26

A Selective Review of Computing Education Research

Lauri Malmi and Aditya Johri

1 Introduction

Whether the goal is to simulate, design, or analyze, computational data and algorithms are core to engineering. Consequently, it is common for all engineering students to take at least one or more courses related to computing during their degree program. For example, programming is required for most engineering majors, and so are courses on the use of computational tools such as Matlab™ for engineering tasks. Given this dependence between engineering and computing, it is important that education scholars in both fields develop a better understanding of the other field to help prepare a stronger future engineering and computing workforce and to be able to conduct new and innovative research. Computing Education Research (CER) is a broad field and has a significant research alignment with EER, including a focus on (engineering) epistemologies, learning mechanisms, learning systems, diversity and inclusiveness, and assessment (Adams et al., 2006). Since we cannot do justice to the entire range of work in CER, this chapter is a selective overview of CER for engineering education scholars. A similar review of EER for computing education researchers with some comparison of research traditions and culture of the fields can be found in Loui and Borrego (2019).

The chapter begins with a brief history of computing education, followed by a discussion of the scope of CER, and the main publication venues and practices in CER. The subsequent section briefly discusses the role and use of theoretical frameworks in CER. Thereafter, we present the two subareas of CER that we selected as particularly relevant from an EER perspective: research on programming education and research on software tools to support computing education. A brief conclusion ends the chapter. Our review focuses on research in tertiary or higher education contexts (refer to Chapter 25, this volume, by Yadav and Lachney (2023) for more information on computing education in K–12 level), mainly in the Western cultural context (United States, Europe, Australasia).

2 Brief History and Overview of Computing Education Research (CER)

Computer science (CS) is a young discipline compared to natural sciences and many fields of engineering. Consequently, computing education is a younger field than engineering education. Computing education emerged in the mid-1950s as computers started being used in industry. This

DOI: 10.4324/9781003287483-31

created a need for specialists, resulting in company-level training programs. In the late 1950s, universities started building educational programs for training computing professionals towards specific jobs but without a shared vision of the profession or of education goals, course requirements, or learning resources needed. In the United States, the Association for Computing Machinery (ACM) set up a curriculum committee in 1968 that was explicitly aimed at advancing academic computing programs in universities (Atchison et al., 1968). The initial recommendation emphasized mathematical approaches, but a decade later, new guidelines put an emphasis on a more practice-based, hands-on approach, including programming and applications (Austing et al., 1979). The purview was further broadened when a new curriculum recommendation, in collaboration with IEEE Computer Society, acknowledged the significance of the social and professional context of computing (Tucker & Barnes, 1990). The increasing breadth of the field resulted in independent recommendations for different subareas of computing after 2000. The Computing Curricula (2005) defined five different subareas: computer engineering, computer science, software engineering, information technology, and information systems. Computing Curricula (2020) further added two more areas, cybersecurity and data science. For simplicity, we use the term "computing" to denote any of these fields. However, it must be noted that the focal areas covered in this chapter belong to computer science (CS), and "computing" in common parlance is used as synonym of it.[1]

Denning et al. (1989) characterized computing as a discipline that combines three tightly intertwined aspects, including theory, abstraction (modelling), and design. Tedre and Sutinen (2008, p. 153) built on this when they discussed the characteristics of computing: "Those aspects rely on three different intellectual traditions (the task force called them paradigms): the mathematical (or analytical, theoretical, or formalist) tradition, the scientific (or empirical) tradition, and the engineering (or technological) tradition." This characterization well demonstrates the analogy between computing and engineering as disciplines, which is naturally reflected in CER and EER too. However, despite these similar approaches, computing should not be considered a subdiscipline of engineering. Computing and computing education have been developed both in engineering schools as well as in non-technical higher education institutes. While EER venues have published numerous CER papers, much of CER takes place and is published in venues and communities which are distinct from EER.

The development of CER as a Discipline-Based Education Research (DBER) field can be traced to several related computing education initiatives (Guzdial & du Boulay, 2019; Tedre et al., 2018). First, formal computing education saw scholarly development in academic institutions with research focused on future computing professionals. Second, for decades, there has been research exploring professional computing, as well as research on how children learn computing in K–12 education. Finally, research in the field of human–computer interaction and related areas such as computer-supported cooperative work has given much focus on computing education. The most significant community focusing solely on CER is ACM Special Interest Group in Computing Education (SIGCSE). However, CER papers are also published in many other venues, such as those focusing on software engineering education, human–computer interaction, educational technology, and e-learning, as well as in numerous educational research journals and, as noted earlier, in many EER venues.

Research on professionals in the field has been ongoing at least since the early 1970s. Weinberg's *Psychology of Programming* (Weinberg, 1971) was published in 1971, and in the 1980s, Soloway carried out seminal research on experts' programming plans (e.g., Soloway & Ehrlich, 1984). Important early venues for presenting such research were the Empirical Studies of Programmers (ESP) conferences and Psychology of Programming Interest Group (PPIG) workshops, which both started in 1986. The ESP conferences ceased in the 1990s, but PPIG is still active.

In the last ten years, computing education in schools has boomed internationally, when many countries have included more computing courses and content in K–12 curricula. Research on children's learning of computing also has a long history. For example, the Logo language was designed in the late 1960s, and Papert's (2020) classic book, *Mindstorms*, which addressed children's learning of programming, turtle graphics, and Logo, was first published in 1980. More elaborative discussions of the history of computing and computing education are available in Denning and Tedre (2019), Guzdial and du Boulay (2019), and Tedre et al. (2018).

2.1 Structure of the Field

Currently, the most comprehensive source for work in CER is *The Cambridge Handbook of Computing Education Research* (Fincher & Robins, 2019), which divides research in the field into the four following topical areas.

Systemic issues are topics which persist in the field, including research on novice or introductory programming, more advanced programming, assessment and plagiarism, various pedagogical approaches, as well as questions on equity and diversity.

New milieux address more recent issues that have arisen with computing's spread beyond the "traditional" university settings (formal classrooms and departments of computing). Such work covers research on computational thinking and computing in schools (K–12), as well as computing for other disciplines and new programming paradigms.[2]

Systems software and technology is an area focusing on how software and hardware tools can support learning, including tangible computing and integrated learning environments.

The final section on *teacher and student knowledge* investigates issues that concern the production and acquisition of computing knowledge, such as teacher knowledge, teacher training and professional development, learning outside classrooms, student knowledge and misconceptions, students' motivation, attitudes and dispositions, as well as students as teachers and communicators.

The Cambridge Handbook of CER (Fincher & Robins, 2019) does not give a canonical definition of CER, nor does it prescribe what should be included in the field. This would be difficult, as the field continuously evolves following the rapid development of computing itself, as well as its ever-growing penetration in society. We could, however, give the following characterization as a draft which widely covers the research topics in the field: *CER investigates complex phenomena related to the teaching and learning of computing, including actors (students, teachers, organizations), curricular content, learning resources and technologies, as well as recruitment and retention in both formal and informal educational settings. CER also investigates research in the field itself by analyzing ongoing or published research and develops domain-specific theories and methods to support the field.*

CER is an interdisciplinary field which draws on methods, theories, and content from several disciplines. It leverages research methods and theoretical frameworks from social sciences, especially from educational sciences and psychology, but also from sociology, anthropology, philosophy, and ethics. It also addresses learning content from all areas of computing and widely applies methods and technologies from computing to develop, apply, and analyze software-based learning resources, tools, and data collected from their use.

2.2 Publication Venues

While CER papers are published in numerous conferences and journals, there are a few venues which focus explicitly on computing education. ACM Special Interest Group of Computer Science Education (SIGCSE) was established after publishing the first ACM computing curriculum in 1968, and it launched its first annual conference, SIGCSE Technical Symposium in 1970. Thereafter, SIGCSE has launched several new conferences, which all have their own profile. In addition, there are a number of other conferences organized by other organizations which have started as regional conferences and grown to international venues later. Table 26.1 lists these venues, followed by a few other significant conferences which also publish CER papers among research from other areas.

