Modelo de identificação de unidades de conhecimento de programação em processo de aplicação durante a codificação

Tiago R. Kautzmann[1], **Patricia A. Jaques**[1]

[1]Programa de Pós-Graduação em Computação Aplicada (PPGCA)
Universidade do Vale do Rio dos Sinos (UNISINOS) – São Leopoldo – RS – Brasil

tkautzmann@gmail.com, pjaques@unisinos.br

Abstract. *Studies that presented models for identifying students' knowledge in programming in computing learning environments did not investigate ways to identify knowledge units in the process of being applied during coding. This type of information can be useful for decision making in adaptive environments, such as offering support when the student is trying to apply specific knowledge. This work presents and evaluates a model based on syntax analysis. It accompanies the student coding and identifies units of programming knowledge in application. The results showed a good agreement between the model's inferences and experts' judgments.*

Resumo. *Trabalhos que apresentaram modelos de identificação do conhecimento de alunos em programação, em ambientes computacionais de aprendizagem, não investigaram formas de identificar unidades de conhecimento em processo de serem aplicadas, durante a codificação. Este tipo de informação pode ser útil para tomadas de decisão de ambientes adaptativos, como oferecer suporte quando o aluno está tentando aplicar determinados conhecimentos. Este trabalho apresenta e avalia um modelo baseado em análise sintática que acompanha a codificação do aluno e identifica unidades de conhecimento de programação em processo de aplicação. Os resultados mostraram uma boa concordância entre as inferências do modelo e os julgamentos de especialistas.*

1. Introdução

Sistemas adaptativos de aprendizagem são ambientes computacionais de ensino que modificam suas ações de acordo com características e necessidades do estudante, proporcionando um ensino individualizado. Um exemplo são os sistemas tutores inteligentes (STI), que acompanham os passos do aluno, representam o conhecimento especialista no domínio (álgebra, programação, etc), modelam características do aprendiz (conhecimento, desempenho, estados afetivos, etc) e aplicam estratégias pedagógicas [Burns and Capps 1988, Woolf 2008]. Uma importante característica desses sistemas é o acompanhamento dos passos do aluno para coleta de informações sobre o desenvolvimento das tarefas, as quais podem ser usadas nas tomadas de decisão do ambiente [Vanlehn 2006]. O presente trabalho está interessado no acompanhamento da codificação do aluno em tarefas de programação de computadores, para identificação de unidades de conhecimento[1] (*If...Else*, declaração de variáveis, etc) em processo de aplicação.

[1]No contexto de STIs, cada unidade de conhecimento pode representar um princípio, um conceito no domínio ou qualquer informação utilizada para realizar uma tarefa [Vanlehn 2006].

DOI: 10.5753/cbie.sbie.2020.982

Na literatura, trabalhos que buscaram identificar unidades de conhecimento de programação usaram *bayesian knowledge tracing* apenas após a codificação do aluno [Kasurinen and Nikula 2009, Vier et al. 2015, Porfirio et al. 2017]. Outro trabalho apresentou um método de estimação de conhecimento procedural[2] usando modelos cognitivos e *knowledge tracing* [Corbett and Bhatnagar 1997]. Os trabalhos que não identificaram unidades de conhecimento específicas, buscaram identificar outros aspectos relacionados ao conhecimento, como a eficiência, a complexidade e o estilo dos códigos, ou apresentaram mecanismos de correção automática, de identificação de erros (sintáticos, lógicos, funcionais, etc) e de fornecimento de *feedbacks*. Estes estudos utilizaram avaliadores de códigos-fonte [Ala-Mutka 2005, Liang et al. 2009, Porfirio 2020]. Estas avaliações podem ser dinâmicas, quando a aplicação do aluno está em execução; estáticas, sem executar a aplicação (avaliação do código-fonte); ou híbridas, uma mescla das anteriores [Ala-Mutka 2005]. Alguns aspectos que as avaliações dinâmicas podem verificar são a funcionalidade do código, como a execução de casos de testes para identificar o atendimento dos objetivos da tarefa [Jackson and Usher 1997], e a eficiência do código, como na verificação de métricas de utilização de hardware [Patel et al. 2015]. Alguns aspectos que avaliações estáticas podem verificar são o estilo de codificação [Jackson and Usher 1997], erros sintáticos [Singh et al. 2013] e plágio [Cheang et al. 2003].

