Um Arcabouço para Construção de Mecanismos de Análise de Códigos de Programação Introdutória

Marlos Tacio Silva[1,3], **Evandro de Barros Costa**[2]
Paulo Henrique Barbosa[2], **Juliana de Carvalho Cavalcante**[2]

[1] Instituto Federal de Alagoas – Piranhas – AL

[2]Universidade Federal de Alagoas – Maceió – AL

[3]Universidade Federal de Campina Grande – Campina Grande – PB

marlos.silva@ifal.edu.br, evandro@ic.ufal.br

Abstract. *This paper presents a framework for building mechanisms for automatic analysis of introductory programming exercises. This framework consists of a structure that provides the combination of different types of analysis mechanisms, providing the configuration of various families of analyzers. For evaluating the framework, we conduct a study with 816 solutions submitted by students from two introductory programming classes. The results indicate the following groups of students: failed to overcome the syntactic barrier; failed to use the constructs of language in the correct manner; failed to propose structurally cohesive solutions. Finally, the results indicate that overcome the semantic barrier does not necessarily mean the construction of a solution of quality.*

Resumo. *O presente trabalho apresenta um arcabouço para a construção de mecanismos de análise automática de exercícios de programação introdutória. Esse arcabouço consiste em uma estrutura que possibilita a combinação de diferentes tipos de mecanismos de análise, provendo a fácil configuração de diversas famílias de analisadores. Para a avaliação do arcabouço foi realizado um estudo com 816 soluções submetidas por estudantes de duas turmas introdutórias de programação. Os resultados apontam a formação dos seguintes grupos de estudantes: não conseguiram ultrapassar a barreira sintática; não conseguiram utilizar os construtos da linguagem da maneira correta; não conseguiram propor soluções estruturalmente coesas. Por fim, os resultados indicam que ultrapassar a barreira semântica não significou, necessariamente, a construção de uma solução de qualidade.*

1. Introdução

O aprendizado de programação, sobretudo em turmas introdutórias, tem sido rotulada como uma tarefa complexa e, por conseguinte, tem sido alvo de diversos estudos (*e.g.*, [McCracken et al. 2001, Robins et al. 2003, Pears et al. 2007]), sendo inclusive rotulado como um dos grandes desafios no ensino de computação [McGettrick et al. 2005]. Tipicamente, as disciplinas introdutórias de programação apresentam uma grande quantidade de estudantes e, consequentemente, de exercícios a serem avaliados. Sobre esse aspecto, prover *feedback* em tempo hábil se mostra uma tarefa extremamente onerosa, causando sobrecarga no professor da disciplina [Ala-Mutka 2005].

Nesse contexto, ferramentas de correção de exercícios de programação têm sido utilizadas para automatizar a verificação de soluções, provendo um *feedback* imediato e diminuindo a carga de trabalho do professor. Habitualmente, os Juízes Online, como ficaram conhecidas essas ferramentas, verificam a solução com base na aplicação de um conjunto de casos de teste previamente cadastrados (*e.g.*, [de Souza et al. 2011, Petit et al. 2012, Chaves et al. 2013]). Entretanto, essa abordagem leva em consideração apenas o desempenho nos testes, não provendo informações mais detalhadas acerca de outras características da solução.

Acerca desse aspecto, é possível destacar um conjunto de iniciativas que fogem à regra e apresentam diferentes perspectivas de análise acerca de uma solução em programação. O trabalho de [Breuker et al. 2011] utiliza um conjunto de métricas de Engenharia de Software para analisar a qualidade do programa construído. O trabalho de [Pelz et al. 2012] utiliza métricas de similaridade estrutural e mais um analisador para verificar a ocorrência de determinadas estruturas de controle. O trabalho de [Bryce et al. 2013] utiliza uma abordagem que visa treinar a habilidade dos estudantes em encontrar erros em um programa.

