Towards the Use of Slice-based Cohesion Metrics with Learning Analytics to Assess Programming Skills

Max Kesselbacher
Department of Informatics Didactics
University of Klagenfurt
Klagenfurt, Austria
max.kesselbacher@aau.at

Andreas Bollin
Department of Informatics Didactics
University of Klagenfurt
Klagenfurt, Austria
andreas.bollin@aau.at

***Abstract*—In programming education, it makes a difference whether you are dealing with beginners or advanced students. As our future students will become even more tech-savvy, it is necessary to assess programming skills appropriately and quickly to protect them from boredom and optimally support the learning process. In this work, we advocate for the use of slice-based cohesion metrics to assess the process of program construction in a learning analytics setting. We argue that semantically related parts during program construction are an essential part of programming skills. Therefore, we propose using cohesion metrics on the level of variables to identify programmers' trains of thought based on the cohesion of semantically related parts during program construction.**

***Index Terms*—programming education, assessment and feedback, static slicing, learning analytics**

I. Introduction

In his WiPSCE'16 keynote [1, p. 14], Raymond Lister stressed that, in his (neo-Piagetian) view[1], quite some steps are necessary to master writing one's own first programs, and curricula and teachers are not adequately prepared for facilitating the learning process. The situation seems to continue at the university level, where high failure rates are observed in programming courses [2], or are at least compounded by faulty teaching strategies [3].

Even if one is not a supporter of (neo-)Piagetian ideas, to develop the programming skills of our future software engineers in the best possible way, it is essential to understand the learning processes, to know more about the strategies experts follow when constructing their program, and to learn how they are mapping their trains of thought to running code. Early work by Weiser [4], Burnstein et al. [5], Broad and Filer [6] and also the application in the formal methods domain [7] already showed that programmers think in structures detectable by slices. This makes slice-based program analysis a promising avenue towards our aim, namely identifying code construction patterns and strategies that can then be used to support novice users in learning to program.

[1]In the neo-Piagetian view of Lister, learning to program occurs in four main stages of overlapping waves, with the ability to mentally execute, i.e. *trace*, program code as a cornerstone of novice programmers' progression.

Slicing techniques are usually applied on finished programs or successive program versions stored in software version control systems. With the emergence of learning analytics (LA) [8], [9], there is a great potential to improve individual assessment and feedback, even during program construction. This strongly affects the next generation of software engineers: with heterogenous previous programming experiences and diversity in the possible tools for learning, they need to be supported individually; a prospect made possible with LA.

We argue that the granularity of current slice metrics does not fit our established aims: assessing the *method-level* cohesion [10] alone is too coarse for individual feedback, while *statement-level* cohesion [11] is too fine-grained to reconstruct a programmer's train of thought.

We advocate the use of slice-based cohesion metrics on the *level of variables* and showcase its use by analyzing two *Java* implementations of the same example problem (implemented by an undergraduate student and by a professional programmer). We demonstrate that trains of thought can be identified in the program construction sequence. Our data sets and tools are publicly available [12].

II. Related Work

Static program slicing has been introduced by Weiser [13], and he showed that experienced programmers tend to think in program slices when reasoning about code, for example during debugging [4]. The computation of slice-based cohesion metrics, based on slice profiles, has been introduced by Ott and Thuss [10]. Their metrics (*Min/Max-Coverage*, *Overlap*, *Tightness*, *Parallelism*) make it possible to quantify the cohesion of a method (on *method-level*). These metrics have been empirically studied by Meyers and Binkley [14], providing baseline measures, establishing that the metrics can identify deteriorating software quality as a result of changes, and investigating the inter-metric relations for sets of metrics that provide a distinct viewpoint on program code. Krinke proposed an adaptation of the cohesion metrics on *statement-level*, measuring cohesive statements to help maintainers identify parts responsible for low module cohesion that should be

978-1-6654-3196-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SEENG53126.2021.00008

restructured [11]. For programming education, static slicing has been used as an instructional method to improve students' programming skills [15] but is not widely considered.

