6-24-2017

A Learning Trajectory for Developing Computational Thinking and Programming

Sean B. Brophy
Purdue University - Main Campus

Tony A. Lowe
Purdue University

Paper ID #19692

A Learning Trajectory for Developing Computational Thinking and Programming

Dr. Sean P. Brophy, Purdue University, West Lafayette (College of Engineering)

Dr. Sean Brophy is a mechanical engineer, computer scientist and learning scientists. His research in engineering education and learning sciences explores how undergraduate engineering students develop skills in design, troubleshooting and analytical reasoning. He is particularly interested in how these skills develop through students' interaction with technology.

Prof. Tony Andrew Lowe, Purdue University, West Lafayette (College of Engineering)

Tony Lowe is a PhD candidate in Engineering Education at Purdue University. He has a BSEE from Rose-Hulman Institute of Technology and a MSIT from Capella. He currently teaches as an adjunct at CTU Online and has been an on-and-off corporate educator and full time software engineer for twenty years.

A learning trajectory for developing computational thinking and programming

Abstract

This research study identifies the relationship between students' prior experiences with programming and their development of computational thinking and programming during their first year engineering experience. Many first year programs teach students basic programming concepts using languages like MATLAB or LABView. These languages are used because many of the disciplinary schools expect students to use computational models to analyze systems of interest. Some undergraduate engineering students are entering college with strong computational backgrounds, while others have no experience at all. This study is the first in a series to better identify students' transition into developing and reasoning with analytical tools. The learning progression across two programming languages is critical to developing a student's ability to generalize across various computational tools. The goal of this study is to identify how students progress in their ability to engage in computational thinking and programming relative to other learning outcomes for the course. This initial investigation uses students' prior background in programming and their exam scores to evaluate their interdependence of prior knowledge on learning programming across their first semester in university. As anticipated, learning a new language is difficult compared to the other course objectives. However, students with some prior knowledge of programming demonstrate a balanced performance between computational thinking and the other course objectives. Students who have limited programming experience do demonstrate a higher variance in their performance in problems related to computational thinking compared to their other course objectives. One of the leading factors is the time spent practicing programming. This paper will be of interest to instructors with the objective of developing computational thinking and programming in classrooms with a large variance in students' backgrounds with programming.

Introduction

This research study explores the developmental trajectory of first year engineering students' abilities to apply computational thinking (CT) combined with computer science(CS)/programming skills used in engineering problem solving. Engineers engage in a range of engineering problem solving activities associated with design, troubleshooting and analysis [1,2] . A critical part of the problem solving processes is transforming the context and system of interest into a model that can be used to analyze the system's performance. This transformation involves deconstructing the problem into the major factors, and identifying the interdependent relationships between these factors. These models can be operated on to generate knowledge about the properties and behavior of these systems [4]. Many of the models engineers produce for their work are represented with a system of mathematical equations (first principles or empirical), or the models are a system of rules that define the behavior of the system. The challenge is how to think with these models, that is, generate useful knowledge to inform decisions related to the problem at hand. In some cases the engineer may be able to perform mental computation, or hand calculation to determine a solution. However, if the model is complex enough, then they will need a computer to perform the thousands of computations needed to run the model. This process of transforming a model into an algorithm requires computational thinking and understanding how a computer will process the model requires computer programming skills.

In the past decade a strong effort has been underway to emphasize computer science for all students [10]. As mentioned, engineering students need computers to support their analytical problem solving needs. We anticipate all students will need to develop computer programs as part of their analytical work in any of the disciplines they will study during their four year academic engineering career. Further we anticipate that many students will need to know how to use computer based models even if they don't build the models, or they will work with other professionals who are running the models as part of the systems they design in industry or for research. Therefore, an important outcome of a first year engineering course is learning to develop and use computational tools to support analytical reasoning associated with engineering problem solving.

Many first year engineering courses have a desired learning outcome related to computer programming of computational models. The course associated with this study originally only taught MATLAB as an analytic tool for engineering problem solving because the professional schools want students to be able to use it in their courses after their first year experience. The target learning outcomes focused on students' abilities to use basic programming structures to control the flow of a program including sequential (procedural), conditional and looping constructs. Matrices and arrays were taught primarily in the context of data structures for processing a collection of numeric data types. Students were taught the basic methods for reading data into their programs from either a user's console or file. Functions were taught as a methods of generalizing procedures and modularizing code for readability and reuse. They also learned how to use formatted output, or plots, to present data to a console or file. These tools made it possible for students to solve basic numerical models or process large data sets. Typically the matrix operations and symbolic tools in MATLAB were considered too advanced for first year engineering students. Therefore, the main learning goals were to support students' development of scripts to process data or run basic numerical models as part of specific analysis.

