5,8	3,20
Exercício 5	2	5	0	1,5
Exercício 6	2	0	2,5	3,5
Exercício 7	0	0	0	0
Exercício 8	1	0	28	3
Exercício 9	0	0	0	0
Exercício 10	2	0	2,5	2,5
Exercício 11	1	0	15	3
Exercício 12	3	0	0	1
Médias	4,83	2,59	6,04	2,08

Podemos perceber que os 5 primeiros exercícios foram os mais respondidos pelos estudantes e também foram os únicos com casos de testes corretos. Observa-se também que a média de 2,08 correções indica que os estudantes não fizeram uso abusivo do corretor, ou seja, a facilidade na correção não levou os estudantes a ingressarem numa abordagem de tentativa e erro sem a devida reflexão.

No total foram detectadas 83 exibições de dicas geradas pelo corretor dinâmico. Em 43 (51,8%) destas situações detectou-se a atenção do estudante nos casos de teste. Estes resultados podem indicar que estas dicas e a forma como os casos de testes estão sendo apresentados no software precisam ser aprimorados de modo a deixar mais claro para o estudante que problemas foram encontrados durante a execução dos testes.

A tabela 2 mostra os *tree walkers* que foram disparados durante o experimento (nem todos chegaram a ser usados), os exercícios que os dispararam, bem como a quantidade de correções em que estes foram utilizados. A primeira linha representa o caso de uma solução modelo incorreta.

Tabela 2. Dados coletados dos tree walkers.

Tree Walker	Exercícios	Correções	FIX	ATT	UNR	Eficiência
Mensagens de Soluções Incorretas	Exercício 4	12	0	6	6	0%
EmptyBlocks	Exercícios 1 e 2	12	8	0	4	66%
ReadAfterOperation	Exercícios 1 e 2	2	0	0	2	0%
NumberOfConditions	Exercícios 1, 2, 8 e 12	26	6	1	19	23%
MandatoryArray	Exercício 8 e 12	3	1	0	2	33%

Algumas dicas não produziram efeito algum sobre a atitude dos estudantes, outras tiveram um taxa de eficiência considerada insatisfatória, e no caso específico do *tree walker* "EmptyBlocks" uma taxa de eficiência considerada satisfatória.

O *tree walker* "MandatoryArray" é um dos mais simples entre os implementados, seu objetivo é descobrir se o estudante está utilizando um vetor ou matriz em um problema que exige que isso seja feito. Este *tree walker* e o "EmptyBlocks" tiveram as melhores taxas de eficiência, o que indica que os erros mais cometidos podem ser simples de serem detectados.

Já o *tree walker* "ReadAfterOperation" praticamente não foi executado, indicando que o padrão de erro que o mesmo encontra não é um erro muito comum entre os estudantes. Por sua vez o *tree walker* "NumberOfConditions" foi o mais executado, porém com uma eficiência considerada baixa (23%). Isto pode indicar que o uso deste *tree walker* poderia ser reconsiderado ou que as dicas geradas por ele precisam ser aprimoradas.

6. Considerações finais

Neste artigo foi apresentado um gerador de dicas para a resolução de problemas algorítmicos baseado na detecção de erros. Além disso, discutiu-se também a avaliação realizada com estudantes aprendizes de programação para mensurar a eficiência das dicas geradas durante a resolução de problemas previamente definidos.

O primeiro contato dos estudantes com o sistema de correção, e consequentemente com as dicas geradas por ele, foi de descoberta sobre o funcionamento do mesmo. Conforme os estudantes foram progredindo na resolução os exercícios tornou-se perceptível que eles usavam o corretor com mais segurança. De modo geral, a recepção dos estudantes ao sistema de correção e geração de dicas foi positiva. No experimento realizado constatamos que os estudantes não usaram as dicas para tentar burlar o sistema de correção tentando obter nota máxima. Isto pôde ser constatado tanto pelos dados de utilização quanto pela observação do comportamento dos estudantes durante a realização do experimento.

As dicas que os estudantes receberam do corretor estático não tiveram um efeito considerado satisfatório, apenas o *tree walker* "EmptyBlocks" conseguiu atingir uma taxa de eficiência considerada adequada. É importante salientar que este *tree walker* especificamente não é de grande utilidade para que o estudante encontre a solução de um determinado problema, mas é útil para melhorar a legibilidade do código.

Consideramos que o mecanismo de correção e geração de dicas proposto neste trabalho teve um impacto positivo e auxiliou os estudantes na resolução dos exercícios. Diferente do trabalho de Marceau et al. (2011) que focalizou em mensagens de erros sintáticos, nossa abordagem combinou a análise estática e dinâmica com foco em

auxiliar os estudantes a identificarem e corrigirem os erros relacionados ao problema que estavam resolvendo. O experimento com os aprendizes indicou além da viabilidade da abordagem a necessidade de ampliar a pesquisa com um número maior de estudantes e com o desenvolvimento de novos tree walkers.

7. Referencias

ALA-MUTKA, K. M., A Survey of Automated Assessment Approaches for Programming Assisments. Computer Science Education, p. 83-102, 2005.

DOUCE, C. Automatic Test-based Assessment of Programming: A Review, Journal on Educational Resources in Computing, Vol. 5, Issue 3, 2006.

IHANTOLA, P.; AHONIEMI, T.; KARAVIRTA, V.; SEPPÄLÄ, O. Review of recent systems for automatic assessment of programming assignments, Proceedings of the 10th Koli Calling International Conference on Computing Education Research, p.86-93, October 28-31, 2010, Koli, Finland

LAHTINEN, E.; ALA-MUTKA, K.; JÄRVINEN, H.-M. A study of the difficulties of novice programmers. Annual Sigcse Conference On Innovation And Technology In Computer Science Education, v. 37, n. 3, p. 14–18, Caparica, Portugal, 2005.

LEVENSHTEIN, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. Cybernetics and Control Theory, 10(8):707–710.

NOSCHANG, L. F. ; DE JESUS, E. A. ; PELZ, Fillipi ; RAABE, André Luís Alice . Portugol Studio: Uma IDE para Iniciantes em Programação. In: Workshop sobre Educação em Informática, 2014, Brasilia. Anais do Congresso Anual da Sociedade Brasileira de Computação. Porto Alegre: SBC, 2014. v. 1. p. 535-545.

MARCEAU, Guillaume; FISLER, Kathi; KRISHNAMURTHI, Shriram. Measuring the effectiveness of error messages designed for novice programmers. In: 42nd ACM technical symposium on Computer science education - SIGCSE '11. Procedings... Dallas, Texas, EUA, 2011.

PELZ, Fillipi ; DE JESUS, E. A. ; RAABE, André L.A. . Um Mecanismo para Correção Automática de Exercícios Práticos de Programação Introdutória. In: SBIE - Simpósio Brasileiro de Informática na Educação, 2012, Rio de Janeiro. Anais do Simpósio Brasileiro de Informática na Educação. Porto Alegre: SBC, 2012. v. 1. p. 398-408.

PELZ, F. Um Gerador De Dicas Para Guiar Novatos Na Aprendizagem De Programação, Dissertação de Mestrado (Mestrado em Computação Aplicada) - Universidade do Vale do Itajaí, Itajaí, SC, Brasil, 2014.

PELZ, Fillipi; RAABE, André L.A. . Análise do Feedback Fornecido por Corretores de Algoritmos com Propósito Educacional. In: Latin American Conference on Learning Objects (LACLO), 2013, Valdívia, Chile. Anais do LACLO 2013, 2013. v. 1. p. 236-247.