O presente artigo descreve e avalia um modelo que acompanha a codificação do aluno - avaliação estática do código-fonte a cada tecla digitada pelo aluno - em tarefas de programação de computadores em ambientes *open-ended*[3]. O objetivo do modelo é identificar unidades de conhecimento (saídas do modelo) em processo de aplicação, quando o aluno está codificando e refletindo sobre suas ações e o código-fonte está inacabado e em desenvolvimento. Não foram identificados trabalhos com essas características. Este estudo é parte de uma pesquisa mais ampla que busca encontrar correlações entre o conhecimento prévio do aluno e as emoções relacionadas à aprendizagem, como a frustração e a confusão, no contexto de tarefas de programação de computadores. A identificação dessas correlações poderá auxiliar no desenvolvimento de detectores de estados afetivos. Com o objetivo de verificar as emoções demonstradas pelo aluno quando ele está tentando aplicar determinadas unidades de conhecimento de programação, trabalhos futuros utilizarão um mecanismo que acompanha a codificação do aluno e identifica unidades de conhecimento em processo de aplicação. Este mecanismo é o objeto de estudo do presente trabalho.

O modelo proposto utiliza um *parser*[4] e uma gramática de linguagem. *Parsers* têm sido utilizados em outros trabalhos na identificação de estruturas de programação [Porfirio et al. 2017, Wang et al. 2017]. Diferente destes trabalhos, o presente modelo acompanha a formação das estruturas sintáticas durante a codificação, a cada tecla digitada pelo aluno. O modelo vincula cada unidade de conhecimento a uma única estrutura sintática da linguagem. Por exemplo, a unidade de conhecimento *If...Else* é vinculada a uma estrutura *If...Else* da linguagem de programação. Para conseguir identificar as estruturas sintáticas quando o código-fonte está em processo de codificação, a gramática de linguagem do modelo proposto reflete não apenas formações completas das estruturas sintáticas, mas também alternativas incompletas, com erros e falsas concepções.

[2]Conhecimento relacionado à habilidades e procedimentos [Corbett and Bhatnagar 1997].

[3]Em ambientes *open-ended*, o indivíduo tem liberdade para codificar, sem precisar seguir uma sequência de etapas de codificação imposta pela interface do sistema.

[4]*Parsers* são analisadores sintáticos que reconhecem linguagens [Parr 2013].

Ambientes adaptativos de aprendizagem, como STIs, poderiam utilizar as saídas do modelo em suas tomadas de decisão. Por exemplo, um STI capaz de estimar o conhecimento prévio do aluno poderia, com o auxílio do modelo, identificar situações durante a codificação em que o aluno está tentando aplicar conhecimento que não domina. O ambiente poderia decidir dar algum suporte imediato ao estudante, como apresentar material de apoio sobre as unidades de conhecimento de programação que o aprendiz não domina.

As próximas seções (2 e 3) descrevem o modelo proposto e uma avaliação. A avaliação verificou o desempenho do modelo ao comparar as saídas do modelo com julgamentos de especialistas humanos. Os resultados das análises realizadas são apresentados na Seção 4. As considerações finais são descritas na Seção 5.

2. Modelo

O modelo vincula cada unidade de conhecimento a uma única estrutura da linguagem de programação. Uma estrutura pode representar uma instrução (*If...Else*, *Do...While*, declaração de variável, invocação de método, etc), uma expressão ou algum outro conceito de programação, como a declaração de um método em linguagens orientadas a objetos.

Tabela 1. Unidades de conhecimento consideradas na avaliação do modelo

#	**Unidade de conhecimento**
1	Expressão
2	Expressão de instanciação de classe
3	Expressão de criação de *array*
4	Declaração de variável local
5	Atribuição
6	*If*
7	*If...Else*
8	*Switch Case*
9	*Do...While*
10	*While*

#	**Unidade de conhecimento**
11	*For*
12	*Foreach*
13	Invocação de método
14	Corpo de método
15	Corpo de método construtor
16	Corpo de classe
17	Declaração de método
18	Declaração de método construtor
19	Declaração de atributo
20	Declaração de classe

O modelo pode ser utilizado com qualquer linguagem de programação baseada em código. Cada unidade de conhecimento deve ser vinculada a um único conceito de programação cuja estrutura seja possível de ser representada como regra de uma gramática de *parser*. Conceitos abstratos como *coesão* e *acoplamento* são mais difíceis de serem representados. A avaliação do modelo (ver Seção 3) considerou a linguagem Java, por ser utilizada nas turmas participantes da avaliação. As unidades de conhecimento consideradas (ver Tabela 1) foram selecionadas a partir das bases curriculares dessas turmas.