Table 26.1 Computing Education–Related Conferences and Journals

Acronym	*Conference Title*	*Since*	*Audience*	*Organized at Locations*
ACM SIGCSE Conferences				
SIGCSE	SIGCSE Technical Symposium	1970	Teachers and researchers	US/Canada
ITiCSE	Innovation and Technology in Computer Science Education	1996	Teachers and researchers	Europe
ICER	International Computing Education Research Conference	2005	Researchers	US/Canada, Europe, Australasia
Non-ACM Conferences with primary focus on computing education (*denotes in-cooperation conferences with proceedings in ACM DL)				
ACE*	Australasian Computing Education Conference	1996	Teachers and researchers	Australasia
Koli Calling*	Koli Calling – International Computing Education Research Conference	2001	Researchers	Finland
WIPCSE*	Workshop in Primary and Secondary Computing Education	2006	Researchers	Europe
ISSEP	International Conference on Informatics in Secondary Schools	2006	Teachers and researchers	Europe
Other conferences that also publish computing education–related research				
ICSE	International Conference on Software Engineering			
CHI	ACM CHI Conference on Human Factors in Computing Systems			
SEFI	European Engineering Education Conference			
ASEE	American Society for Engineering Education Annual Conference			
IEEE-associated conferences that also publish computing education–related research				
FIE	Frontiers in Education			
EDUCON	Global Engineering Education Conference			
TALE	Teaching, Assessment, and Learning in Engineering			
EDUNINE	World Engineering Education Conference			
Journals publishing computing education–related research				
ACM Transactions on Computing Education (TOCE)				
Computer Science Education				
IEEE Transactions on Education				
IEEE Transactions on Learning Technologies				
Computers & Education				
Journal of Educational Computing Research				
Journal of Information Technology Education				

Two journals, *ACM Transactions on Computing Education* and *Computer Science Education*, focus solely on publishing work in CER, while there are many others which publish CER papers in addition to other education-related papers. We list some of the more commonly known ones in Table 26.1, but there are many others.

It should be noted that CER-focused conferences follow the publication tradition of computing sciences, where conference papers are considered almost equally valuable scientific contributions as journal papers. Contributions are submitted as full papers and undergo a rigorous review process (for an overview, see Petre et al. (2020)). In the past few years, publishing CER papers has become more competitive, resulting in lower acceptance rates.

3 Theoretical Frameworks in Computing Education Research

As CER has grown as a field, the use of theories from social sciences in research has received considerable attention (Malmi et al., 2014; Lishinski et al., 2016; Szabo et al., 2019; Szabo & Sheard, 2023; Malmi et al., 2020), along with domain-specific theoretical developments within CER itself (Malmi et al., 2019, 2023). The interest reflects the maturation of CER as a DBER field as methodological rigor and use of theoretical contributions come to be valued more (Tenenberg & Malmi, 2022). Also, see Chapter 7, this volume, by Goncher et al. (2023) for more on the use of theory and theoretical frameworks.

Currently, there are no dominant theoretical frameworks within the field; rather, there is a richness of theories that have been adopted. A recent survey by Szabo et al. (2019) extensively reviewed CER papers in the entire ACM digital library, looking for references to a predefined list of 75 learning theories. The top 10 most cited theories were Csikszentmihalyi's flow, learning styles, mental models, self-efficacy theory, progression of early computational thinking, constructivism, problem-based learning, metacognition, mindsets, and communities of practice. These covered roughly two-thirds of the over 15,000 citations found in the library to some learning theory in their list; each of these theories were cited in 4–13% of the total pool of citations. The other 65 theories covered the remaining one-third of citations.

Malmi et al. (2023) explored the development of new domain-specific theoretical constructs that address learning processes, studying, or learning performance in computing. They identified 85 constructs first published in three major CER venues during 2005–2020. Further investigation into how these constructs had been used in papers citing the original paper, however, revealed that only a small fraction of the citing papers actually used the construct to guide further research or developed the construct further. It seems that CER is still in an early phase of developing its own domain-specific theory base. Another common finding in this work was that the terms *theory*, *model*, and *theoretical framework* are often used quite loosely in CER literature (Malmi et al., 2014). Tedre and Pajunen (2023) discuss this rather in depth in their paper. It is also worth noting that in computer science, the concept *theory* is often associated with mathematical theorems and logical proofs that are heavily used in theoretical computer science, algorithms research, and machine learning research, a very different interpretation from theories in social sciences.

In what follows, we give a few examples of how some theoretical frameworks have been applied within CER to address challenges in learning computing. We encourage the reader to look at the original references for more details.

Research of human cognition states that humans' *working memory* capacity is highly limited and it is impossible to address more than a few items at the same time. *Schema theory*, which we discuss more later, gives one explanation for differences between novice and expert programmers; experts, when compared to novices, employ a wide variety of schemas that they can recognize and apply in different abstraction levels. *Cognitive load theory*, a commonly used theory in CER (Duran et al., 2022), can be used to build on this to explain why learning programming is difficult. For novices, programming tasks have many tightly connected topics (e.g., syntax, language constructs and how they are combined to build an algorithm to implement some goal, underlying notional machine, programming tools that must be used) which together cause high *intrinsic load* for the student that is difficult to reduce. On the other hand, *extrinsic load* (concerning presentation of the task and environment) can be reduced. Mayer's *cognitive theory for multimedia learning* (Mayer & Mayer, 2005) provides many guidelines for this when designing learning resources. These guidelines have frequently been applied when developing learning resources for computing.

Motivation theories have received substantial interest in CER, especially in programming education research. For instance, *self-efficacy* has been widely used as a factor associated with programming success and retention (Lishinski & Yadav, 2019). Kinnunen and Simon (2011) carried out a semester-long study in an introductory programming course finding how self-efficacy and emotional reactions varied during the course. Task difficulty and *goal orientation* may moderate these changes. Specific instruments have been developed to measure self-efficacy in computing contexts (Ramalingan & Wiedenbeck, 1998; Danielsiek et al., 2017). *Metacognition* and *self-regulation theories* have been used to explore students' metacognitive strategies (Falkner et al., 2014) for identifying positive associations with programming ability (Bergin & Reilly, 2005). See Loksa et al. (2022) for many other examples.

Bloom and *Solo taxonomies*, which describe and classify knowledge and skills, have been used in research seeking to evaluate programming task complexity. The difficulty of tasks depends on participants' previous knowledge as well as the programming language used. Thus, more elaborate models of knowledge representation have been developed (e.g., Duran et al., 2018). Lister and Teague applied the *Neo-Piagetian framework* to analyze students' progress in programming, thus giving explanations for observations they had made in a longitudinal think-aloud study of novice programmers (Teague & Lister, 2014).

For interested readers, we recommend exploring more examples and references for applying theoretical frameworks in the chapters in *Cambridge Handbook* that discuss theories from learning sciences (Margulieux et al., 2019), cognitive sciences (Robins et al., 2019), as well as motivation, attitudes, and dispositions (Lishinski & Yadav, 2019).

4 Review of Selected Subareas in CER

As discussed in Section 3.1, CER covers a broad selection of topics. Most of these areas have also been addressed in EER. The goal of this chapter is not to contrast the results in CER and EER. Rather, we wish to highlight two characteristic areas of CER which have a long trajectory within it while they are less discussed in EER contexts, though they are relevant there as well. First, *research on programming* knowledge and skills is at the core of CER, and programming, at least in the introductory level, is essential in engineering programs. Second, we will discuss *tools research*, which gives CER a specific profile among other DBER fields, as many CER practitioners develop their own tools (naturally domain-specific educational software, such as simulation tools, are developed and used in other fields as well).