A new version of the course has evolved over the years from a one semester to a two semester course and increased its learning objectives to include engineering design and design/control of mechatronic machines. These added objectives supported students' development for managing complex systems, projects and team dynamics. Further, a choice was made to teach multiple programming languages based on learning sciences research which demonstrated how knowledge will generalize when it is taught in multiple contexts [3,9]. This lead to the premise that exposing students to basic constructs of programming across multiple programming languages could lead to an increase in their abilities and confidence to engage in complex problem solving contexts involving computational modeling. This premise lead to identifying two to three programming languages to develop students' programming skills involving interesting engineering contexts to support learning and interest in engineering as a career.

Python was selected as the first programming language for students for several reasons. First, Python can be used for a wide range of applications including numerical modeling, database management and the internet (networks). Python language was designed to be readable and well structured which can be a benefit to new learners. The interpretive language provides a development environment that makes it easy for programmers to experiment with basic instructions as a starting point for investigating how these instructions and built-in functions work. It is cross platform. Data structures can be well defined. Programs can be defined as basic procedural programs to start, and later object oriented programming can be introduced for more advanced management of the code. Modular coding techniques can be easily applied by beginning programmers. The available libraries and the ease of use of these libraries makes it accessible to students. Further many computer science and engineering programs use Python as an introduction to programing for similar reasons [6,8].

MATLAB was introduced next in the sequence of programming languages. The primary reason to include it was because of the professional schools' desire to include it in the tool set of all engineering students. Also, its mathematical capabilities make it an excellent choice for future engineering and scientific modeling. Like Python, MATLAB's interpretive environment makes it easy to experiment with coding scripts. The integrated development environment has a similar interface to other programming environments, or analysis tools (e.g. SPICE, ANSYS) students my need to learn for other courses or as tools they may use in industry.

Finally C is introduced to complement the basic programming skills developed with Python and MATLAB. C offers insights into more advanced computer science concepts such as explicit data typing, pointers and memory allocation. Further, it provides more control of hardware like obtaining data from sensors and regulating motors connected to embedded control systems. Teams also use RobotC with Mindstorm mechatronic systems during their second semester term projects.

Introducing first year engineering students to multiple programming languages may seem daunting. To meet this challenge the instructional model for this course leveraged the benefits of problem based learning, peer instruction and studio learning to provide students with an effective learning environment to learn these languages. This study is part of a multi-phase study associated with developing students ability to construct computational models to support their analysis of natural and engineered systems. For this study the central research questions included:

1. How well do students develop computational thinking and computer programming skills while learning multiple programming languages?
2. What differences in student learning exist between developing computational learning objectives and other course objectives?

Methods

The primary research goal was to identify how students' computational skills develop while learning multiple programming languages. In this context learning may depend on the prior programming experience students brought to this first year experience. For many this could have been a completely new experience and required a new way of thinking about problem solving and analysis. Therefore, the second layer of analysis was to compare their computational and programming performance with the other course objectives. The following outlines the participants and the instructional methods used to develop their programming skills. A brief description of the various instruments used to evaluate students' learning and attitudes toward programming as they developed across multiple years follows.

Participants: The target audience for this study were students enrolled in a first year engineering honors course. The course was conducted across two semester to develop students' abilities to engage in innovative design processes with teams and develop computational skills to produce numerical models for analytical problem solving and control algorithms for autonomous devices.

Instructional approach: The course engaged students in a blended instruction method to develop basic skills and problem based skills to manage more complex contexts. Instruction for computational/ programming skills began with pre-activities. Students were required to review readings and exercises prior to coming to class. In Fall 2014 online video modules were used to introduce major topics to students. In Fall 2015, an online textbook, authored by Zybooks [11], was adopted to organize content into four modules each for Python and MATLAB. Each week students were assigned a module to review prior to coming class. The typical topical sequence included, 1) an introduction to data types, instructions, procedural control and libraries, 2) conditional control and functions, 3) collections and looping control, 4) Input/Output control. The Zybooks text had traditional text narrative to introduce new concepts to students. It also had an added feature of an embedded coding window that provided exercises with feedback so students could test their understanding. Students reviewed the material and then posted questions they still had about the content. These questions were evaluated by the instructional team and the most frequently asked questions were discussed in class.