A Figura 1 mostra o fluxo de funcionamento do modelo. A cada tecla digitada, o modelo deve receber como entrada as seguintes informações: i) o conteúdo do código-fonte em edição (em Java, um arquivo com extensão *.java*); ii) o número da linha e iii) o número da coluna onde o aluno estava codificando no momento da coleta. A saída do modelo são as unidades de conhecimento identificadas. Os mecanismos internos do modelo (*reconhecedor de linguagem* e *inspetor de árvore sintática*) são descritos a seguir.

A Figura 2 mostra o fluxo de funcionamento do **reconhecedor de linguagem**. A entrada é o código-fonte do aluno. Primeiramente, o *lexer* (analisador léxico) rea-

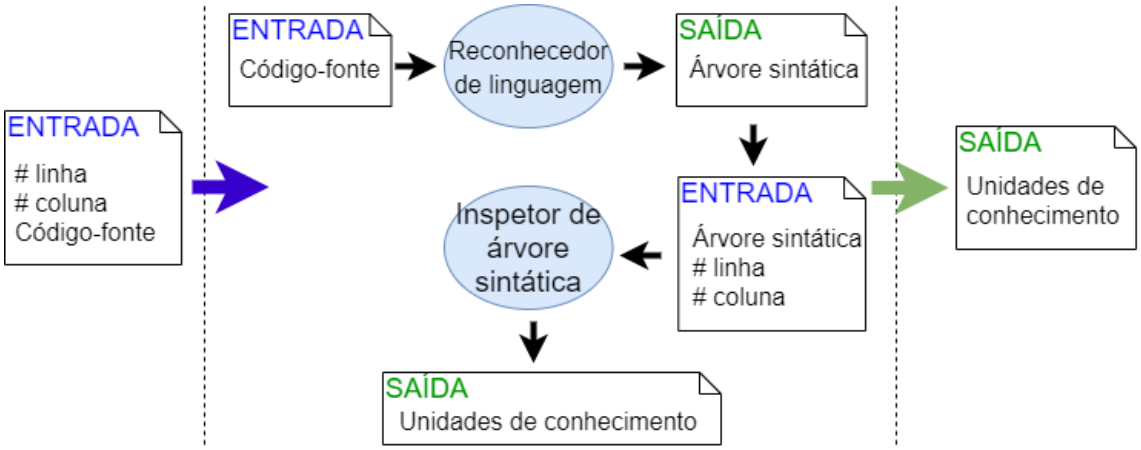

Figura 1. Fluxo de funcionamento do modelo

liza a leitura de um *stream* (caractere por caractere) do código, e agrupa os caracteres em sequências de palavras (*tokens*). Os *tokens* são entrada para o mecanismo de *parser* (analisador sintático). O *parser* utiliza as regras de uma gramática de linguagem de programação para tentar identificar estruturas da linguagem na sequência de *tokens*. Cada regra da gramática representa uma única estrutura (instrução, expressão, etc). A saída do reconhecedor de linguagem é uma árvore sintática abstrata que registra a forma como o *parser* reconheceu as regras. A Figura 4 mostra um exemplo de árvore sintática. Os nodos folhas representam os *tokens* e os nodos intermediários representam as regras referentes às estruturas. O nodo também guarda os números da linha e da coluna do início da regra (primeiro *token* da regra) no código-fonte e os números da linha e da coluna do fim da regra (último *token* da regra).