4.1 Research on Programming in CER

CER has, from its inception, had a strong interest in programming education, especially in the introductory programming courses. More advanced aspects of software development education are addressed in research as well, but not to the same extent. The same applies to other topics in computing, such as data structures and algorithms, databases, computing security, human–computing interaction, computing graphics, data mining, machine learning, or artificial intelligence. An obvious reason for this biased emphasis is that novice programming courses are often large in the number of enrolled students, and many students struggle to pass them or drop out. Another reason is that understanding programming, software development, and acquiring good programming skills are core competencies needed for further studies in the field as well as professional work regardless of whether one is directly involved in software projects.

The role of programming for computing studies could be comparable to the role of calculus in engineering studies. While, in professional life, engineers generally use tools, such as Matlab, to carry out computing, understanding the underlying mathematics is still needed.

First-year introductory programming courses are often labeled as Computer Science 1 or 101 and 2 or 201 (CS1/CS2), or some variation thereof. However, there is no common agreement on what these courses should cover. As the field has developed, "advances in the field have led to an even more diverse set of approaches in introductory courses than the models set out in Computing Curriculum, 2001" (ACM/IEEE-CS, 2013). The body of research work on introductory programming is extensive; therefore, we narrowed our scope to rely on two recent review papers. Robins (2019) presents an extensive summary of various focal areas in research on introductory programming and looks at the history of the research from the 1970s until the current day; Luxton-Reilly et al. (2018) present a systematic literature survey in introductory programming education, identifying 1,666 papers addressing this area published in numerous journals and conferences between 2003 and 2017. In the following, we discuss some of the main challenges identified in research. The list is not comprehensive, and we recommend readers to search for more results from the aforementioned survey papers and the cited original research papers.

4.1.1 Variation in Learning Outcomes

For decades, it has been widely agreed that learning programming is difficult, which is demonstrated by high failure and drop-out rates in introductory courses. In a survey by Bennedsen and Caspersen (2007), data was collected from 63 institutions internationally, including numbers of enrolled students who withdraw from the course, skip the final exam, or sit and fail. The results showed an average failure rate of roughly 33% and often up to 50% or higher. Subsequently, Watson and Li (2014) surveyed literature relating to 51 institutions and reported very similar results. In 2019, a new study was carried out, which compared pass rates in introductory courses in STEM disciplines, including computing (Simon et al., 2019). In their findings, the pass rates in computing were, on average, 75%, which was on the low end of this comparison, but not alarmingly low. The results of these studies must be interpreted with some caution due to issues with data and large institutional differences; however, the low pass rate of students taking programming is a consistent finding.

While low pass rates are a serious concern, there are other equally relevant challenges. There is much evidence that learning among students who pass basic courses is weak, that is, it is relatively short-term or conceptually deficient. This was established in the 1980s (Soloway et al., 1983; Kurland et al., 1989) and reaffirmed by McCracken et al. (2001) when their large multinational study demonstrated serious deficiencies in students' ability to solve common problems that students in any type of CS program should hypothetically be able to solve. The study was repeated ten years later, giving better support for students (i.e., a test harness they could use to check their code correctness), finding both poorly and well-performing student groups (Utting et al., 2013). In another large multinational study, Lister et al. (2004) investigated students' tracing skills, that is, how well students could read code and trace its execution, and they found much evidence of weak and fragile learning.

A common finding is that in many programming courses, the results are not normally distributed. In addition to low-performing students, typical CS1 courses have high rates of very well-performing students, and the grade distribution can have two clearly separate peaks, often called "bimodal" distribution (Robins, 2019). This complicates the picture and challenges the simple conclusion that programming is just difficult to learn (Kölling, 2009). Some explanations given to this phenomenon include the cumulative nature of the learning content in CS1. If one fails to learn enough in the early weeks of the course, one faces more and more troubles later because the whole course is building on the previously covered content in the course (Robins, 2010). Moreover, Höök and Eckerdal (2015) presented a finding that students who failed an exam had studied considerably less time at the computer than did well performing students, and despite their interest in the topic, they failed to learn programming by focusing on lectures and reading the course textbook. Pedagogical approaches that build on weekly compulsory exercises with adequate feedback and support can address this challenge.

On the other hand, the whole finding of bimodality has also been questioned by carrying out a rigorous analysis of grade distributions and noting that the claim may reflect instructors' confirmation bias and beliefs of their students (Patitsas et al., 2019).

4.1.2 Difficulty Due to Complex Learning Goals

Du Boulay (1986) summarized the main challenges to learning programming as follows. First, there is *orientation*, what programming is for and what kind of problems can be addressed with this skill. Second, programs are abstract entities; to understand program execution, one must understand the underlying *notional machine*, that is, an abstract description of program execution (Sorva, 2013). Third, programs are written in formal languages with strict syntax and semantics, which regulate what kind of programs are syntactically correct and what operations are acceptable. Fourth, programming proficiency requires developing knowledge of and applying a large number of programming *schemas*, that is, standard methods for implementing common goals. Finally, programming tasks must be carried out with tools (compilers, interpreters, integrated development environments, debuggers, etc.). Though introductory courses largely focus on the used programming language, the other topics listed previously cannot be ignored or fully isolated, which increased the complexity of learning for students. Cognitive load theory (Sweller et al., 1998; Sweller, 2010) provides an explanation for these challenges as *high element interactivity*, the extent to which the task involves interacting elements that must be held in working memory simultaneously. The situation thus causes high intrinsic load for students, and it is difficult to reduce this load.

Robins et al. (2003) reviewed programming education literature and summarized the following learning goals and challenges in programming. Students need to acquire *knowledge* of programming language and tools, they need to learn *strategies* on how to apply this knowledge appropriately, and they need to construct and compare *mental models* of program state, that is, what happens "under the hood" when a program is executed. Moreover, all these aspects are relevant in three phases, *designing* the program, *implementing* it, and finally, *evaluating* the result, which covers testing and debugging the program. Achieving these goals is not a minor task for students.

Whereas most papers have historically considered the challenge of teaching programming from a teacher's point of view, researchers have also investigated students' perspectives and what they have perceived as difficulties. Compared with communicating with a natural language, which is flexible for presenting things, programs are written in formal languages, which have a strict syntax. Indeed, syntax errors are frequently reported as challenges for students (e.g., Robins et al., 2006). Moreover, it is challenging for students to write programs that are without error or ambiguity, and students often have difficulties understanding the task, designing the program structure, and using some language constructs, such as loops and arrays (Robins et al., 2006). In a large-scale study involving six European universities, students reported that the most difficult aspects of programming were understanding how to design a program to solve a certain task, how to divide functionality into procedures, and how to find bugs in their own programs (Lahtinen et al., 2005).

4.1.3 Novice and Expert Knowledge Differences

Soloway and Spohrer (1989) explored the differences between novice and expert programmers. Their findings indicated that novices had deficiencies in understanding some key language constructs, such as loops, arrays, and recursion. Winslow (1996) found that novices were often limited to surface and superficially organized knowledge. They lacked adequate programming schemas and mental models, and they considered programs line by line instead of focusing on larger meaningful structures. On the other hand, programming experts have a large selection of schemas (also called chunks, plans, or scripts) which organize knowledge. For example, there are schemas that store data into an array, schemas that browse array content, and schemas that find the largest item in the array. When reading, tracing, and writing code, experts can operate with these schemas, which can be very complex, while novices struggle with low-level language constructs and how to assemble or put things together. Thus, learning programming could be described as a process of creating, applying, modifying, combining, and evaluating schemas (Rist, 2004). Furthermore, expert programmers can operate with schemas in very different abstraction levels; for example, when managing bit operations in low-level programming, managing multidimensional arrays, implementing graph algorithms, finding and using various available library functions, designing meaningful class structures, or selecting appropriate software architecture models. For novices, it takes years of learning and practice to build such a selection of multilevel schemas which they could use efficiently. An additional challenge is that schemas are often language-dependent because different programming languages provide different constructs for implementing similar goals. Thus, knowledge of schemas learned with one language does not necessarily transfer when learning a new language (Kao et al., 2022).