In class students worked on tasks as a team. Each task consisted of a series of small exercises targeting specific programming concepts presented in the pre-activities. These exercise were designed to practice using various instructions and the syntax associated with performing them. More importantly, the exercises were presented with a context to help illustrate how to translate various problem contexts into algorithms. Each session ended with a short quiz called a check for understanding (CFU). The CFU targeted fundamental concepts relative to the topic for the particular day. The assumption was students would perform well on the CFU if they prepared for the class by interacting with the pre-activities and engaged with their peers to perform the programming tasks.

After the in-class tasks students had two opportunities in the following week to further develop their programming skills. First, students had the option to complete the in-class tasks their team did not complete to receive bonus points. Second, students were required to complete a post activity for homework points. The Post Activity (PA) was presented in either a scientific or engineering context that would benefit from a computational model. Students were required to generate an algorithm in the form of a flowchart that summarize the logic for their computational model and then develop and test code to perform the algorithm.

Instruments: Students' prior experience was determined with a background survey profiling students' prior experience with programming, solid modeling tools, and working in teams. The primary purpose of the background survey was to facilitate team formation. The background information was used to establish a balance of team members based on programming background, ethnicity, gender, and schedule. Students' self reports on prior experience and training were used to formulate a prior programming metric. Survey items included -

A. Indicate your general level of ability when it comes to computer programming.
    1. I cannot program at all -- even programming my calculator is hard work.
    2. I can produce a working program, but it would be very simple or require a long time for me to complete it.
    3. I can create a reasonably complex program in at least one language.
    4. I can program in multiple languages, and I can produce fairly complex programs in at least one language.

B. Indicate how difficult it would be for you to do each of the following programming-related tasks, assuming you had an appropriate reference manual available.

1 Very Difficult; 2 Difficult; 3 Somewhat Difficult;4 Neutral; 5 Somewhat Easy; 6 Easy; 7 Very Easy

a. I could readily write a sequentially-structured program in some programming language.
b. I could readily write a program in some programming language that includes conditional and repetition structures.
c. I could readily write a program in some programming language that includes complex structures (e.g., pointers, struct classes, recursive functions, etc.)

C. For the following computer languages, please make a self-appraisal of your level of knowledge.
1 Very Poor; 2 Poor; 3 Fair; 4 Good; 5 Very Good

1 Basic
2 C
3 C++
4 C#
5 Python 2
6 Python 3
7 Fortran 90
8 Java
9 Pascal
10 Visual Basic for Applications
13 NXT Lego Mindstorm

D. For the following computer languages, indicate whether or not you have taken a formal class

1 Basic
2 C
3 C++ or C#
4 Python (2 or 3)
5 Fortran
6 Java
7 Pascal
8 Visual Basic for Applications
9 Matlab
10 LabVIEW

The scores were combined into a subscore and divided into 4 categories defined as, 1) novice, 2) rudimentary, 3) intermediate, 4) competent. Higher levels of expertise could be defined, but it was assumed none of the students had a level of expertise that would suggest they were accomplished programmers ready to work in industry.

Students' computational thinking and computer science skills (CT/CS) were determined by a subscore from items on the exam targeting the learning objectives. A second subscore is associated with all the other test items that target other learning objectives for the course.

Analysis: A within subjects quantitative analysis was used to compare students' prior programming experience with their ultimate performance in the course. A delta score was computed between the CT/CS and students' self reports on their prior programming abilities. A correlational analysis was performed between the student's cumulative CT/CS subscore and total test scores across the three exams.

Results

Students' computational thinking and computer science scores from the exams were compared for students across Programming Experiences shown in Figure 1 and Figure 2. The dependent variable represents the percentage of total cumulative points that could be earned for exam in each semester.