Slice-based cohesion metrics have been adapted to formal specifications [16], and Bollin has shown that high cohesion relates to single trains of thought, while low cohesion and small module slice intersections relate to multiple trains [7].

LA enables researchers to analyze educational data on a completely new scale [8], [9]. In programming, this includes recording program versions that are not considered finished, enabling a process-oriented analysis. Applications for text-based programming include large-scale investigations of programming mistakes in the *BlueJ* programming environment [17], clustering of program versions of novice programmers solving a specific problem [18], investigations of a state model for programming behaviour [19], and learning curve analysis for programming concepts [20].

We propose to use slice-based cohesion metrics in a LA setting for text-based programming education, following the ideas and findings of Bollin [7], to improve individual assessment and feedback for learning programmers. To the best of our knowledge, there is no previous work that applies slice-based cohesion metrics in a LA setting.

III. Showcase Study: Cohesion on Variable-Level

A. Participants and Example Problem

The example problem used for this showcase study is given in Figure 1 (a). We recruited participants from two populations (17 undergraduate and graduate students from a Software Engineering project management course at the University of Klagenfurt, and 9 professional programmers from an Austrian software development company) to implement the example problem in *Java*. We use this example problem specifically because there are multiple ways to solve the problem: syntactically (loops and conditional branching may be used but are not required) and semantically (computing the output numbers directly from the array or computing one number from the binary array and the second one from the first number).

With the example problem's openness, it is possible to test participants of varying programming skills while also providing potential differences to be assessed with a variety of metrics. In our showcase study, we use slice-based cohesion metrics to differentiate ways to solve the example problem.

We selected two implementations to showcase the use of slice-based cohesion metrics on *variable-level* in a LA setting: one implementation by an undergraduate programmer showing two trains of thought with little cohesion between parts (Section III-C), and one implementation by a professional programmer that features a single train of thought and high cohesion (Section III-D). Both implementations are fully functional and correctly implemented. We do not claim experimental validity by the selection of the implementations.

B. Collection and Computation of Cohesion Metrics

The implementation is recorded with an IDE-based LA approach [9], recording changes on *keystroke* granularity [8] (each keystroke and the resulting program version). The data collection is part of an ongoing project with a baseline collection of *keystroke* granularity [12]. We convert the data into a sequence of consecutive, compilable program versions.

The participants used the IDE *IntelliJ* to implement the example problem. The implementation was recorded with an IDE plugin. The consecutive program versions are stored on a headless data collection endpoint. Access to the data is possible at our data collection repository [12]. We adapted the slice-based cohesion metrics of Ott and Thuss [10] to compute the *Coverage* for each local variable and parameter of a method, and implemented an automatic computation of slice profiles and slice-based cohesion metrics for variables as an adaptation to the Java slicer *JRazor* [21]. The IDE plugin, the data collection server and the Java slicer are developed and hosted at our department and available as open source[2].

The metrics computation is based on unions of forward and backward slices, originally called *metric slices* [10]. We are focusing on semantics, simply calling them *union slices*. Our computation of *union slices* is similar to those of *metric slices*: the backward slice is computed from the last reference of the slicing variable, the forward slice is computed from the definitions of the slicing variable that are included in the backward slice. However, we only compute the forward slice from the first definition. This way, multiple *union slices* can be computed for a slicing variable: iteratively computing pairs of backward and forward slices for references and definitions of the slicing variable not covered so far, until all of them are included in at least one *union slice*. Different cohesive parts of a method can be found, each captured by a *union slice*. Our adaptation also makes the computation of the pairwise cohesion of all variables possible, which is not showcased here[3]. *Coverage* is computed as follows:

Definition 1: Coverage(M,v). Coverage, for method M and slice profile SP for variable v, is the average of the variable *union slice* lengths divided by the method length.

$$Coverage(M, v) = \frac{1}{|SP_v|} \sum_{i=1}^{|SP_v|} \frac{|UnionSlice(v)_i|}{|M|}$$

C. Showcase I: Undergraduate Programmer

The undergraduate programmer implements two computational parts to solve the example program (Figure 1 (b), lines 6-12). They compute the decimal sum (*dec*) and the two hexadecimal digits (*hD1 / hD2*) separately, controlled by the loop variable *i*. After the loop, they invoke `Integer.toHexString(...)` (line 13, in pseudo-code) to compute the hexadecimal output, and print both results.