Closely related to schemas are *mental models*, a concept adopted from cognitive science and widely used in CER. Mental models are personal internal models of how something works. Greca and Moreira (2000, p. 5) contrast them with teachers' *conceptual models*, as follows:

> [C]onceptual models are precise and complete representations that are coherent with scientifically accepted knowledge. That is, whereas mental models are internal, personal, idiosyncratic, incomplete, unstable and essentially functional, conceptual models are external representations that are shared by a given community, and have their coherence with the scientific knowledge of that community. These external representations can materialize as mathematical formulations, analogies, or as material artifacts.

Viable mental models can be useful, as they provide means for explaining and predicting interaction of subjects with the world. However, mental models are implicit, incomplete, imprecise, and sometimes inconsistent with conceptual models. Not all mental models are viable, which can challenge novice students trying to comprehend program execution. Ma et al. (2007) explored the viability of students' mental models in a CS1 course in Java and found that one-third of students had non-viable mental models of value assignment and only one-sixth had a viable mental model of reference assignment. Considering how central these constructs are in programming, it is understandable how novices struggle in tracing program execution.

Ben-Ari (2001, pp. 56, 60), in his critique of constructivism in computer science education, argued that "a model of a computer . . . must be explicitly taught and discussed, not left to haphazard construction and not glossed over with facile analogies," because "novice computer science students have no effective model of a computer," and "the computer forms an accessible ontological reality." Program execution can be taught with the help of a *notional machine* that is an abstract conception of how software and hardware are working during program execution. Fincher et al. (2020, p. 22) define *notional machine* (NM) as follows:

> An NM has a pedagogical purpose, its generic function is to draw attention to, or make salient, some hidden aspect of programs or computing. It will have a specific focus within programs or computing, and will adopt a particular representation that highlights specific aspects of the focus.

Notional machines can be presented at different abstraction levels, and they are language-dependent (Sorva, 2013). In teaching, they are often presented with visualizations, which abstract away details and allow students to grasp the dynamic process and how data is presented and manipulated in the computer memory.

4.1.4 Challenges in Development of Programming Skills

A highly important aspect of programming is that it is a dynamic process. Basic knowledge of language constructs and schemas is not sufficient. Conceptual knowledge ("knowing what") goes hand in hand with practical knowledge, that is, strategies and skills to apply it ("knowing how"). Practices include, for example, utilizing problem-solving strategies, using patterns or analogies in design, evaluating the impact of the structure of the program, implementing or designing algorithms, understanding the pros and cons of different data structures and algorithms, selecting programming tools, and applying testing and debugging strategies. Experts can apply a wide selection of strategies, while novices use only a small set of rudimentary strategies (Robins et al., 2003). Eckerdal (2009) summarizes that concepts and practices are equally important parts of learning goals, and they are equally difficult for students to learn.

Perkins et al. (1989) studied novices' programming process and found three types of behavior groups: stoppers, movers, and tinkerers. Stoppers simply stopped and gave up when facing difficulties or lacking clear directions on how to proceed. Movers, on the other hand, kept trying, experimenting by modifying their code to try to find ways forward. Tinkerers also modified code frequently, but without understanding it, working more or less randomly. These results reflect students' insufficient pool of strategies for addressing challenges they face in programming.

It is important to understand that reading and writing code are separate, though related, skills. Reading and tracing skills (on code execution) are prerequisite for code writing (Xie et al., 2019). If they are not mastered well enough, students' coding process may end up in endless tinkering. Students write code, but when they do not fully understand how it works, they may try random changes in the code when facing difficulties in getting it to work correctly. Moreover, code-reading skills are essential in later phases of study and in professional work in the context of code reviews and code maintenance.

How students learn code reading and tracing is not yet well understood. Luxton-Reilly et al. (2018, p. 60) suggest that "[g]iven the evidence that has been found for the value of code-reading skills in novice programmers, there is an ongoing need to explore further ways of encouraging the development of these skills."

4.1.5 Factors Influencing Students' Learning

Researchers have sought to find out which student-related factors influence students' learning. For decades, there have been observations that some professional programmers are much more productive than others. This has led to assumptions that programming skill is an innate characteristic for some people, a claim for which later work has found little support. There has been a lot of research which has tried to identify factors predicting academic success, including success in learning programming (Hellas et al., 2018). However, the results concerning programming are inconclusive. Robins (2019, p. 349) summarizes, "In short, no factor or combination of factors which clearly predict success in learning a first programming language has been found."

Despite this, there is considerable work investigating the role of various psychological factors associated with success in learning programming (Lishinski & Yadav, 2019; Malmi et al., 2020), in line with similar research that examines learning in other domains. Much work has addressed students' *self-efficacy* in programming contexts, finding out that higher levels of self-efficacy are associated with greater student performance in computing, which is analogous to findings in other STEM fields (Lishinski & Yadav, 2019). Self-efficacy varies during the first programming course due to emotional reactions to work done (Kinnunen & Simon, 2011). Another relevant factor is Dweck's notion of *mindset*, one's belief whether one can grow and develop. Some students risk developing a fixed mindset, thus failing to believe that they can learn programming well (Murphy & Thomas, 2008). *Engagement* and *self-regulated learning* have also received considerable attention in research literature, and there are numerous attempts to explore the impact of various pedagogical interventions on student engagement (Luxton-Reilly et al., 2018, p. 63; Loksa et al., 2022).

While the aforementioned factors are theory-based, substantial research has focused on analyzing students' programming processes, which can be carried out by analyzing log data from programming environments and automatic assessment systems, even at the keystroke level. Much of this work falls under *learning analytics* and *educational data mining* (Grover & Korhonen, 2017). There are some large-scale data repositories (e.g., Brown et al., 2018) which have supported a generation of statistical models on programming behavior. Much of this analysis has focused on seeking to identify students at risk of dropping out, and some results indicate that behavior-based analysis is better in this sense than test-based analysis (Watson et al., 2014).

4.1.6 Teaching Approaches in Programming

Finally, another area that should be briefly mentioned here is teaching methods and approaches. For the interested reader, we recommend the reviews by Falkner and Sheard (2019) and Luxton-Reilly et al. (2018, Section 6). Two unique and relevant CER-related approaches are discussed here. *Pair programming* is a technique frequently used in professional programming, where two people work together on the same program. One person (driver) writes and edits the code, while the other one (navigator or observer) reviews the written code. Roles are switched regularly. Empirical research has reported that pair programming provides students with better support to produce a higher quality of work and improves pass rates among students with low academic performance. It also improves student enjoyment, though conflicts may still appear. For more-experienced students, there is less evidence of benefits. Forming the pairs with the right balance of skills is important (Luxton-Reilly et al., 2018; Umapathy & Ritzhaupt, 2017). *Peer reviewing* of code, when carried out in collaborative settings, also shows clear benefits in identifying errors and discussing higher-level design and implementation issues (Hundhausen et al., 2009).

Media computation (Guzdial, 2003, 2016) is an approach targeted initially to non-computer science majors to set up a motivating application context where computation skills also increase creativity. Media, including text, images, sound, and videos, is now digital data and thus provides ample possibilities for meaningful programming tasks. Empirical evaluation studies have found evidence that media computation increases retention and increases the sense of relevance of programming studies (Guzdial, 2013).

4.2 Research and Development of Tools and Software-Based Learning Resources in CER

Since the inception of the CER field, many computing educators have developed their own software tools to support their own teaching and their students' learning. This line of work continues to this day, and it is a fairly distinctive characteristic of computing education and CER, given the sheer volume of such software.[3] Valentine (2004) analyzed 20 years of papers (1984–2003) addressing CS1/CS2 courses in SIGCSE Technical symposium and found that, out of the 444 papers in this pool, 22% focused on various self-developed tools to support education. A broader review by Luxton-Reilly et al. (2018) identified over 250 papers addressing tools for introductory programming education with a growing trend towards tool development. Overall, research on tools is a mix of educational development, software development, and empirical research. Even though many papers in this area lack rigorous empirical analysis, it is important to examine tools research as this work sheds light on the CER field more broadly. Furthermore, several of the tool categories presented in what follows address analogical goals which are present in engineering education too. We discuss this more at the end of the section.