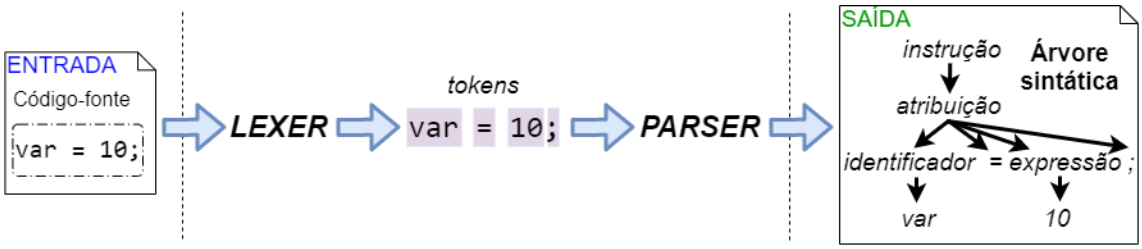

Figura 2. Fluxo de funcionamento do reconhecedor de linguagem

Para que o modelo consiga identificar estruturas em processo de codificação, a gramática precisa de regras com alternativas que representem estruturas completas, incompletas, com erros e falsas concepções. A Figura 3 mostra um fragmento de regra para a estrutura de uma instrução *If*, relacionada à unidade de conhecimento *If* (ver Tabela 1). Cada regra pode possuir mais de uma alternativa, separadas pelo caractere *pipe* ("|") [Parr 2013]. A primeira alternativa (*IF '(' expression ')' statement*) representa a estrutura completa da instrução. A segunda alternativa (*IF '(' expression ')' (';')? statement?*) representa uma falsa concepção: colocar um ponto e vírgula após o fecha parêntese da expressão condicional. As falsas concepções representadas na gramática deste trabalho foram fornecidas por um dos especialistas participante da avaliação, com 10 anos de experiência no ensino de programação. As demais alternativas da Figura 3 representam formas incompletas da estrutura. Quanto mais alternativas forem fornecidas a uma regra,

maior será a capacidade do modelo de identificar a estrutura em tempo de codificação. É preciso cuidar para não inserir alternativas redundantes na gramática [Parr 2013].

A avaliação do trabalho utilizou uma gramática base (disponível em *https://github.com/antlr/codebuff/blob/master/grammars/org/antlr/codebuff/Java8.g4*) que reflete a especificação da versão 8 da linguagem Java (disponível em *https://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html*). A gramática foi estendida neste trabalho para também representar formas incompletas, erros e falsas concepções. A regra original da Figura 3 apresentava apenas a primeira alternativa. Todas as demais alternativas foram acrescentadas neste trabalho. O mesmo ocorreu para todas as demais regras vinculadas às unidades de conhecimento da Tabela 1. A gramática completa utilizada está disponível em *https://github.com/tkautzmann/gramatica-java8-estudo*.

```
ifThenStatement
 : IF '('expression')'statement
 | IF '('expression')'(';')?statement?
 | IF '('')'statement?
 | IF '('statement?
 | IF ')'statement?
 | IF statement
 ;
```

Figura 3. Fragmento de regra gramatical

Figura 4. Ordem de visita aos nodos

O **inspetor de árvore sintática** recebe como entrada a árvore sintática gerada pelo reconhecedor de linguagem, além dos números de linha e de coluna onde o aluno estava codificando no momento da coleta da amostra. O inspetor percorre a árvore sintática, visitando cada nodo, e identifica as estruturas nas quais o aluno estava codificando. Ao visitar um nodo, o mecanismo verifica se os números de linha e de coluna do local de codificação do aluno pertencem ao intervalo de linha e de coluna da estrutura do nodo. Sabendo que cada estrutura corresponde a uma unidade de conhecimento e que a árvore sintática identifica a hierarquia das estruturas no código-fonte, o inspetor obtém como saída todas as unidades de conhecimento identificadas de forma hierárquica, em níveis. O nível 1 representa a unidade de conhecimento (estrutura sintática) mais próxima do local de codificação. A Figura 4 mostra um exemplo de árvore sintática para o trecho de código "*var = 10;*". A figura também mostra a ordem que o inspetor visita cada nodo da árvore. Se o aluno estivesse posicionado no caractere "1"do trecho, o inspetor encontraria as seguintes estruturas: *expressão* (nível 1), relacionada à unidade de conhecimento *Expressão*, e *atribuição* (nível 2), relacionada à unidade de conhecimento *Atribuição*.

Para a avaliação do modelo, o reconhecedor de linguagem e o inspetor de árvore sintática foram implementados com o auxílio do *ANTLR* (disponível em *https://www.antlr.org*), um *framework* gerador de *parsers* de linguagens [Parr 2013].