The separated method concerns can be observed in the evolution of *Coverage* on variable-level (Figure 1 (d)). With the implementation of the second computation part (the addition of

[2]Source code available at: https://gitlab-iid.aau.at/seqtrex

[3]Additional information can be found in the Ph.D. thesis of the first author, currently as work in progress: https://www.aau.at/wp-content/uploads/2021/03/Kesselbacher_Thesis.pdf

```
1  | public class
   | ConvertBinary {
2  | // Converts the input
3  | // array of 8 bits, and
4  | // prints decimal
5  | // and hexadecimal.
6  | // Example input:
7  | // convertBinArr(
7  | // {1,1,0,0,0,0,1,1})
8  | // Example output:
9  | // 195 / C3
10 |  void convertBinArr(
   |   int[] bN){}}
```

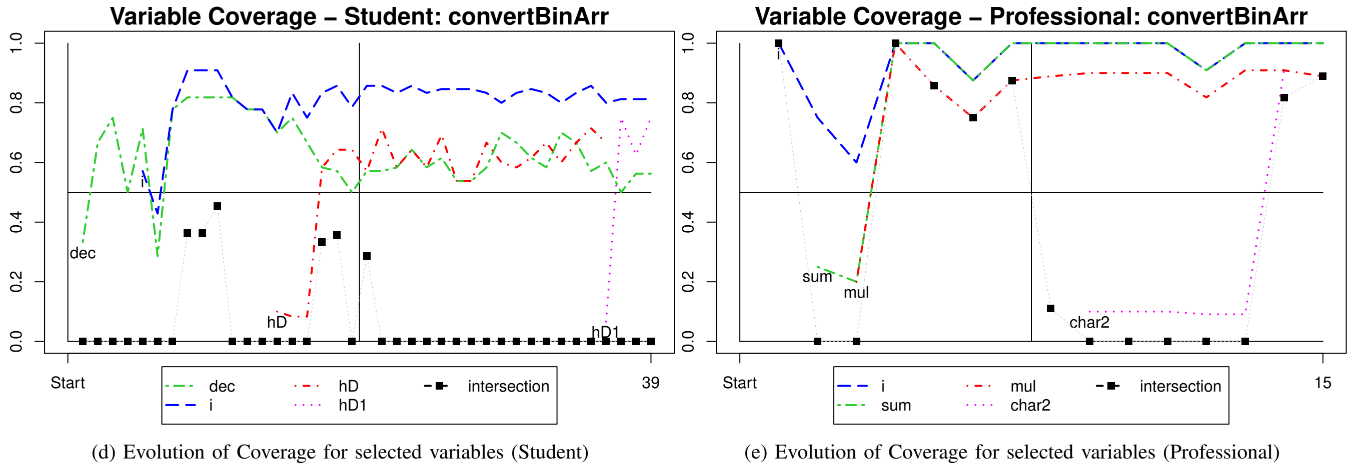
(a) Example problem

```
1  | void convertBinArr(
   |   int[] bN){
2  |  double dec=0;
3  |  String hex="";
4  |  double hD1=0;
5  |  double hD2=0;
6  |  for(int i=0;
   |    i<bN.length;i++){
7  |   if(bN[i]==1){
8  |    dec+=Math.pow(2,i);
9  |    if(i<=3)
10 |     hD2+=Math.pow(2,i);
11 |    if(i>3)
12 |     hD1+=Math.pow(2,i-4);}}
13 |  hex=hexString(hD1)
   |   +hexString(hD2);
14 |  out.println(dec);
15 |  out.println(hex);}
```