The use of tools in computing education can be explained by two forces: (1) adequate software development skills among computing researchers and faculty to develop educational software and (2) easy access to skilled expertise in the form of students who support development through capstone projects, summer internships, or thesis work. While educational software is certainly developed in other fields as well, teachers in those fields are likely to have less options for developing their own tailored software. Although the development and use of tools is common, most tools are used only in the context where they were originally developed, and very few have gained a wide international dissemination, when we exclude commercial tools.[4] Some notable exceptions[5] include BlueJ for learning object-oriented programming in Java, Python Tutor for interactive visualization of program execution for multiple programming languages, JFLAP for learning theoretical computer science topics, and Web-CAT for automatic assessment of programming exercises (Edwards & Perez-Quinones, 2008).

Tools can be categorized broadly based on their technical underpinnings or their functional use. From a technical standpoint, there are tools that are *software applications*, which can be downloaded and installed, such as BlueJ and Web-CAT. Then, there are a number of tools that provide some *service at a specific website*, such as Python tutor or PeerWise[6] (Denny et al., 2008), where students can generate multiple-choice questions for other students and respond to and rate the available questions. In addition to these popular services, there are hundreds of small interactive applications that demonstrate the workings of individual concepts, such as a particular data structure or sorting algorithm, which can be found online. A third category of tools is *software frameworks*, which support building smart, interactive learning content (Brusilovsky et al., 2014). An example is the Jsvee framework (Sirkiä, 2016) that enables building program visualizations of the execution of Python programs. Another one is jsParsons (Helminen et al., 2012), a tool for building Parsons problems, also called *programming puzzles*. In this learning activity, textual program code is split into sentence-level visual blocks which are given to the student in random order, and the student should drag and drop the blocks through a direct manipulation interface into the right order to generate a working program. Finally, even *programming languages* can be considered tools; Logo (Solomon et al., 2020) and Pascal (Wirth, 1971), for instance, were initially designed to be simple programming languages targeted at children or novice programmers. Modern block-based languages, such as Scratch™ and App Inventor™, have a similar goal.

4.2.1 Functional Categories of Tools

From a functional perspective, *educational programming environments* include tools specifically tailored for programming education, which aim for simplification by excluding most of the complexities in professional tools. Specific environments have been developed for many programming languages, mostly for C, C++, Java, and Python (see Luxton-Reilly et al., 2018, p. 74). A comparable group of tools includes various *libraries* and *application programming interfaces (APIs)*, which have been developed to simplify complexities of professional languages. For instance, implementing graphical user interfaces with Java using basic libraries, such as Swing, is a fairly complex process; therefore, many simplified libraries focusing on basic operations have been developed for programming courses (e.g., DoodlePad, Squint for Java, and cs1graphics for Python).[7] Finally, a natural part of programming education is learning to use professional development tools; typical examples of these are integrated development environments, such as Eclipse, IntelliJ IDEA, and Visual Studio, as well as basic command-level tools, such as gcc or javac.[8] However, there are no conclusive results on whether students learn better in a professional or educational environment (Luxton-Reilly et al., 2018).

The second category of tools based on functionality includes *visualization tools* that demystify the software and help the user understand the structure of complex software and the execution process, which are inherently abstract and invisible for the user. They are tools that can be used to teach and learn notional machines for a specific programming language.

There are two kinds of visualization tools: *program visualization tools* and *algorithm visualization tools*. Program visualization tools, for example, Python Tutor, visualize how code execution proceeds step by step or in larger steps and how the memory content, that is, states of variables, program's runtime stack, and heap change during the execution (see Sorva et al., 2013). Algorithm visualization tools visualize execution on a more abstract level and typically show dynamic visualizations of data structures that a specific algorithm is manipulating; for example, demonstrating how a sorting algorithm switches contents of an array or how a search algorithm proceeds in a binary search tree (see Shaffer et al., 2010). An important aspect of both program and algorithm visualization is how students interact with the learning content. In a meta-analysis of empirical research on algorithm visualization, Hundhausen et al. (2002) found that students who worked actively with visualization – to edit, change, adapt it – learned better compared with those who just viewed it. Overall, visualization tools respond to the need for simplifying and making the learning of programming more accessible for students developing this skill.

A wide category of tools focuses on *automatic assessment or feedback* on students' academic work. Such tools have been extensively used to address challenges in programming education. For reviews, see Ala-Mutka (2005), Ihantola et al. (2010), Keuning et al. (2018), Lajis et al. (2018), and Paiva et al. (2022). Introductory programming courses (not only MOOCs[9]) are very large in many institutes, ranging from hundreds to even thousands of enrolled students. Automatic assessment tools can be used to address a very large share of the work needed for assessment and giving formative or summative feedback on students' solutions to programming exercises. This enables teachers to give more guidance to students when they solve the exercises or do not understand or disagree with the automatic feedback.

Automatic assessment tools are most often used to check program correctness, that is, whether a student's program passes test cases defined by the teacher. However, some tools have also been developed to give feedback on programming style, program structure, use of specific programming language constructs, program runtime efficiency, or how well the program has been tested. Some tools work on a more abstract level, giving feedback on algorithmic tasks (Malmi et al., 2004). Automatic and human assessment can also be combined for better support of learning (Ala-Mutka et al., 2004; Novak et al., 2019); for example, the system can check that a student's program passes given requirements before it is forwarded to the teacher for closer evaluation.

The previous examples emphasized the teacher's point of view of using tools. From students' point of view, this group of tools provides several benefits: (1) the systems allow students to revise their solution based on the feedback and resubmit their work several times; (2) the feedback from the tool is available immediately anywhere they are working with an Internet connection; and (3) the feedback is available anytime 24/7, unless the teachers wish to limit the time. Some tools provide more specific advantages, such as enhanced error messages. A well-known problem for novice programmers is struggling with programming language syntax, and a part of the problem is that compiler error messages are not always easy to comprehend. Thus, some tools provide students tailored *enhanced compiler error messages*. Despite the seemingly obvious advantages, the results for improved student performance are still inconclusive (Pettit et al., 2017). *Drill-and-practice systems* allow students to train their skills with a systematically ordered set of exercises or lessons. CodeWorkout and CodeWrite are examples of such systems; the latter one includes an additional feature that students can themselves create new exercises with test cases, and the new exercise can be rated by other students (Edwards & Murali, 2017; Denny et al., 2011). *Intelligent tutoring systems (ITS)* provide additional context and adaptive feedback for students during the programming process. They can, for example, integrate additional learning resources, for example, videos and exercises, in a structured order that matches learning objectives and support problem-solving (Gross & Pinkwart, 2015), or they can track student progress and adapt their feedback accordingly (Pullan et al., 2013). Modern trends include integrating AI techniques into the programming system to provide improved hints for students (e.g., Rivers & Koedinger, 2017). Recent advances in AI technologies can even solve typical CS1 course programming exercises, which creates new challenges for programming pedagogy (Finnie-Ansley et al., 2022).

One specific aspect of tools research concerns *e-books*, that is, interactive online books, which integrate a mix of smart learning content with static content, such as text, images, and videos. In this context, the interactive components could be automatically assessed exercises, code visualization elements, interactive code execution demonstrations where students can modify the code, algorithm simulation exercises, etc. Examples of such learning resources are the CS principles book[10] for learning programming (Ericson et al., 2016) and OpenDSA[11] (Fouh et al., 2014) for learning data structures and algorithms. While these resources are highly valuable in education, building and maintaining such resources is more complex than authoring traditional printed textbooks or static online books.