3. Avaliação

Uma avaliação verificou o desempenho do modelo em identificar unidades de conhecimento em processo de aplicação durante a codificação. Foram coletadas amostras de

códigos-fonte de alunos, enquanto realizavam tarefas de programação. Especialistas em programação visualizaram estes códigos e anotaram as unidades de conhecimento identificadas. Os mesmos códigos foram apresentados ao modelo. Por fim, as unidades identificadas pelo modelo foram comparadas com as unidades anotadas pelos especialistas.

3.1. Coleta de dados

Foram coletadas 425.554 amostras de estudantes de três turmas introdutórias de programação de computadores em três instituições de ensino localizadas no sul do Brasil: duas de ensino técnico e a outra de ensino superior. As amostras são de 24 alunos que retornaram assinado termos de consentimento e de assentimento livre e esclarecido[5]. Cada amostra foi gerada a partir da ação de pressionamento de tecla dos alunos no editor de código-fonte de uma versão modificada (para realizar a coleta das amostras) do ambiente *BlueJ*, enquanto os alunos resolviam tarefas de programação. Cada amostra guarda o conteúdo do arquivo de código-fonte em edição, além dos números de linha e de coluna de onde o aluno estava codificando no momento da coleta. Em função da pandemia de COVID-19 no período de coleta das amostras, entre março e junho de 2020, os estudantes realizaram as atividades de casa.

Em relação à coleta de julgamentos de anotadores humanos, foram selecionados cinco especialistas na linguagem Java que atendessem pelo menos um destes critérios: i) ter uma certificação oficial Java; ii) atender todos os seguintes subcritérios: ter experiência mínima de cinco anos com a linguagem Java; a linguagem Java ser a principal linguagem de programação que já utilizou (tempo e quantidade de código gerado).

Diferentes amostras foram atribuídas aleatoriamente (distribuição uniforme) aos anotadores (total de 750 amostras). Eles visualizaram os códigos-fonte e anotaram as unidades de conhecimento identificadas. Estas mesmas amostras também foram atribuídas ao modelo, cujas saídas (unidades de conhecimento) foram comparadas às anotações humanas. Um conjunto adicional de 30 amostras aleatórias foi atribuído somente aos anotadores (todos receberam o mesmo conjunto). Os julgamentos dos especialistas sobre este conjunto serviram para verificar o acordo entre os anotadores. No total, cada especialista anotou, no mínimo, 100 amostras, de acordo com sua disponibilidade (anotador 1 = 300 amostras; anotador 2 = 200; anotador 3 = 200; anotador 4 = 100; anotador 5 = 100).

Após uma etapa inicial de treinamento do processo de anotações, os especialistas registraram seus julgamentos em uma ferramenta *web* desenvolvida para este trabalho. A Figura 5 mostra um fragmento desta ferramenta. No exemplo da figura, o especialista anotou como nível 1, a partir do local onde o aluno estava codificando no momento da coleta (cursor em vermelho na linha em verde), a unidade de conhecimento *Expressão*. Como esta expressão é um argumento de uma invocação de método, o nível 2 foi anotado como *Invocação de método*. Os níveis 3 e 4 foram anotados, respectivamente, como *Corpo de método* e *Corpo de classe*.

3.2. Medidas

Este trabalho verificou o nível de acordo entre os anotadores através do índice *Randolph Kappa* [Randolph 2005], apropriado quando há mais de dois anotadores e eles são livres

[5]A presente pesquisa está registrada na Plataforma Brasil sob o CAAE (Certificado de Apresentação para Apreciação Ética) de número 24435519.2.0000.5344.