Esse trabalho tem o objetivo de apresentar um arcabouço para construção de mecanismos de análise de soluções de programação com foco no ensino introdutório. Basicamente, o arcabouço consiste em uma estrutura que permita a combinação de diferentes tipos de analisadores (*e.g.*, analisador sintático, analisador semântico, analisador estrutural, analisador de construtos, entre outros). Para tal, o arcabouço será concebido com base em duas visões complementares: visão conceitual; e visão computacional. Do ponto de vista conceitual, será utilizada uma abordagem para solucionar problemas baseados em múltiplos critérios conhecida como *Valuated State Space* [Michalewicz and Fogel 2004]. Do ponto de vista computacional, o arcabouço será construído com base no padrão de projeto *Decorator*, que tem o objetivo anexar responsabilidades adicionais a um objeto dinamicamente [Gamma et al. 1995], possibilitando a construção de diferentes configurações entre os analisadores.

Para a avaliação da proposta será apresentado um estudo com soluções submetidas por estudantes de duas turmas introdutórias de programação do curso de Sistema de Informação, modalidade a distância, da Universidade Federal de Alagoas. Nesse estudo foram analisadas soluções em quatro atividades iniciais que correspondem, respectivamente, aos seguintes tópicos: variáveis e comandos de entrada e saída; operadores e expressões; estruturas de seleção; e estruturas de repetição. Para a execução do estudo será utilizada uma instância do arcabouço com os seguintes analisadores: um analisador sintático; um analisador de construtos; e um analisador estrutural.

Os resultados indicam os seguintes grupos de estudantes: não conseguiram ultrapassar a barreira sintática; não conseguiram utilizar os construtos da linguagem da maneira correta; não conseguiram propor soluções estruturalmente coesas. Além disso, foi possível identificar um grupo de estudantes que submeteram soluções altamente coesas com a solução esperada pelo professor da disciplina. Por fim, foi possível identificar um conjunto de estudantes que submeteram soluções contendo um conjunto de construções não esperadas, mas que se mostraram bastante coerentes com a proposta do problema, demonstrando um comportamento de programação excepcional.

2. Proposta

A construção do arcabouço tem como principal característica a composição de diferentes tipos analisadores de código (*e.g.*, analisador sintático, analisador semântico, analisador estrutural, analisador de construtos, entre outros). Tal composição tem a função de prover uma análise mais abrangente, pois permite que diferentes aspectos de um código possam ser verificados. Assim, serão apresentadas duas visões complementares: a modelagem conceitual visa estabelecer os conceitos basilares do mecanismo de análise, fundamentando seu funcionamento a partir de uma técnica de tomada de decisão com múltiplos critérios denominada *Valuated State Space* [Michalewicz and Fogel 2004]; a modelagem computacional visa estabelecer uma base arquitetural para o arcabouço, destacando seus pontos de extensão por meio da utilização de um conjunto de Padrões de Projeto de Software [Gamma et al. 1995].

2.1. Modelo Conceitual

Conceitualmente um problema de programação é formado por dois elementos distintos: um conjunto de enunciados, que devem apresentar uma instrução clara e objetiva do problema a ser resolvido; e um conjunto de restrições, que definem os critérios que deverão ser cumpridos na solução a ser proposta. Tais critérios podem, por exemplo, definir um conjunto de linguagens de programação permitidas, um conjunto de entradas e saídas que serão verificados, um conjunto de construtos que devem ser utilizados, um conjunto de soluções de referência que servirão como parâmetro de comparação, entre outros.

Definição 1 *Seja P um problema de programação, um enfoque para esse problema pode ser definido por $P = (E, R)$, onde E denota um conjunto de enunciados que descrevem o problema e R denota um conjunto de restrições sobre esse problema.*

Uma solução para esse problema pode ser descrita por meio de um conjunto de símbolos pertencentes ao alfabeto de uma linguagem de programação.