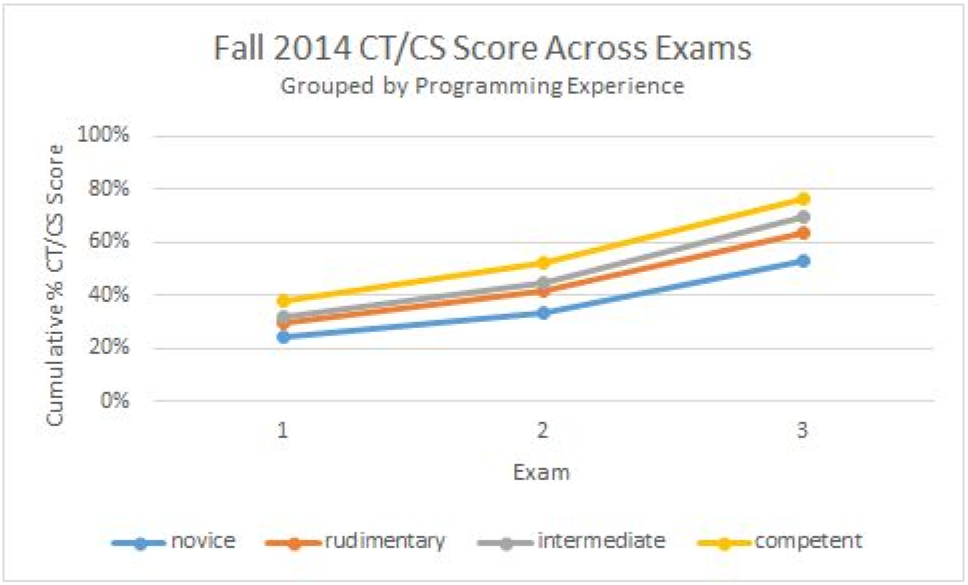

Figure 1 - Cumulative scores across exams for each expertise level of programming

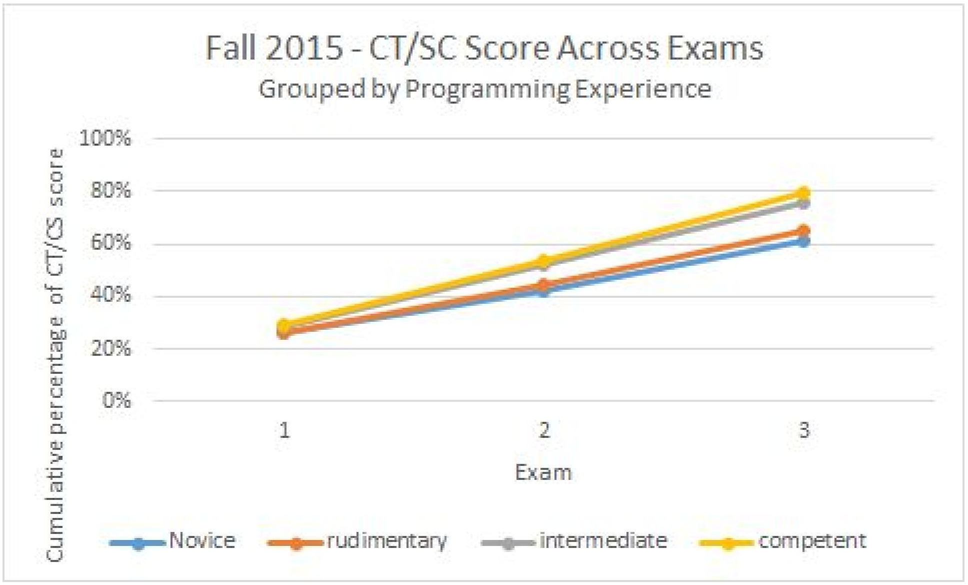

Figure 2 - Cumulative scores of students based on prior experience.

Figures 3 and 4 illustrate a box whisker graph of students' CT/CS performance relative to the other exam questions. The assumption was this factor provided a relative indicator of students' prior experience relative to their overall performance on other common first year learning objectives. The cumulative

CT/CS scores were subtracted from the total exam score and offset by 1.   That is, students who performed similarly on the CT/CS items compared with the other test items would have a delta level of 1. Students scoring better on the CT/CS items would have a positive delta meaning these students actually performed better on CT/CS items compared to other test items. Students who score worse on the CT/CS items would have a delta level less than one. In both years students coming into the course with even an intermediate level of programming competence demonstrated equal abilities across all test items with a relatively low level of variance. Alternatively, novice programmers tended to score lower on the programming portions. However, the variance of scores for novice programmer was larger indicating that lack of prior experience was a strong indicator of future success on exams.