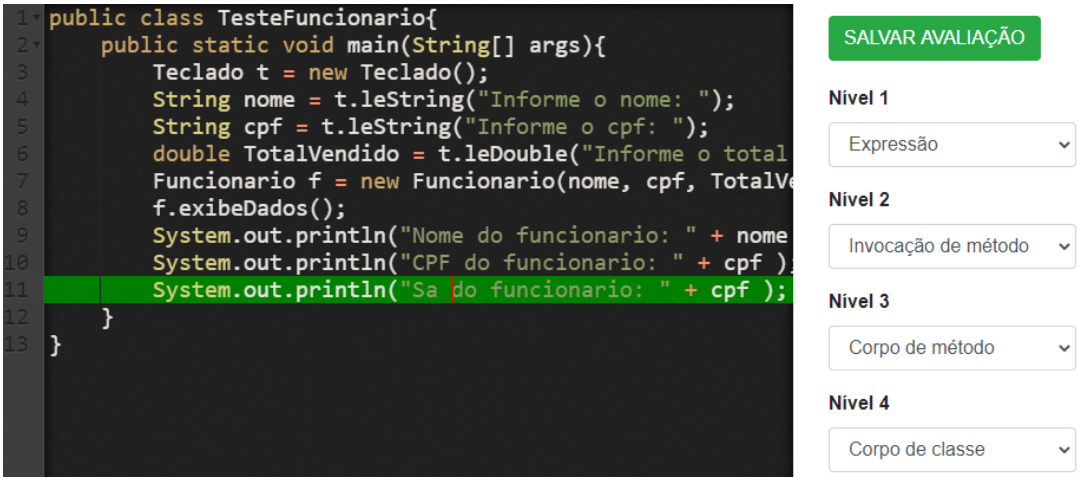

Figura 5. Fragmento do ambiente *web* utilizado para as anotação de amostras

para fazer seus julgamentos, sem precisar seguir uma distribuição específica de anotações [Brennan and Prediger 1981, Randolph 2005]. Julgamentos discordantes poderiam indicar algum bias dos anotadores, como o entendimento incorreto do processo de anotação. Isso poderia comprometer a qualidade das anotações e detrair os resultados.

O nível de concordância entre os julgamentos do modelo e dos anotadores foi verificado através do índice *Kappa de Cohen* (k) [Cohen 1960]. Ele reflete a diferença entre a concordância observada e a concordância esperada ao acaso, e é adequado para verificar o nível de acordo entre dois observadores independentes avaliando um mesmo objeto [Cohen 1960]. O k é um valor real entre -1,0 e 1,0. O valor 1,0 representa um acordo perfeito acima do acaso, enquanto que -1.0 representa um desacordo perfeito e 0.0 um acordo ao acaso. Valores inferiores a 0,4 indicam acordos leves e justos, enquanto que entre 0,4 e 0,6 são moderados e entre 0,6 e 0,8 são bons e substanciais. Índices acima de 0,8 são acordos quase perfeitos [Fleiss et al. 2004]. Também foram gerados índices de concordância observada (*total de concordâncias / total de julgamentos*).

4. Resultados

Esta seção apresenta diferentes análises sobre os resultados. Primeiramente, foi verificado o nível de concordância entre os anotadores, através da comparação dos julgamentos de nível 1 dos especialistas, no conjunto de 30 amostras atribuído a todos os anotadores. Foi encontrado 0,713 (concordância geral de 0,727) para *Randolph Kappa*. O índice encontrado demonstra um bom e substancial acordo [Fleiss et al. 2004] entre os anotadores. Esta é uma evidência positiva sobre a qualidade das anotações no restante das amostras.

A Tabela 2 mostra a concordância entre o modelo e os especialistas, quando comparados apenas os julgamentos de nível 1. Com *Kappa* de 0,786 e concordância observada (Co) de 0,816, o modelo demonstrou um bom e substancial desempenho em identificar unidades de conhecimento próximas da edição de código (nível 1). A Tabela 3 apresenta resultados de outras duas análises. A primeira comparou todos os julgamentos de cada nível (nível 1 com nível 1, nível 2 com nível 2, etc) e encontrou 0,593 de *Kappa* (Co = 0,664). Uma segunda análise desconsiderou a hierarquia dos níveis e realizou a intersecção dos julgamentos do modelo e dos anotadores, em cada amostra. Esta análise encontrou 0,799 de concordância observada.

Tabela 2. Concordância entre o modelo e os anotadores no nível 1

	n	***# Co* (%)**	***k***
Anotador 1	270	226 (0,837)	0,810
Anotador 2	170	134 (0,788)	0,757
Anotador 3	170	129 (0,759)	0,720
Anotador 4	70	61 (0,871)	0,848
Anotador 5	70	62 (0,886)	0,860
Todos	**750**	**612 (0,816)**	**0,786**