(b) Method cohesion matrix (Student)

```
1 | void convertBinArr(
  |   int[] bN){
2 |  int sum=0;
3 |  int mul=1;
4 |  for(int i=0;
  |    i<bN.length;i++){
5 |   sum+=mul*bN[i];
6 |   mul*=2; }
7 |  out.println(sum);
8 |  out.println(
  |   convertIntToStr(sum/16)
  |   +convertIntToStr(sum%16)
  |   );}
```

(c) Method cohesion matrix (Professional)



(d) Evolution of Coverage for selected variables (Student)

(e) Evolution of Coverage for selected variables (Professional)

Fig. 1: The top row shows the example problem (a) and end versions implemented by an undergraduate student (b) and a professional programmer (c). The bottom row shows the evolution of *Coverage* for selected variables and the relative method slice intersection for all consecutive, compilable program versions (undergraduate (d), professional (e)). Each point on the x-axis represents a compilable program version during program construction, starting with the first compilable program change (e.g. variable declaration) and ending with the program versions shown in the top row. The final number on the x-axis represents the number of consecutive, compilable program versions. The y-axis represents the *Coverage* and method intersection values.

the variable *hD*), the coverage of the control variable *i* remains high while the coverage of the data variables *dec* and *hD* are only moderately high. Towards the end, the undergraduate replaces *hD* with *hD1/hD2*, which further decreases the coverage of *dec*. Both data variables have a coverage greater than 0.5, but the relative method slice intersection is low during the whole method construction and even 0 for most program versions. This means that, for most program versions, there is no method statement that is included in all *union slices*. Even when removing line 3 (which causes the empty intersection), the relative method slice intersection is only 0.27.

D. Showcase II: Professional Programmer

The professional programmer follows a different approach to solve the example program. They only compute the decimal sum (*sum*) in their loop (Figure 1 (c), lines 4–6). To handle the hexadecimal output, they implement the method `intToStr` (not shown for brevity) to compute the hexadecimal digit corresponding to the integer parameter, and feed both half-bytes to this method (line 8).

The evolution of *Coverage* on variable-level (Figure 1 (e)) shows that the professional programmer eventually achieves a high *Coverage* for all variables, in contrast to the undergraduate's program construction sequence. During program construction, the relative method slice intersection decreases whenever a new variable is introduced (*sum / mul / char2*). Subsequently, they are integrated or refactored (in the case of *char2*, which was a temporary variable to hold half of the hexadecimal result), which results in a high coverage for all variables as well as a high relative method slice intersection.

The relative method slice intersection of the end version is $\frac{8}{9} = 0.\dot{8}$ - all method nodes other than the definition of *sum* in line 2 is included in every union slice. Note that the loop corresponds to three nodes in the method PDG.

IV. Discussion

A. Identifying Trains of Thought in the Implementations

Bollin argues that single trains of thought translate to slice intersections covering all predicates of cohesive Z specifications. Different thoughts result in smaller intersections [7]. By analogy, single trains of thought should result in cohesive method implementations and high relative method slice intersections. In this paper, we propose the notion of a *cohesive* method having a relative method slice intersection > 0.5, with the rationale that, above this threshold, all variable union slices should have method statements in common.

Following this notion, we can establish that the undergraduate implements multiple trains of thought: the variable-level *Coverage* metrics show a reasonably high cohesion greater than 0.5, the slice intersection never exceeds 0.5 and shows that there are no cohesive method parts across all variables. It is therefore important to look at both variable-level *Coverage* as well as method slice intersection measures.

The professional programmer integrates all variables into a cohesive method implementation, following a single train of thought. The slice intersection exceeds 0.5 multiple times during the program construction, including most method parts in all *union slices*. The variable-level *Coverage* metrics also indicate high method cohesion.

B. Limits of Cohesion Metrics and Threats to Validity

Applying the slice-based cohesion metrics on variable-level revealed some limits. For small methods, typically used in the training of novice programmers, implementations will often result in a single union slice per variable that covers the better part of the method. Moreover, primitive input parameters are usually not re-defined, which makes iteratively computed *union slices* less important for those parameters. For non-primitive input parameters, the approach is still worthwhile.

