Tools and environments

Lauri Malmi, Ian Utting, Andrew J. Ko

1 Introduction

The roots of computing as a discipline stem from the need to present and manipulate data automatically. Building software tools has therefore always been an important activity in the field, both in professional practice and in academic work. Correspondingly, learning to use tools needed for professional programming has been one of the core elements in computing education. Languages have, of course, changed and developed over the decades, but the basic skills in using compilers, interpreters, editors, and debuggers have always been necessary to learn. In current computing education, similar learning goals persist even though modern IDEs, like Eclipse, have reduced (but not excluded) the need to learn to use separate tools.

There is a large diversity of types of tools that contribute to the learning of computing. Some programming languages and environments have been developed primarily as tools for education. Early examples of these included BASIC (Kemeny & Kurtz, 1964) and Pascal (Wirth, 1973). More recent cases include, for example, Scratch (Resnick et al., 2009) and Alice (Cooper et al., 2000). The basic idea behind these languages has been to provide a simplified environment for learning programming, and hide many complexities in the world of professional programming. The more recent graphical programming environments also aim at reaching the younger population (e.g., Meerbaum-Salant et al., 2013), from preschool onwards, instead of tertiary-level students only. Though, some institutions also use these environments as a gentle introduction to programming (Cooper et al., 2003).

Other attempts at simplifying tools and environments for beginners have looked at ameliorating the complexity of elements of the professional tools which tend to be more complex and general than is needed in education. Areas of concern have been I/O and graphics libraries (Wolz & Koffman, 1999; Bruce et al., 2001), and even the language itself, revealing more complete subsets of the language as beginners develop their skills (Felleisen et al., 1998)

Learning programming has always been challenging for a large number of students. Therefore, it is natural that computing educators have started to develop educational tools which focus on supporting various aspects of students' learning process. A notable early example was the BALSA algorithm animation tool (Brown and Sedgewick, 1984), which provided visualizations of algorithm execution. Such tools were initially standalone applications and were often platform-dependent, limiting access to them. With the widespread availability of network access to resources which came with the internet, dissemination of tools became much easier. Moreover, due to the rapid expansion in the number and diversity of web development tools, and with the dramatic increase in the bandwidth available to students (at least in some

countries), most current educational tools are either wholly browser-based or have a browser interface which interacts with server-side applications.

In this chapter we seek to survey these various categories of tools and the research on them. We start by discussing the motivation for such research, what forms of tools exist and for whom they have been developed. Then, we present overviews of several important application areas. Finally, we discuss some generic challenges in tools research.

2 Why are tools important for Computing Education?

Computing is partly a practical field, and so most computer science education focuses on teaching students to design and implement complex systems to solve problems. Tools are usually a central part of the learning, but only implicitly, as technical implementations needed to reach a desired goal. It is therefore natural that computing educators have been highly active in developing and evaluating tools to support learning, as well as their own work as teachers. Tools can often be considered pedagogical support mechanisms, and compared with many other fields, computer scientists have the privilege that they (and their students) are much more often capable of designing and building the tools themselves.

It is thus not surprising that significant share of computing education research concern tools. Valentine (2004) surveyed 444 papers addressing CS1/CS2-level education published in 1983-2004 in SIGCSE Symposium proceedings, and identified that in 99 of them (22%) the main topic was some form of tool. Simon (2007) analyzed 4 years of ACE and NACCQ conferences (2004-2007) and identified that in 16% of the papers the topic was teaching/learning/assessment tools. In further analysis Simon (2009), showed that 26% of conference papers in the first 7 years (2001-2007) of the Koli Calling conference concerned tools, as did 17% of ITiCSE papers in years 2005-2008 (Sheard et al., 2009).

3 What is a tool?

Educational tools in computing take many forms. An obvious case is a *software application*, which one can download and install, such as BlueJ (www.bluej.org) environment for learning object-oriented programming in Java. A tool could also be a *web application providing some service*, like Problets (www.problets.org), which are “problem solving software assistants for learning, reinforcement and assessment of programming concepts”, and PeerWise (peerwise.cs.auckland.ac.nz), which supports student-generated multiple choice questions as a pedagogical method. On the other hand, there are hundreds or thousands of small interactive applications available via the web which demonstrate the operation of individual concepts, like particular data structures or sorting algorithms. Often these have been developed as student projects, and published, “as is”, sometimes collected and curated, and possibly incorporating other non-tool resources such as YouTube videos.