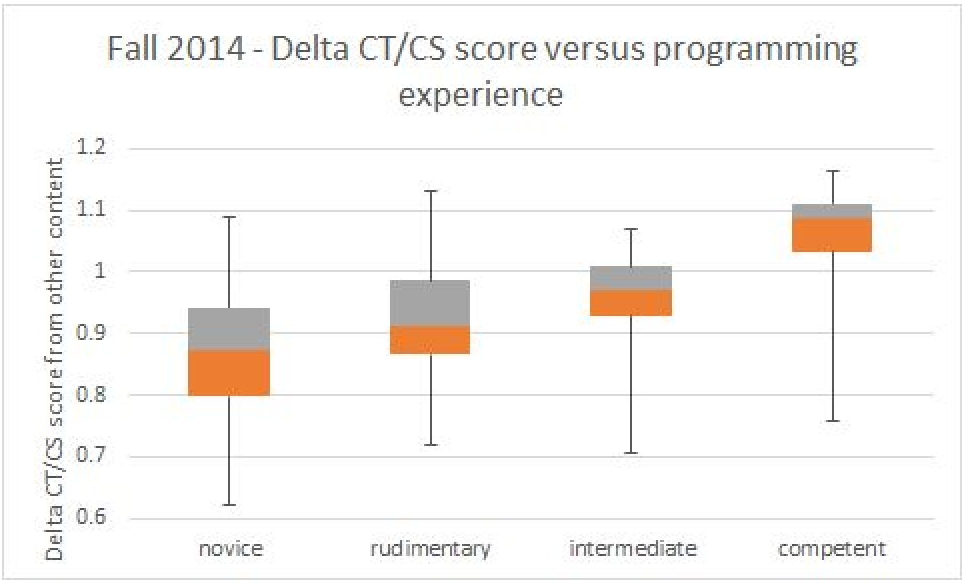

Figure 3 - Difference between CT/CS item scores and other test items relative to prior programming experience.

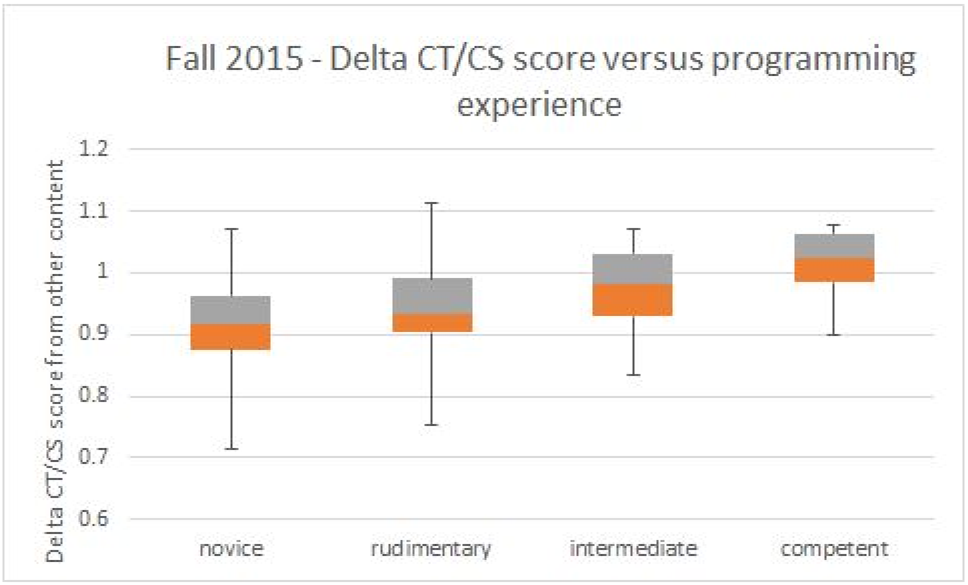

Figure 4 - Difference between CT/CS item scores and other test items relative to prior programming experience.