Tabela 3. Concordância observada nível à nível e na intersecção das anotações

	Nível a nível			**Intersecção das anotações**	
	itens	**# *Co* (%)**	***k***	**itens**	**# *Co* (%)**
Anotador 1	735	530 (0,721)	0,661	752	626 (0,832)
Anotador 2	455	290 (0,637)	0,555	470	365 (0,777)
Anotador 3	514	306 (0,595)	0,520	528	403 (0,763)
Anotador 4	209	125 (0,598)	0,510	213	167 (0,784)
Anotador 5	187	144 (0,770)	0,715	194	163 (0,840)
Todos	**2100**	**1395 (0,664)**	**0,593**	**2157**	**1724 (0,799)**

A Tabela 4 mostra resultados da concordância entre os anotadores e o modelo nos julgamentos de nível 1 em amostras de códigos com erros (*Cce*) e sem erros (*Cse*). A concordância foi quase perfeita quando os códigos não possuíam erros de codificação (k = 0,888, Co = 0,903). No entanto, o resultado mais significativo foi que mesmo em códigos com erros, o acordo encontrado foi bom e substancial (k = 0,741, Co = 0,778). Sabendo que a maior parte dos códigos possuem erros (69,7% das amostras), por serem coletados em tempo de codificação, quando estão inacabados, com erros ou refletindo falsas concepções, este resultado é uma evidência significativa sobre o bom desempenho do modelo em identificar unidades de conhecimento em processo de aplicação. Este resultado é animador para o uso do modelo em ambientes *open-ended* adaptativos de aprendizagem de programação, quando o aluno está aprendendo e espera-se que ele cometa erros.

Tabela 4. Concordância observada no nível 1 em amostras com erros e sem erros

	n	***# Cce* (%)**	***# Cse* (%)**	***# Co Cce* (%) [*k*]**	***# Co Cse* (%) [*k*]**
Anot. 1	270	202 (0,748)	68 (0,252)	164 (0,812) [0,778]	62 (0,912) [0,898]
Anot. 2	170	112 (0,659)	58 (0,341)	82 (0,732) [0,695]	52 (0,897) [0,880]
Anot. 3	170	117 (0,688)	53 (0,312)	83 (0,709) [0,665]	46 (0,868) [0,843]
Anot. 4	70	48 (0,686)	22 (0,314)	40 (0,833) [0,802]	21 (0,955) [0,945]
Anot. 5	70	44 (0,629)	26 (0,371)	38 (0,864) [0,824]	24 (0,923) [0,908]
Todos	**750**	**523 (0,697)**	**227 (0,303)**	**407 (0,778) [0,741]**	**205 (0,903) [0,888]**

5. Considerações finais

O artigo descreve o modelo de um mecanismo que identifica unidades de conhecimento de programação em processo de aplicação, durante a codificação de tarefas de programação. Os resultados mostraram que o uso de um analisador sintático com uma

gramática composta por regras que representam estruturas completas, incompletas, erros e falsas concepções dos alunos sobre a linguagem de programação, pode apresentar um bom desempenho na identificação das unidades de conhecimento em aplicação.

O mecanismo poderia ser utilizado em ambientes adaptativos de aprendizagem, como os STIs, auxiliando nas tomadas de decisão. Ao identificar, com o auxílio do modelo deste artigo e de um modelo de aluno mantido pelo ambiente, que o aluno está em processo de aplicar algum conhecimento que não domina, o sistema poderia decidir dar assistência ao aluno. Além disso, a representação hierárquica (em níveis) das unidades de conhecimento poderia auxiliar os ambientes computacionais a identificar informações sobre a aprendizagem do aluno em determinados contextos de aplicação de conhecimento. Por exemplo, um ambiente poderia identificar, com o auxílio do modelo, que um aluno possui uma dificuldade específica em aplicar a unidade de conhecimento *If* dentro do escopo de uma estrutura *Do...While*. O sistema poderia realizar alguma ação de suporte ao aluno relacionada a este contexto de aplicação de conhecimento.

Apesar dos resultados apresentarem uma concordância satisfatória entre os anotadores, foram encontradas evidências nas anotações discordantes de que os anotadores podem ter precipitado alguns julgamentos ao considerar informações contextuais não referentes às estruturas sintáticas, como a semântica dos identificadores. O treinamento dos anotadores poderia ter reforçado a necessidade de julgamentos baseados somente na observação das estruturas sintáticas.