Definição 2 *Seja S uma solução, um enfoque para essa solução pode ser definida por $S = \{\sigma_1, \cdots, \sigma_i, \cdots, \sigma_{|S|}\}$, onde cada $\sigma_i \in \Sigma$ denota um símbolo utilizado para descrever essa solução. Esses símbolos são definidos por um alfabeto Σ que, nesse contexto, definem uma linguagem de programação qualquer.*

Um conjunto de analisadores visa verificar se determinada solução viola alguma restrição do problema. Cada analisador é definido por uma função que determina um nível de conformidade entre uma solução e uma determinada restrição, onde o máximo valor representa máxima conformidade e o mínimo valor representa mínima conformidade.

Definição 3 *Sejam P e S, respectivamente, um problema e uma solução, existe um conjunto de mecanismos que visa analisar atributos dessa solução frente às restrições definidas no problema. Esses mecanismos são definidos por $A : P \times S \rightarrow \{a : 0 \leq a \leq 1\}$, onde cada $a \in A$ denota um determinado mecanismo de análise.*

As definições anteriores utilizam uma visão simplificada de um problema de programação, que para essa proposta se mostra suficiente. Nesse sentido, é importante ressaltar que o conjunto de restrições e o conjunto de analisadores representam um ponto de extensão do arcabouço. Desse modo, novas restrições e, consequentemente, novos analisadores podem ser adicionados no intuito de estender o mecanismo de análise.

Para possibilitar uma quantificação dos diversos analisadores foi utilizada a técnica *Valuated State Space* [Michalewicz and Fogel 2004]. Nessa técnica, um tomador de decisões (*e.g.*, um especialista em programação) determina um nível para cada um dos analisadores. Para definição dos níveis pode-se utilizar, por exemplo, uma escala de 0 até 5, todavia, outras escalas poderão ser usadas. Assim, os analisadores poderão apresentar diferentes níveis de influência, o que irá impactar no cálculo da pontuação resultante.

Definição 4 *Seja A o conjunto de mecanismos de análise, atribui-se um nível de relevância para cada um dos analisadores. Os níveis de relevância são definidos por $L : A \rightarrow \{l \in \mathbb{N} : 0 \leq l \leq n\}$, onde cada $l \in L$ denota o nível de um determinado analisador e a constante n denota o valor máximo da escala.*

O conjunto de níveis representa um sistema de ponderação mais intuitivo, entretanto, é necessário relativizar os valores definidos pelo especialista humano para que o cálculo da pontuação resultante fique compreendido no intervalo entre 0 e 1. Para tal será utilizado um conjunto de pesos que visa extrair e medida relativa de cada mecanismo de análise.

Definição 5 *Seja L o conjunto de níveis, atribui-se um peso para cada um dos seus elementos visando relativizar as medidas de relevância. Os pesos são definidos por $W : L \rightarrow \{w \in \mathbb{R} : w = l/sum(L)\}$, onde cada $w \in W$ denota o peso de um mecanismo de análise e sum denota o somatório dos elementos do conjunto de níveis.*

Para calcular a pontuação resultante, será efetuado o produto de cada analisador elevado ao seu respectivo peso [Michalewicz and Fogel 2004]. A partir dessa combinação será obtido um valor resultante, onde cada um dos avaliadores apresentará sua "contribuição" para a pontuação final.

Definição 6 *Sejam P e S, respectivamente, um problema e uma solução, uma função $score : P \times S \rightarrow \{s \in \mathbb{R} : s = \prod a(P,S)^w\}$ visa atribuir uma pontuação que consiste na combinação entre os mecanismos de análise $a \in A$ e os seus respectivos pesos $w \in W$.*

Em síntese, tal mecanismo tem a função de analisar uma determinada solução sob diferentes perspectivas, provendo uma visão mais abrangente, e extrair uma pontuação com base em critérios definidos previamente por um especialista humano. Com relação aos critérios de avaliação é importante destacar dois pontos:

Analisador Crítico: afeta diretamente a cadeia de análise. Um analisador sintático, por exemplo, representa um ponto crítico em todo processo de análise, pois caso a solução falhe (*i.e.*, resultado igual a zero), talvez não seja necessária a contabilização dos demais avaliadores (*e.g.*, analisador semântico). Com relação a esse ponto, a função de $score$ irá propagar o efeito desse analisador e, assim, a pontuação resultante sempre irá apresentar valor igual a 0 (zero). É importante ressaltar que a utilização desse mecanismo representa uma decisão no projeto do mecanismo de análise. Caso o uso de analisadores críticos não seja desejável, basta alterar o mecanismo de $score$ para a utilização de uma média aritmética ponderada;

Analisador Nulo: em um determinado contexto esse tipo de analisador não deve afetar a cadeia de análise. Um analisador de construtos, por exemplo, pode não ser importante para determinado problema e, por conseguinte, não deveria apresentar qualquer influência no resultado final. Nesse sentido, bastaria ao especialista definir esse avaliador com nível de relevância mínimo (*i.e.*, nível igual a zero) e, assim, tal avaliador não irá afetar na contabilização dos demais.

2.2. Modelo Computacional

Para a construção do mecanismo de análise de soluções investiu-se em um estilo arquitetural baseado em *Pipes* e *Filters*. Esse estilo arquitetural define um conjunto de elementos (*filters*) e um fluxo de processamento (*pipes*), permitindo a reconfiguração dos elementos para a construção de diferentes famílias de sistemas. Como ilustrado na Figura 1, o mecanismo de análise define um analisador sintático como elemento inicial do processo e, a partir deste, ramifica seu processamento através de outros analisadores.

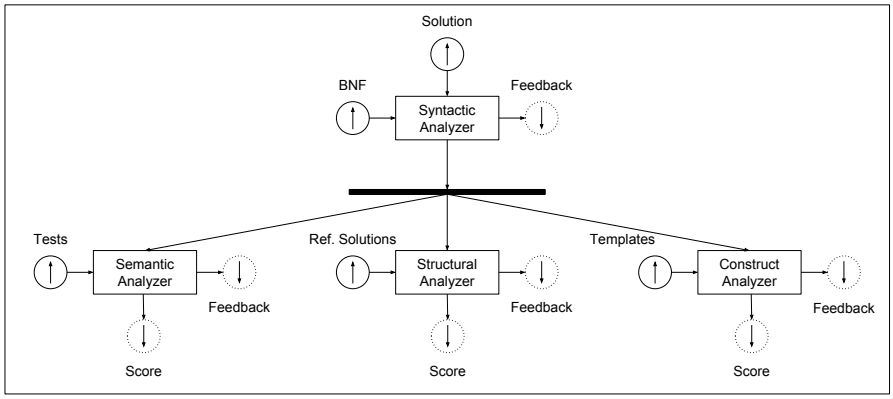

Figura 1. Arquitetura Conceitual

Dentre os analisadores ilustrados destaca-se o analisador estrutural, que utiliza métricas de similaridade para comparar uma determinada solução com um conjunto de soluções de referência, e o analisador de construtos, que utiliza técnicas de teoria dos grafos para verificar se uma determinada solução satisfaz um modelo composto por estruturas de controle. Além disso, cada uma das etapas de análise gera, além de uma pontuação, um *feedback* que poderá ser utilizado tanto por estudantes quanto por professores.

Para modelar a estrutura de analisadores utilizou-se o padrão de projeto *Decorator*, que visa anexar responsabilidades adicionais a um objeto (vide Figura 2). Esse padrão define uma entidade abstrata para representar as características comuns entre os diversos tipos de analisadores. A partir disto, define uma hierarquia composta por duas entidades mais específicas: a primeira define um analisador concreto, doravante denominado de analisador primário, que será a base de todo o processo de análise; e a segunda define um ponto de extensão para a criação de filtros, doravante denominados analisadores secundários, que serão anexados ao analisador primário.

Para modelar o analisador primário definiu-se a análise sintática como ponto fundamental de todo o processo. Para esse analisador utilizou-se a combinação de dois padrões de projeto: o padrão de projeto *Builder* para abstrair o mecanismo de análise sintática e, consequentemente, o processo de construção da árvore sintática abstrata de uma determinada solução; o padrão de projeto *Composite* para compor objetos em estruturas de árvores para representar uma árvore sintática abstrata. A combinação desses dois padrões define um ponto de extensão que visa abstrair a linguagem de programação utilizada.