Moreover, threats to validity need to be addressed. First, the uncontrolled openness of the example problem can be seen as a threat to validity. There are various ways to implement a solution, which are not inherently tied to the level of programming skills. However, for this pre-study, this uncontrolled openness is a strength to collect different implementations.

Second, the recruitment of participants was uncontrolled regarding their programming skills. Undergraduate and graduate students of the curricula of *Applied Computer Science* and *Management of Information Systems* have participated in the study. This leads to uncontrolled variance in the students' programming experience. The professional programmers reported a mean programming experience of 17.29 ± 9.62 years. We mitigate this threat by only reporting a selected case study.

A third threat to validity arises from the adaptation to the *Java* slicer *JRazor* [21] to compute the slice-based cohesion metrics, which was not validated elsewhere. To mitigate this threat, we double-checked the reported metric results by hand.

C. Use of the Cohesion Metrics for Education and Training

The outlined IDE-based LA approach facilitates real-time assessment and feedback of program construction sequences. We see different use cases to employ the reported cohesion metrics and visualizations in education and training. First is the direct use by educators, which provides an additional, automatically processed, aspect during the assessment of student-written programs and supports the educators to give in-depth feedback regarding the program construction as a process. This is especially valuable when assessing long methods.

Second is the use of IDE-based *interventions* [9] to augment the information accessible to students during programming, including *variable-level* cohesion reports on demand, or cohesion information in the programming view for each variable.

By incorporating the topic of cohesion in the process of learning to program, students benefit from experiencing program structures that lead to low and high cohesion, respectively. Programming tasks with target cohesion values can be designed so that students practice the creation of highly cohesive programs - starting from program code that is functional but non-cohesive. Such tasks teach students the design principle of *single responsibilities* in a quantifiable manner. Students are thereby empowered to develop a holistic view of programming: not writing single source code statements, but constructing semantically meaningful program parts.

In light of the neo-Piagetian learning process model established by Lister, we suggest that students need to operate on the third, *concrete operational*, stage (being capable of '*a purposeful approach to writing code*' [1]) in order to benefit most from the cohesion information. However, they can thereby be supported in their process of developing expert programming skills towards the last, *formal operational*, stage.

V. Conclusion and Future Work

To better and faster prepare our students for future challenges, we aim to improve the individual assessment and feedback of programming skills by incorporating slice-based cohesion metrics into IDE-based LA research. In this contribution, we report on a pre-study to showcase the use of the slice-based cohesion metric *Coverage* on *variable-level*.

We showcase the metric, investigating two *Java* implementations of an example problem (conversion of a binary input array to decimal and hexadecimal numbers), done by an undergraduate and a professional programmer. Different trains of thought during program construction can be identified by measuring the intersection of all variable union slices. Our data sets and tools can be openly accessed (see Section III-B).

We plan to conduct a systematic evaluation of the relation between programming skills and slice-based cohesion metrics in the future. A tabulated experimental design can be formed by controlling the programming skills to form homogeneous experimental cohorts, while separating implementations by strategic implementation approaches.

VI. Author Profile

Max Kesselbacher is a University Assistant at the Department of Informatics Didactics at the University of Klagenfurt. He completed graduate studies in computer science as a teaching profession as well as computer science with a focus on software engineering and now aims to improve the education of computer science and software engineering in particular. He contributes to informatics workshops, organizes coding contests, and operates a platform to investigate the lasting effects of interventions. In his Ph.D. thesis he focuses on improving the acquisition of programming skills by employing a learning analytics approach, measuring program construction, and making individual support and feedback possible.

Andreas Bollin is a Full Professor at the University of Klagenfurt and head of the Department of Informatics Didactics. He worked on numerous projects dealing with computer science education and different types of new media in education. He authored and co-authored over 70 international peer-reviewed publications, combining informatics didactics, serious games, programming, formal methods, and software engineering. He heads a non-profit center for teaching computing science to the public (the "Informatikwerkstatt") with more than 14.000 attendees in the past four years. The focus of his research includes computing education in primary and secondary education, educational and serious games, computational thinking, competency, maturity models and gender/personality aspects in teaching.