For more information of tools for the interested reader, we recommend Luxton-Reilly et al. (2018, Section 6.4) and Malmi et al. (2019).

4.2.2 Challenges in Tools Research and Implementation

While tools can greatly support both students' and teachers' work, there are also significant challenges involved in developing and using them. Developing new learning resources which heavily employ smart learning content requires resource-intensive software engineering practices (Haaranen et al., 2020). In particular, sustained use of any software requires maintenance and persistent updates to the technology environment, which include installing new hardware, new versions of operating systems or other system software, bug fixes, or improving data and communication security, as well as keeping the systems compatible with other in-house or external systems. A second challenge is reliability of services. For example, if the automatic assessment system breaks for any reason during the weekend or just before submission deadlines, it is likely difficult to organize support at a short notice. The alternative is to use commercial providers, which includes costs and reduced opportunities for tailored system development. It is also worth recognizing that, if the teacher has invested significant effort into developing software-based learning resources based on products from a specific vendor, there exists a risk of becoming too dependent on the service. Vendor lock-in implies that work cannot be transferred to another system without significant new effort if the service shuts down at a short notice or the vendor's cost policy changes, making costs become too high.

4.2.3 Perspectives for Tools Research in Engineering Education

The previous presentation of tools research included several categories where analogical tools exist or can be developed for engineering education. Automatic assessment tools can be used naturally in many cases where engineering students submit program code. On the other hand, when exercises deal with mathematical expressions, automatic assessment tools developed for mathematics education, for example, Stack[12] or Numbas,[13] may apply. Regardless of tools, there are similar types of benefits and challenges for teachers, and engineering education may benefit from pedagogical results of these kind of tools in CER literature.

Program and algorithm visualization tools are used for visualizing dynamic processes, and there has been substantial work to investigate their pedagogical use cases. Dynamic processes and simulations are present in many areas of engineering. Despite the different visual presentations, the pedagogical findings may apply across both fields, and they may learn from each other. Intelligent tutoring systems can naturally be developed for engineering education contexts, and the technologies on which they are built are applicable and can be adapted to present and analyze engineering knowledge. Finally, within computing education research, many technological solutions have been built for interactive e-books, such that can be applied for building e-books in engineering domains.

5 Conclusion

In this chapter, we provided a selective overview of CER, highlighting a few unique and important aspects of the work in the field, including programming, tools, and environments. Through this review, we want to present a window into the growing field of CER and hope that this chapter will appeal to newcomers in both CER and EER, as well as allow those who are familiar with either (or both) of the fields to find common ground for future research and scholarship. We acknowledge that in a single chapter we can cover only a small share of work in CER, and therefore, we have given many references to review papers and other relevant research as examples of the work carried out in CER. We encourage readers to familiarize themselves with the examples in the original reviews and publications.

Acknowledgments

This work was partly supported by a Nokia-Fulbright Distinguished Chair award and by US National Science Foundation Awards No. EEC-1941186, 1937950, 1939105. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

Notes

1 *Computing* and *computer science* are widely used terms in the English-speaking world. Other terms are also used elsewhere, such as *informatics or information technology*. For simplicity, we use the previous ones systematically.
2 Programming paradigms basically characterize the principles of how programs are structured. Commonly identified paradigms include procedural programming, object-oriented programming, functional programming, and declarative programming. As modern programming languages support different ways of building programs, the whole concept of paradigm has been questioned (Krishnamurthi & Fisler, 2019).
3 Computing teachers naturally use many other educational technologies, too, including generic learning management systems (LMS) and tools for specific learning activities, for example, discussion forums or data visualization, but we are not discussing them here.
4 Some commercial tools have originally been developed in universities, before they were commercialized, such as Blackboard (originally WebCT), or CodeGrade (www.codegrade.com/).
5 www.bluej.org/; https://pythontutor.com/; www.jflap.org/; https://web-cat.org/.
6 https://peerwise.cs.auckland.ac.nz.
7 https://doodlepad.org/; http://dept.cs.williams.edu/~tom/weavingCS/s07/doc/squintDoc/; www.cs1graphics.org/.
8 https://www.eclipse.org; www.jetbrains.com/idea/; https://visualstudio.microsoft.com/.
9 There is a lot of research on MOOCs, including computing-specific MOOCs, most of which is disseminated in aligned fields, such as educational data mining and learning analytics.
10 https://runestone.academy/runestone/books/published/StudentCSP/index.html. Runestone Academy (https://runestone.academy/ns/books/index) has a wide set of free e-books for computer science.
11 https://opendsa-server.cs.vt.edu/.
12 https://stack-assessment.org/.
13 www.numbas.org.uk/.

References

ACM/IEEE – CS Joint Task Force on Computing Curricula. (2013). *Computer science curricula 2013*. ACM Press and IEEE Computer Society Press.

Adams, R., Aldridge, D., Atman, C., Barker, L., Besterfield-Sacre, M., Bjorklund, S., & Young, M. (2006). The research agenda for the new discipline of engineering education. *Journal of Engineering Education*, *95*(4), 259–261.

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

Ala-Mutka, K., & Jarvinen, H. M. (2004). Assessment process for programming assignments. In *IEEE international conference on advanced learning technologies, 2004. Proceedings*. (pp. 181–185). IEEE.

Atchison, W. F., Conte, S. D., Hamblen, J. W., Hull, T. E., Keenan, T. A., Kehl, W. B., . . . Young Jr, D. M. (1968). Curriculum 68: Recommendations for academic programs in computer science: A report of the ACM curriculum committee on computer science. *Communications of the ACM*, *11*(3), 151–197.

Austing, R. H., Barnes, B. H., Bonnette, D. T., Engel, G. L., & Stokes, G. (1979). Curriculum'78: Recommendations for the undergraduate program in computer science – a report of the ACM curriculum committee on computer science. *Communications of the ACM*, *22*(3), 147–166.

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, *20*(1), 45–73.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, *39*(2), 32–36.

Bergin, S., & Reilly, R. (2005). Programming: Factors that influence success. *ACM SIGCSE Bulletin*, *37*(1), 411–415.

Brown, N. C., Altadmri, A., Sentance, S., & Kölling, M. (2018, August). Blackbox, five years on: An evaluation of a large-scale programming data collection project. In *Proceedings of the 2018 ACM conference on international computing education research* (pp. 196–204).

Brusilovsky, P., Edwards, S., Kumar, A., Malmi, L., Benotti, L., Buck, D., Ihantola, P., Prince, R., Sirkiä, T., Sosnovsky, S., & Urquiza, J. (2014). Increasing adoption of smart learning content for computer science education. In *Proceedings of the working group reports of the 2014 on innovation & technology in computer science education conference* (pp. 31–57).

Computing Curricula. (2005). *ACM*. www.acm.org/education/curricula-recommendations

Computing Curricula. (2020). *ACM*. www.acm.org/education/curricula-recommendations

Danielsiek, H., Toma, L., & Vahrenhold, J. (2017). An instrument to assess self-efficacy in introductory algorithms courses. In *Proceedings of the 2017 ACM conference on international computing education research (ICER '17)* (pp. 217–225). ACM.

Denning, P. J. (Chairman), Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., & Young, P. R. (1989). Computing as a discipline. *Communications of the ACM*, *32*(1), 9–23.

Denning, P. J., & Tedre, M. (2019). *Computational thinking*. MIT Press.

Denny, P., Hamer, J., Luxton-Reilly, A., & Purchase, H. (2008). PeerWise: Students sharing their multiple choice questions. In *Proceedings of the fourth international workshop on computing education research* (pp. 51–58).

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). Codewrite: Supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 471–476).

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73.

Duran, R., Sorva, J., & Leite, S. (2018). towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 2018 ACM conference on international computing education research* (pp. 21–30).

Duran, R., Zavgorodniaia, A., & Sorva, J. (2022). Cognitive load theory in computing education research: A review. *ACM Transactions on Computing Education (TOCE)*, *22*(4).