Para modelar os analisadores secundários utilizou-se o padrão de projeto *Template Method* que visa definir o esqueleto de um algoritmo permitindo que suas subclasses redefinam determinados passos sem alterar sua estrutura. Nesse sentido, foi definido um

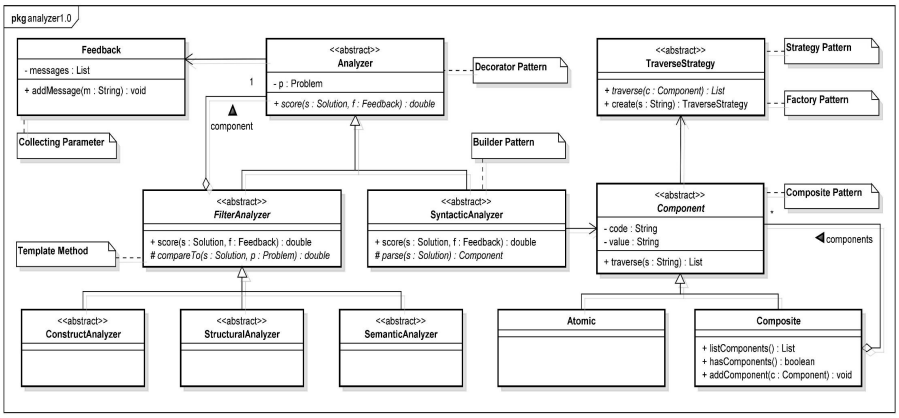

Figura 2. Arquitetura Lógica

ponto de extensão para construção de analisadores que visam conferir uma determinada solução face as restrições impostas por um problema.

Por fim, foram utilizados outros padrões para prover algumas funcionalidades desejadas: o padrão de projeto *Strategy* para encapsular algoritmos de caminhamento em árvore que são utilizados principalmente nos analisadores estruturais e de construtos; o padrão *Factory* para definir um ponto central para a criação dos algoritmos de caminhamento; e o padrão *Collecting Parameter* para armazenar as informações obtidas a partir da execução dos analisadores no intuito de prover algum tipo de *feedback* para o usuário.

3. Avaliação

Para avaliar o arcabouço foi realizado um estudo inicial com códigos submetidos por estudantes de turmas de Algoritmo e Estrutura de Dados I dos períodos letivo de 2010 e 2013 do curso de Sistema de Informação da Universidade Federal de Alagoas. As seções a seguir descrevem o planejamento e execução do estudo, bem como os resultados obtidos.

3.1. Planejamento e Execução

Para efetuar o estudo serão utilizadas as soluções submetidas pelos estudantes das turmas introdutórias de programação do curso de Sistema de Informação da Universidade Federal de Alagoas. Mais especificamente, serão analisados 816 códigos em linguagem *Python* divididos em quatro atividades iniciais da disciplina que correspondem, respectivamente, aos seguintes tópicos: variáveis e comandos de entrada e saída; operadores e expressões; estruturas de seleção; e estruturas de repetição. Cada solução será definida como uma unidade experimental do estudo e, por conseguinte, o estudo será estratificado com base nas atividades da disciplina.

Para a execução do estudo será utilizada uma instância do arcabouço contendo os seguintes analisadores: analisador sintático baseado na linguagem *Python* 3.x, linguagem padrão utilizada na disciplina; analisador estrutural baseado na métrica de similaridade conhecida como distância de edição de *Levenshtein* [Levenshtein 1966], para analisar a similaridade com a solução de referência provida pelo professor da disciplina; e analisador de construtos baseado na técnica conhecida como *Tree Inclusion* [Kilpelainen and Mannila 1995],

para analisar a conformidade com um modelo de estruturas de controle previamente definido. Aos analisadores será atribuído o mesmo nível de relevância. Além disso, é importante ressaltar que nesse estudo não será utilizado o analisador semântico, haja vista que as soluções utilizadas não seguem um padrão estrito de entradas e saídas. Sendo assim, será realizada uma inspeção visual das soluções submetidas.