References

[1] R. Lister, "Toward a developmental epistemology of computer programming," in *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, ser. WiPSCE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 5–16. [Online]. Available: https://doi.org/10.1145/2978249.2978251

[2] R. Bornat, S. Dehnadi, and Simon, "Mental models, consistency and programming aptitude," in *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ser. ACE '08. AUS: Australian Computer Society, Inc., 2008, p. 53–61.

[3] A. Luxton-Reilly, "Learning to program is easy," in *ITiCSE '16*. New York, USA: ACM, 2016, pp. 284–289.

[4] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[5] I. Burnstein and F. Saner, "Applying fuzzy reasoning to the program understanding," in *Proceedings of SEKE'98, Tenth International Conference on Software Engineering and Knowledge Engineering*, June 1998, pp. 394–401.

[6] A. Broad and N. Filer, "Applying case-based reasoning to code understanding and generation," in *Proceedings of the Fourth United Kingdom Case-Based Reasoning Workshop (UKCBR4)*, University of Salford, Salford, England, September 1999, pp. 35–48.

[7] A. Bollin, "Metrics for quantifying evolutionary changes in z specifications," *Journal of Software: Evolution and Process*, vol. 25, no. 9, pp. 1027–1059, 2013. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1596

[8] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll, "Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies," *ITiCSE WGR'16*, pp. 41–63, 2015.

[9] C. D. Hundhausen, D. M. Olivares, and A. S. Carter, "IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda," *ACM Trans. Comput. Educ*, vol. 17, no. 26, pp. 1–26, 2017.

[10] L. M. Ott and J. J. Thuss, "Slice based metrics for estimating cohesion," *Proceedings - 1st METRIC 1993*, pp. 71–81, 1993.

[11] J. Krinke, "Statement-level cohesion metrics and their visualization," *SCAM 2007 - Proceedings 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 37–46, 2007.

[12] M. Kesselbacher, K. Wiltschnig, and A. Bollin, "Block-based learning analytics repository and dashboard: Towards an interface between researcher and educator," in *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*, ser. WiPSCE '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3421590.3421662

[13] M. Weiser, "Program slicing," *Proceedings - International Conference on Software Engineering*, pp. 439–449, 1981.

[14] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 1, pp. 1–27, 2007.

[15] K. L. Eranki and K. M. Moudgalya, "Program Slicing Technique : A novel approach to improve programming skills in novice learners," *SIGITE 2016 - Proceedings of the 17th Annual Conference on Information Technology Education*, pp. 160–165, 2016.

[16] A. Bollin, "Slice-based Formal Specification Measures–Mapping Coupling and Cohesion Measures to Formal Z," *NFM 2010 - Second NASA Formal Methods Symposium*, p. 24, 2010.

[17] N. C. Brown, A. Altadmri, S. Sentance, and M. Kölling, "Blackbox, five years on: An evaluation of a large-scale programming data collection project," *Proceedings of the 2018 ACM Conference ICER*, pp. 196–204, 2018.

[18] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller, "Programming Pluralism: Using Learning Analytics to detect patterns in the learning of computer programming," *Journal of the Learning Sciences*, vol. 23, no. 4, pp. 561–599, 2014.

[19] A. S. Carter, C. D. Hundhausen, and O. Adesope, "The Normalized Programming State Model : Predicting Student Performance in Computing Courses Based on Programming Behavior," *ICER '15*, pp. 141–149, 2015.

[20] K. Rivers, E. Harpstead, and K. Koedinger, "Learning Curve Analysis for Programming," in *Proceedings ICER '16*, 2016, pp. 143–151.

[21] A. Rama, "Slicing von objektorientierten java programmen," Master's thesis, University of Klagenfurt, 2013.