Eckerdal, A. (2009). *Novice programming students' learning of concepts and practice*. (Doctoral dissertation). Acta Universitatis Upsaliensis.

Edwards, S. H., & Murali, K. P. (2017). CodeWorkout: Short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM conference on innovation and technology in computer science education* (pp. 188–193).

Edwards, S. H., & Perez-Quinones, M. A. (2008). Web-CAT: Automatically grading programming assignments. In *Proceedings of the 13th annual conference on innovation and technology in computer science education* (pp. 328–328).

Ericson, B. J., Rogers, K., Parker, M., Morrison, B., & Guzdial, M. (2016). Identifying design principles for CS teacher Ebooks through design-based research. In *Proceedings of the 2016 ACM conference on international computing education research* (pp. 191–200).

Falkner, K., & Sheard, J. (2019). Pedagogic approaches. In S. A. Fincher & A. V. Robins (Eds.). *The Cambridge handbook of computing education research*. Cambridge University Press.

Falkner, K., Vivian, R., & Falkner, N. (2014). Identifying computer science self-regulated learning strategies. In *Proceedings of the 2014 conference on innovation & technology in computer science Education (ITiCSE '14)* (pp. 291–296). ACM.

Fincher, S., Johan Jeuring, J., Miller, C., Donaldson, P., Du Boulay, B. Hauswirth, M., Hellas, A., et al. (2020). Notional machines in computing education: The education of attention. In *Proceedings of the working group reports on innovation and technology in computer science education* (pp. 21–50).

Fincher, S. A., & Robins, A. V. (Eds.). (2019). *The Cambridge handbook of computing education research*. Cambridge University Press.

Finnie-Ansley, J., Denny, P., Becker, B. A., Luxton-Reilly, A., & Prather, J. (2022). The robots are coming: Exploring the implications of OpenAI codex on introductory programming. In *Australasian computing education conference* (pp. 10–19).

Fouh, E., Karavirta, V., Breakiron, D. A., Hamouda, S., Hall, S., Naps, T. L., & Shaffer, C. A. (2014). Design and architecture of an interactive eTextbook – The OpenDSA system. *Science of Computer Programming*, *88*, 22–40.

Goncher, A., Hingle, A., Johri, A., & Case, J. (2023). The role and use of theory in engineering education research. In A. Johri (Ed.), *International handbook of engineering education research* (pp. 137–155). Routledge.

Greca, I. M., & Moreira, M. A. (2000). Mental models, conceptual models, and modelling. *International Journal of Science Education*, *22*(1), 1–11.

Gross, S., & Pinkwart, N. (2015, July). towards an integrative learning environment for Java programming. In *2015 IEEE 15th international conference on advanced learning technologies* (pp. 24–28). IEEE.

Grover, S., & Korhonen, A. (2017). Unlocking the potential of learning analytics in computing education. *ACM Transactions on Computing Education*, *17*(3), (pp. 1–4).

Guzdial, M. (2003, June). A media computation course for non-majors. In *Proceedings of the 8th annual conference on innovation and technology in computer science education* (pp. 104–108).

Guzdial, M. (2013, August). Exploring hypotheses about media computation. In *Proceedings of the ninth annual international ACM conference on international computing education research* (pp. 19–26).

Guzdial, M. (2016). Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, *8*(6), 1–165.

Guzdial, M., & du Boulay, B. (2019). The history of computing education research. *The Cambridge Handbook of Computing Education Research (2019)*, pp. 11–39.

Haaranen, L., Mariani, G., Sormunen, P., & Lehtinen, T. (2020). Complex online material development in CS courses. In *Koli calling'20: Proceedings of the 20th Koli calling international conference on computing education research* (pp. 1–5).

Hellas, A., Ihantola, P., Petersen, A., Ajanovski, V. V., Gutica, M., Hynninen, T., Knutas, A., Leinonen, J., Messom, C., & Liao, S. N. (2018, July). Predicting academic performance: a systematic literature review. In *Proceedings companion of the 23rd annual ACM conference on innovation and technology in computer science education* (pp. 175–199).

Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012). How do students solve parsons programming problems? An analysis of interaction traces. In *Proceedings of the ninth annual international conference on international computing education research* (pp. 119–126).

Hundhausen, C., Agrawal, A., Fairbrother, D., & Trevisan, M. (2009). Integrating pedagogical code reviews into a CS 1 course: An empirical study. *ACM SIGCSE Bulletin*, *41*(1), 291–295.

Hundhausen, C. D., Douglas, S. A., & Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, *13*(3), 259–290.

Höök, L. J., & Eckerdal, A. (2015). On the bimodality in an introductory programming course: An analysis of student performance factors. In *Learning and teaching in computing and engineering (LaTiCE 2015)* (pp. 79–86). IEEE.

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010, October). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research* (pp. 86–93).

Kao, Y., Matlen, B., & Weintrop, D. (2022). From one language to the next: Applications of analogical transfer for programming education. *ACM Transactions on Computing Education (TOCE)*, *22*(4).

Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, *19*(1), 1–43.

Kinnunen, P., & Simon, B. (2011). CS majors' self-efficacy perceptions in CS1: Results in light of social cognitive theory. In *Proceedings of the seventh international workshop on computing education research (ICER '11)* (pp. 19–26). ACM.

Krishnamurthi, S., & Fisler, K. (2019). Programming paradigms and beyond. *The Cambridge handbook of computing education research*, 377–413.

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1989). A study of the development of programming ability and thinking skills in high school students. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 83–112). Lawrence Erlbaum.

Kölling, M. (2009). *Quality – oriented teaching of programming*. Retrieved November 11, 2022, from https://blogs.kcl.ac.uk/proged/2009/09/04/quality-oriented-teaching-of-programming/

Lahtinen, E., Ala – Mutka, K., & Järvinen, H. M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, *37*(3), 14–18.

Lajis, A., Baharudin, S. A., Ab Kadir, D., Ralim, N. M., Nasir, H. M., & Aziz, N. A. (2018). A review of techniques in automatic programming assessment for practical skill test. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, *10*(2–5), 109–113.

Lishinski, A., Good, J., Sands, P., & Yadav, A. (2016). Methodological rigor and theoretical foundations of CS education research. In *Proceedings of the 2016 ACM conference on international computing education research* (pp. 161–169).

Lishinski, A., & Yadav, A. (2019). Motivation, attitudes, and dispositions. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 801–826). Cambridge University Press.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., & Simon, B. (2004). A multi – national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119–150.

Loksa, D., Margulieux, L., Becker, B. A., Craig, M., Denny, P., Pettit, R., & Prather, J. (2022). Metacognition and self-regulation in programming education: Theories and exemplars of use. *ACM Transactions on Computing Education (TOCE)*, *22*(4).

Loui, M. C., & Borrego, M. (2019). Engineering education research. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 292–322). Cambridge University Press.

Luxton-Reilly, A., Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., Paterson, J., Scott, M. J., Sheard, J., & Szabo, C. (2018). Introductory programming: A systematic literature review. In *Proceedings companion of the 23rd annual ACM conference on innovation and technology in computer science education* (pp. 55–106).

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007). Investigating the viability of mental models held by novice programmers. *ACM SIGCSE Bulletin*, *39*(1), 499–503.

Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppälä, O., & Silvasti, P. (2004). Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, *3*(2), 267–288.

Malmi, L., Sheard, J., Bednarik, R., Helminen, J., Kinnunen, P., Korhonen, A., Myller, N., Sorva, J., & Taherkhani, A. (2014). Theoretical underpinnings of computing education research: What is the evidence? In *Proceedings of the tenth annual conference on international computing education research* (pp. 27–34).

Malmi, L., Sheard, J., Kinnunen, P., Simon, & Sinclair, J. (2019). Computing education theories: What are they and how are they used? In *Proceedings of the 2019 ACM conference on international computing education research* (pp. 187–197).