3.2. Resultados e Discussão

Os resultados obtidos após a execução do estudo estão sumarizados na Figura 3 e na Figura 4. Considerando esses resultados, inicia-se uma discussão no intuito de ressaltar os seguintes aspectos:

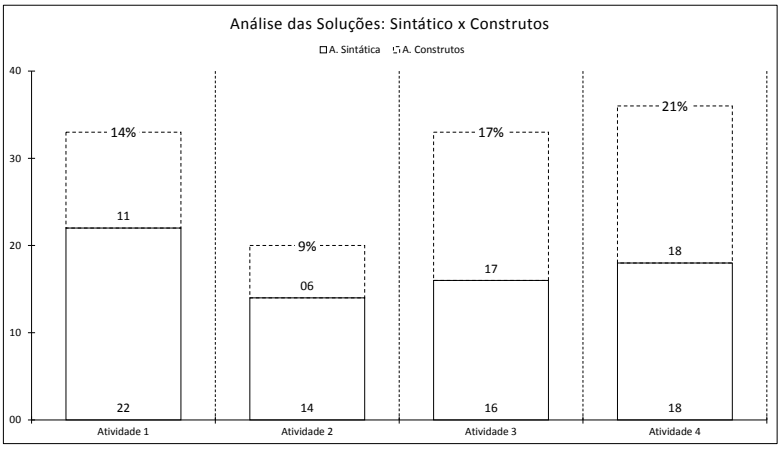

Figura 3. Análise das Soluções (Sintático x Construtos): as barras inferiores representam a quantidade de soluções que falharam no analisador sintático; as barras superiores representam a quantidade de soluções que falharam no analisador de construtos; as taxas superiores representam a porcentagem de soluções que falharam em um dos dois analisadores.

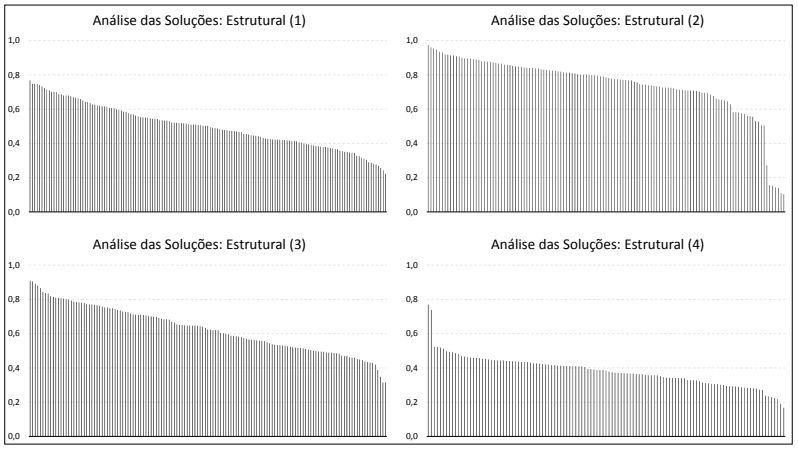

Figura 4. Análise das Soluções (Estrutural): cada um dos gráficos representa a pontuação obtida no processo de análise estrutural em cada uma das quatro atividades; os resultados estão organizados em ordem decrescente.

Analisador sintático: os resultados apontam um baixo número de estudantes ($\mu \approx 17$) que não conseguem vencer a barreira sintática. Além disso, é importante notar que esse número não sofre grandes alterações ao longo do tempo. Essa situação se mostra inesperada, pois previa-se que os erros sintáticos fossem diminuindo ao longo das atividades. Uma inspeção mais detalhada dessas soluções revelou diversos tipos de problemas

sintáticos, entre eles é possível destacar: soluções descritas na linguagem algorítmica vista na primeira semana do curso; soluções contendo os registros do modo interativo do *Python*; soluções descritas em versões antigas da linguagem *Python*; outros tipos de erros sintáticos mais básicos.