Discussion

The instructional design of the course should balance students' effort and performance on the major learning outcomes for the course. Learning a new programming language can be very challenging for students who do not have a programming background. Although learning to engage in evidence based design may be unfamiliar to students as well, and potentially just as difficult to learn and teach. One concern the course designers had was introducing these challenging topics to novice engineering students because of the variance in prior background students have with programming and design. Figure 1 and 2 illustrate that for the two years of analysis students were successful in learning the major programming objectives. Even novice programmers eventually obtain nearly 65% of the points associated with computational thinking and programming assessment questions. This lower performance was due to a slow start in performance on programming skills which was often the most challenge concepts assessed on the initial exam. This also explains some of the variance of the novice programmers who demonstrated early their ability to grasp programming concepts and perform well on the exams. Interestingly, the prior experience did have some effect on novice students' performance on the first exam in 2015. The introduction of the new online textbook, with embedded practice session may have had some impact on students' early success with the programming activities. Additional studies are underway to investigate students' perceptions of these resources for learning and the impact of their participation with this resource on their overall programming performance.

Learning to program could be challenging for first year students because it was new and not comparable to other skills that were more in line with the difficulty for achieving other course learning objectives. Figure 3 and 4 helped to illustrate this point for both years. In both years the majority of students were more successful on the general course objectives compared to their programming performance. However, this difference between the overall CS performance compared with the rest of the score was not significant. This suggests successful students achieved a balance across all the course learning objectives. Also, the progression of multiple learning programming language lead to increasing all students' performance on programming tasks.

In the past year several changes were made to the instructional method to provide more in-class practice with feedback. The intent was to provide students with with more formal feedback from the instructional team. This form of active learning was based on models of guided discovery using formative assessment tools like Classroom Presenter [12]. Further training of the Peer Teaching Assistant (PTA) was increased to better support their approach to evaluating student work and generating meaningful feedback on individual student's work product and during office hours. The goal is to reduce the effect from a slow start and to increase the learning curve of novice so they achieve at least the same performance as the competent student.

Future studies are underway to better determine the specific transfer effects of programming multiple language and students' development of computational thinking skills. These impacts include their ability to transform a complex system into a model they can be used to either explain patterns and behaviors of a system they design and develop models they can convert into algorithms a computer can process. These are critical skills first year engineering students need as they prepare for the academic and professional engineering careers.

Acknolwedgements

We appreciate the support of Purdue University's School of Engineering Education and the First Year Engineering Honors Program for their support of this study. The views expressed by the authors do not necessarily reflect the views of these agencies.

References

[1] Brophy, S., Klein, S., Portsmore, M., & Rogers, C. (2008). Advancing engineering education in P‐12 classrooms. *Journal of Engineering Education*, *97*(3), 369-387.

[2] Bennedsen, J., & Caspersen, M. (2008). Model-driven programming. In *Reflections on the Teaching of Programming* (pp. 116-129). Springer Berlin Heidelberg.

[3]Cognition and Technology Group at Vanderbilt. (1997). *The Jasper Project: Lessons in curriculum, instruction, assessment, and professional development*. Psychology Press.

[4] Hmelo-Silver, C. E., & Azevedo, R. (2006). Understanding complex systems: Some core challenges. The Journal of the learning sciences, 15(1), 53-61.

[5] Koulouri, T., Lauria, S., & Macredie, R. D. (2015). Teaching introductory programming: a quantitative evaluation of different approaches. *ACM Transactions on Computing Education (TOCE)*, *14*(4), 26.

[6] Langtangen, H. P. (2009). A primer on scientific programming with Python (Vol. 2): Springer.

[7] Morris, C. D., Bransford, J. D., & Franks, J. J. (1977). Levels of processing versus transfer appropriate processing. Journal of verbal learning and verbal behavior, 16(5), 519-533.

[8] Radenski, A. (2006). Python First: A lab-based digital introduction to computer science. ACM SIGCSE Bulletin, 38(3), 197-201.

[9] Schwartz, D. L., & Bransford, J. D. (1998). A Time For Telling. Cognition and Instruction, 16(4), 475-5223.

[10] Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33-35.

[11] Zybook. (2017). Introduction to Python and MATLAB. Last accessed March 19, 2017, from zybooks.com.

[12] Anderson, R., Davis, P., Linnell, N., Prince, C., Razmo, V., & Videon, F. (2007). Classroom presenter: Enhancing interactive education with digital ink. *Computer*, *40*(9).