Malmi, L., Sheard, J., Kinnunen, P., Simon, & Sinclair, J. (2020). Theories and models of emotions, attitudes, and self-efficacy in the context of programming education. In *Proceedings of the 2020 ACM conference on international computing education research* (pp. 36–47).

Malmi, L., Sheard, J., Kinnunen, P., & Sinclair, J. (2023). Development and use of domain-specific learning theories, models and instruments in computing education. *ACM Transactions on Computing Education (TOCE)*, *23*(1).

Margulieux, L. E., Dorn, B., & Searle, K. A. (2019). *Learning sciences for computing education*. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 208–230). Cambridge University Press.

Mayer, R., & Mayer, R. E. (Eds.). (2005). *The Cambridge handbook of multimedia learning*. Cambridge University Press.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*, 125–180.

Murphy, L., & Thomas, L. (2008, June). Dangers of a fixed mindset: Implications of self-theories research for computer science education. In *Proceedings of the 13th annual conference on innovation and technology in computer science education* (pp. 271–275).

Novak, M., Joy, M., & Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Transactions on Computing Education (TOCE)*, *19*(3), 1–37.

Paiva, J. C., Leal, J. P., & Figueira, Á. (2022). Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)*, *22*(3), 1–40.

Papert, S. A. (2020). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.

Patitsas, E., Berlin, J., Craig, M., & Easterbrook, S. (2019). Evidence that computer science grades are not bimodal. *Communications of the ACM*, *63*(1), 91–98.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1989). Conditions of learning in novice programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 261–279). Lawrence Erlbaum.

Petre, M., Sanders, K., McCartney, R., Ahmadzadeh, M., Connolly, C., Hamouda, S., Harrington, B., Lumbroso, J., Maguire, J., Malmi, L., & McGill, M. M. (2020). Mapping the landscape of peer review in computing education research. In *Proceedings of the working group reports on innovation and technology in computer science education* (pp. 173–209).

Pettit, R. S., Homer, J., & Gee, R. (2017, March). Do enhanced compiler error messages help students? Results inconclusive. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education* (pp. 465–470).

Pullan, W., Drew, S., & Tucker, S. (2013, September). An integrated approach to teaching introductory programming. In *2013 second international conference on E-learning and E-technologies in education (ICEEE)* (pp. 81–86). IEEE.

Ramalingam, V., & Wiedenbeck, S. (1998). Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research*, *19*(4), 367–381.

Rist, R. S. (2004). Learning to program: Schema creation, application, and evaluation. In S. Fincher & M. Petre (Eds.), *Computer science education research* (pp. 175–195). Taylor & Francis.

Rivers, K., & Koedinger, K. R. (2017). Data-driven hint generation in vast solution spaces: A self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, *27*(1), 37–64.

Robins, A. V. (2010). Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, *20*(1), 37–71.

Robins, A. V. (2019). Novice programmers and introductory programming. In *The Cambridge handbook of computing education research* (pp. 327–377). Cambridge University Press.

Robins, A. V., Haden, P., & Garner, S. (2006). Problem distributions in a CS1 course. In *Proceedings of the eighth Australasian computing education conference (ACE 2006)*, CRPIT, 52 (pp. 165–173). Australian Computer Society.

Robins, A. V., Margulieux, L. E., & Morrison, B. B. (2019). Cognitive sciences for computing education. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 231–275). Cambridge University Press.

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

Shaffer, C. A., Cooper, M. L., Alon, A. J. D., Akbar, M., Stewart, M., Ponce, S., & Edwards, S. H. (2010). Algorithm visualization: The state of the field. *ACM Transactions on Computing Education (TOCE)*, *10*(3), 1–22.

Simon, Luxton-Reilly, A., Ajanovski, V. V., Fouh, E., Gonsalvez, C., Leinonen, J., Parkinson, J., Poole, M., & Thota, N. (2019). Pass rates in introductory programming and in other STEM disciplines. In *Proceedings of the working group reports on innovation and technology in computer science education* (pp. 53–71).

Sirkiä, T. (2016). October. Jsvee & Kelmu: Creating and tailoring program animations for computing education. In *2016 IEEE working conference on software visualization (VISSOFT)* (pp. 36–45). IEEE.

Solomon, C., Harvey, B., Kahn, K., Lieberman, H., Miller, M. L., Minsky, M., Papert, A., & Silverman, B. (2020). History of Logo. *Proceedings of the ACM on Programming Languages*, *4*(HOPL), 1–66.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *Transactions on Software Engineering* (5), 595–609.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1983). What do novices know about programming? In B. Shneiderman & A. Badre (Eds.), *Directions in human – computer interactions* (pp. 27–54). Ablex.

Soloway, E., & Spohrer, J. C. (Eds.). (1989). *Studying the novice programmer*. Lawrence Erlbaum.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31.

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)*, *13*(4), 1–64.

Sweller, J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, *22*(2), 123–138.

Sweller, J., Van Merrienboer, J. J., & Paas, F. G. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, *10*(3), 251–296.

Szabo, C., Falkner, N., Petersen, A., Bort, H., Cunningham, K., Donaldson, P., Hellas, A., Robinson, J., & Sheard, J. (2019). Review and use of learning theories within computer science education research: Primer for researchers and practitioners. In *Proceedings of the working group reports on innovation and technology in computer science education* (pp. 89–109).

Szabo, C., & Sheard, J. (2023). Learning theories use and relationships in computing education research. *ACM Transactions on Computing Education*, *23*(1), 1–34.

Teague, D., & Lister, R. (2014). Longitudinal think aloud study of a novice programmer. In *Proceedings of the sixteenth Australasian computing education conference (ACE2014), conferences in research and practice in information technology series* (Vol. 148).

Tedre, M., & Pajunen, J. (2023). Grand theories or design guidelines? Perspectives on the role of theory in computing education research. *ACM Transactions on Computing Education (TOCE)*, *23*(1).

Tedre, M., Simon, & Malmi, L. (2018). Changing aims of computing education: A historical survey. *Computer Science Education*, *28*(2), 158–186.

Tedre, M., & Sutinen, E. (2008). Three traditions of computing: What educators should know. *Computer Science Education*, *18*(3), 153–170.

Tenenberg, J., & Malmi, L. (2022). Editorial: Conceptualizing and use of theory in computing education research. *ACM Transactions on Computer Science Education*, *22*(4).

Tucker, A. B., & Barnes, B. H. (1990). *Computing curricula 1991–Report of the ACM/IEEE-CS joint curriculum task force*. Jointly published by ACM Press, IEEE Computer Society Press.

Umapathy, K., & Ritzhaupt, A. D. (2017). A meta-analysis of pair-programming in computer programming courses: Implications for educational practice. *ACM Transactions on Computing Education (TOCE)*, *17*(4), 1–13.

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., Paterson, J., Caspersen, M., Kolikant, Y. B. D., Sorva, J., & Wilusz, T. (2013). A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE 2013* (pp. 15–32).

Valentine, D. W. (2004). CS educational research: A meta-analysis of SIGCSE technical symposium proceedings. *ACM SIGCSE Bulletin*, *36*(1), 255–259.

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on innovation & technology in computer science education* (pp. 39–44). ACM.

Watson, C., Li, F. W., & Godwin, J. L. (2014). No tests required: Comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 469–474).

Weinberg, G. M. (1971). *The psychology of computer programming*. Van Nostrand Reinhold.

Winslow, L. E. (1996). Programming pedagogy – A psychological overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22.

Wirth, N. (1971). The programming language Pascal. *Acta Informatica*, *1*(1), 35–63.

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Tan, A. H., Hwa, L., Li, M., & Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, *29*(2–3), 205–253.

Yadav, A., & Lachney, M. (2023). Towards a techno-social realist approach in primary and secondary computing education. In A. Johri (Ed.), *International handbook of engineering education research* (pp. 553–572). Routledge.