Analisador de construtos: os resultados apontam um baixo número de estudantes ($\mu = 13$) que não conseguem utilizar as estruturas de controle da maneira esperada. Esse número aumenta sobretudo nas atividades que envolve estruturas de seleção e estruturas de repetição. Uma inspeção mais detalhada dessas soluções revelou que grande parte dos erros reside no alinhamento das estruturas de controle. Sobretudo nas estruturas de seleção, foi possível notar que os estudantes não compreenderam a diferença entre as estruturas *if*, *elif* e *else*. Assim, as soluções utilizam apenas a estrutura *if* para verificar todas as condições, o que não necessariamente faz com que o programa funcione incorretamente, contudo, dada a natureza introdutória da disciplina, era esperado que as soluções utilizassem estruturas alinhadas. Adicionalmente, o analisador de construtos se mostrou útil na descoberta de soluções excepcionais. Na segunda atividade, por exemplo, era esperado que os estudantes calculassem uma raiz quadrada importando uma função do módulo *math*, no entanto, alguns estudantes utilizaram um caminho alternativo, aplicando o operador de potenciação da linguagem;

Analisador Estrutural: os resultados apontam uma taxa média de conformidade para com a solução de referência ($\mu \approx 0.52$). Com relação a essas taxas uma análise mais profunda aponta que soluções com baixa conformidade (≤ 0.30) apresentam inúmeras construções equivocadas e, geralmente, um conjunto demasiado de estruturas desnecessárias. Foi possível notar, sobretudo na quarta atividade (a mais complexa dentre as quatro), que essas soluções não seguem qualquer tipo de planejamento e, geralmente, são resultado da cópia de trechos de programas realizados anteriormente.

Em síntese, o analisador de soluções se mostrou útil na verificação das soluções dos estudantes, pois com ele foi possível identificar diversos tipos de comportamento inesperado como, por exemplo, utilização de versões antigas da linguagem, utilização de estruturas de controle não usuais, o não alinhamento de estruturas de controle, entre outros. Tais características não poderiam ser notadas por analisadores convencionais, haja vista que estes geralmente realizam apenas verificações sintáticas e semânticas. Além disso, o arcabouço aqui proposto permite a adição de novos tipos de analisadores (*e.g.*, analisador de plágio) e novas versões dos analisadores já existentes (*e.g.*, novas métricas de similaridade para comparação estrutura).

4. Trabalhos Relacionados

Para o levantamento dos trabalhos relacionados buscou-se focalizar trabalhos que apresentem algum diferencial com relação as abordagens centradas apenas na verificação de entradas e saídas. Para essas abordagens visa-se evidenciar seus limites e possibilidades frente ao arcabouço aqui proposto.

O trabalho de [Breuker et al. 2011] utiliza um conjunto de métricas de Engenharia de Software para analisar a qualidade do programa construído. Diferentemente do arcabouço aqui proposto não são utilizados outros tipos de análise. Contudo a abordagem baseada em métricas de qualidade se mostra bastante interessante e poderia ser integrada ao arcabouço no intuito de aumentar a abrangência do mecanismo de análise.

O trabalho de [Pelz et al. 2012] apresenta um mecanismo que combina um analisador estrutural e um analisador de construtos. Assim como o arcabouço aqui proposto, essa abordagem utiliza a ideia de combinar diferentes analisadores para prover uma verificação mais abrangente. Todavia, um dos diferenciais do arcabouço aqui proposto reside na base conceitual definida pela técnica *Valuated State Space* e na arquitetura lógica que permite fácil extensão e reconfiguração dos mecanismos de análise.