Fornecer mais alternativas para as regras gramaticais pode melhorar o desempenho preditivo do modelo, mas levar a uma sobrecarga de processamento [Parr 2013] que torne inviável a execução do modelo para cada tecla digitada. Em computadores com baixo poder de processamento, a execução também poderia ser inviável. Nestes casos, os implementadores podem decidir mudar a periodicidade de execução do modelo, como a cada duas ou mais teclas pressionadas ou a cada intervalo de tempo. Além disso, os autores acreditam que não seja possível melhorar substancialmente a capacidade preditiva do modelo, pois definir regras para todas as falsas concepções ou erros que os alunos poderiam cometer em tempo de codificação é uma tarefa exaustiva [Sison and Shimura 1998].

Trabalhos futuros poderiam investigar outras formas de representação do problema, como através de modelos de aprendizagem profunda baseados em redes neurais. No entanto, como o treinamento destes modelos costumam exigir uma quantidade de amostras muitas vezes inviáveis de serem obtidas em determinados contextos educacionais, a solução apresentada neste trabalho parece ser satisfatória para grande parte dos contextos de aplicação dos ambientes adaptativos de aprendizagem.

Agradecimentos

Este trabalho foi realizado com apoio da Capes (código de financiamento 001), FAPERGS (processo 17/2551-0001203-8) e CNPq (processo 309218/2017-9).

Referências

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102.

Brennan, R. L. and Prediger, D. J. (1981). Coefficient kappa: Some uses, misuses, and alternatives. *Educational and Psychological Measurement*, 41(3):687–699.

Burns, H. L. and Capps, C. G. (1988). Foundations of intelligent tutoring systems: An introduction. *Foundations of intelligent tutoring systems*, pages 1–19.

Cheang, B., Kurnia, A., Lim, A., and Oon, W.-C. (2003). On automated grading of programming assignments in an academic institution. *C&E*, 41(2):121–131.

Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46.

Corbett, A. T. and Bhatnagar, A. (1997). Student Modeling in the ACT Programming Tutor: Adjusting a Procedural Learning Model With Declarative Knowledge. *User Modeling*, pages 243–254.

Fleiss, J. L., Levin, B., and Paik, M. C. (2004). The Measurement of Interrater Agreement. *Statistical Methods for Rates and Proportions*, pages 598–626.

Jackson, D. and Usher, M. (1997). Grading student programs using assyst. *SIGCSE Bull.*, 29(1):335–339.

Kasurinen, J. and Nikula, U. (2009). Estimating programming knowledge with Bayesian knowledge tracing. *ACM SIGCSE Bulletin*, 41(3):313.

Liang, Y., Liu, Q., Xu, J., and Wang, D. (2009). The recent development of automated programming assessment. *Proceedings of CiSE 2009*.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition.

Patel, A., Panchal, D., and Shah, M. (2015). Towards improving automated evaluation of java program. In *49th Convention of the CSI Volume 1*, pages 489–496. Springer.

Porfirio, A., Pereira, R., and Maschio, E. (2017). Atualização do Modelo do Aprendiz de Programação de Computadores com o Uso de Parser AST. *CBIE 2017*.

Porfirio, A. J. (2020). *Identifying evidences of computer programming skills through automatic source code evaluation*. PhD thesis, Universidade Federal do Paraná.

Randolph, J. J. (2005). Free-marginal multirater kappa (multirater κfree): An alternative to fleiss' fixed-marginal multirater kappa. In *JLIS 2005*.

Singh, R., Gulwani, S., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In *34th ACM SIGPLAN*, NY, USA.

Sison, R. and Shimura, M. (1998). Student Modeling and Machine Learning. *International Journal of Artificial Intelligence in Education*, 9:128–158.

Vanlehn, K. (2006). The behavior of tutoring systems. *International journal of artificial intelligence in education*, 16(3):227–265.

Vier, J., Gluz, J. C., and Jaques, P. A. (2015). Empregando Redes Bayesianas para modelar automaticamente o conhecimento dos alunos em Lógica de Programação. *Revista Brasileira de Informática na Educação*, 23(02):45.

Wang, L., Sy, A., Liu, L., and Piech, C. (2017). Deep knowledge tracing on programming exercises. *L@S 2017 - Proceedings of the 4th (2017) ACM L@S*, pages 201–204.

Woolf, B. P. (2008). *Building Intelligent Interactive Tutors: Student-Centered Strategies for Revolutionizing e-Learning*. Morgan Kaufmann, San Francisco, CA, USA.