Por fim, destaca-se um conjunto de trabalhos como base para direcionamento futuros. O trabalho de [Bryce et al. 2013] utiliza uma abordagem que visa treinar a habilidade dos estudantes em encontrar erros em um programa. Os trabalhos de [Wang et al. 2012, Rong et al. 2012] utilizam uma abordagem baseada em revisão por pares que visa treinar a habilidade de análise dos estudantes. Tais trabalhos poderiam ser futuramente integrados ao arcabouço aqui proposto e, desse modo, forneceriam uma diversidade maior de ferramentas para os professores de programação.

5. Conclusão

O presente trabalho apresentou a concepção de um arcabouço para a construção de mecanismos de análise de soluções de programação introdutória. Desse arcabouço destacam-se o modelo conceitual definido pela técnica *Valuated State Space* e o modelo computacional que foi refinado por *Padrões de Projeto de Software*.

Dentre os resultados obtidos é importante ressaltar a identificação de diversos grupos de estudantes, dentre os quais pode-se destacar: estudantes com problemas a nível sintático; estudantes com problemas na utilização de estruturas de controle; e estudantes com problemas em construir soluções que mantenham um nível mínimo de conformidade com as expectativas do professor da disciplina. Além disso, destaca-se a identificação de soluções que superaram as expectativas definidas, ao apresentaram construções incomuns para estudantes iniciantes.

Por fim, espera-se que o arcabouço apresentado tenha gerado uma contribuição para a área de aprendizagem de programação. Nesse sentido, acredita-se que novos analisadores serão desenvolvidos e novos estudos serão realizados no intuito de garantir uma maior abrangência da abordagem apresentada.

Referências

Ala-Mutka, K. M. (2005). A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15(2):83–102.

Breuker, D. M., Derriks, J., and Brunekreef, J. (2011). Measuring static quality of student code. In *Proceedings of the 16th ACM conference on Innovation and technology in computer science education - ITiCSE '11*, page 13. ACM Press.

Bryce, R. C., Mayo, Q., Andrews, A., Bokser, D., Burton, M., Day, C., Gonzolez, J., and Noble, T. (2013). Bug Catcher: A System for Software Testing Competitions. In *ACM Technical Symposium on Computer Science Education*, page 513. ACM Press.

Chaves, J. O. M., Castro, A. F., Lima, R. W., Lima, M. V. A., and Ferreira, K. H. A. (2013). Integrando Moodle e Juízes Online no Apoio a Atividades de Programação. In *Simpósio Brasileiro de Informática na Educação*, pages 244–254. Sociedade Brasileira de Computação.

de Souza, D. M., Maldonado, J. C., and Barbosa, E. F. (2011). ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In *IEEE Conference on Software Engineering Education and Training*, pages 1–10. IEEE.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc.

Kilpelainen, P. and Mannila, H. (1995). Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2):340–356.

Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10(8):707–710.

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., and Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, page 125. ACM Press.

McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G., and Mander, K. (2005). Grand Challenges in Computing: Education–A Summary. *The Computer Journal*, 48(1):42–48.

Michalewicz, Z. and Fogel, D. B. (2004). *How to Solve It: Modern Heuristics*. Springer, 2nd edition.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *ACM Technical Symposium on Computer Science Education*, volume 39, pages 204–223.

Pelz, F. D., de Jesus, E. A., and Raabe, A. L. A. (2012). Um Mecanismo para Correção Automática de Exercícios Práticos de Programação Introdutória. In *Simpósio Brasileiro de Informática na Educação*. Sociedade Brasileira de Computação.

Petit, J., Giménez, O., and Roura, S. (2012). Jutge.org: An Educational Programming Judge. In *ACM Technical Symposium on Computer Science Education*, page 445. ACM Press.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172.

Rong, G., Li, J., Xie, M., and Zheng, T. (2012). The Effect of Checklist in Code Review for Inexperienced Students: An Empirical Study. In *IEEE Conference on Software Engineering Education and Training*, pages 120–124. IEEE.

Wang, Y., Li, H., Feng, Y., Jiang, Y., and Liu, Y. (2012). Assessment of programming language learning based on peer code review model: Implementation and experience report. *Computers & Education*, 59(2):412–422.