Another class of tools includes *software frameworks* that can be used to build new educational applications. For example, Tango (Stasko, 1990) was a software library to build algorithm animations. The increase in availability of rich interactive content, also called smart learning content (Brusilovsky et al., 2014), is often the result of the availability of such frameworks, because the same technological framework can been used to generate multiple different instances of examples or exercises. For instance, the jsVEE framework (Sirkiä, 2016) has been developed to enable building visualizations of Scala and Python programs, and jsParsons (Helminen et al., 2012) is a tool for building Parsons problems. Instances of these visualizations and problems have been made available at the ACOS content server (Sirkiä and Haaranen, 2017, acos.cs.hut.fi), which is a software architecture supporting dissemination of smart learning content into different learning management systems. All of these frameworks and servers can be considered tools for teachers to create and disseminate smart learning content for students.

However, a tool can also be a *definition language*. Initially, the Pascal language was designed as an educational programming language. While, there had to be Pascal compilers to build executable programs, these were separate tools generating code for different platforms. Another example of language tools is that some algorithm visualization systems, like JAWAA (www.cs.duke.edu/csed/jawaa2), are built on using a scripting language, which can be processed into a visualization (Rodger, 2002).

Of course, there are many tools used in computing education that are not specific to the discipline, and many tools which are specific to the discipline but are not in essence educational. In the rest of this chapter we will leave aside discussion of generic educational tools and platforms such as learning management and support systems like Moodle or PeerWise, and general interactive tutorial systems. Professional tools, like Eclipse or general purpose programming languages and corresponding systems software are frequently used in computing education, and learning to use them is an important goal of learning in computing curriculum. While these may be complex for novices, it is then possible for teachers to focus on using only a core set of operations, and even provide a simplified user interface for students. Later on, actual professional tools can be adopted. We do not discuss professional tools explicitly in this chapter.

Finally, we restrict our discussions to software tools, leaving out tools for physical computing or CS unplugged (www.csunplugged.org) (Chapters 3.7 and 3.11 focus on these topics).

4 Stakeholders

Tools can be considered from several stakeholders' points of view. Many educational tools and applications focus on supporting *students'* work, including, for example, tools for visualizing and concretizing program execution (program visualization, algorithm visualization), intelligent tutoring systems, automatic feedback tools, etc. But some tools have primarily been developed for *teachers*, such as plagiarism detection tools, (e,g. Lancaster & Culwin, 2004; Rosales et al.,

2008) or frameworks (see above) which allow teachers to build new learning content or assist in assessment. These tools can, of course, also be used by students, for instance in project work where the task is to delve into some topic and prepare a learning resource as the outcome.

Teachers can act both as *producers* and *consumers*. Firstly, teachers can build new learning content using some tool and use it in their own education, while also providing other teachers the possibility to access and adopt the material. Then the other teachers act as consumers. They might use the available resources as such, but they could also wish to tune them, for example, by adding annotations to the material. Kelmu (Sirkiä, 2016) is such a tool for consumers, which has been used to annotate program visualizations, but could be used to annotate other types of animations, too.

Finally there are *developers*, who implement and maintain tools. Of course, teachers could act as developers, too, and often do. However, it is useful to differentiate between these roles, because the needs for tools are quite different from the point of views of various roles. For teachers and students, easy access, low learning curve and high usability are important, while for developers, maintainability, code quality, choice of technologies, architecture, and available documentation are more important.

5 Classification of tools

The range of tools computing education is broad, and it is impossible to discuss thoroughly all tools research in this chapter. It is, however, useful to provide some structure for the area. A good resource here is the paper by Kelleher and Pausch (2005) who presented a taxonomy of educational tools for programming, identifying several dozen different tools designed for novice programmers to learn to code. While the paper focuses narrowly on programming, and not all topics in computing, their paper actually covers a great majority of the tools relevant tools to this chapter.

Kelleher and Pausch split tools into two major categories. *Teaching systems* attempt to teach programming for its own sake and have goals, such as simplifying the creation of code, finding alternatives to typing programs, providing new ways of structuring programs, supporting better understanding of program execution, supporting social learning, and providing a motivating context. The second major category they called *empowering systems*. They argued that “the designers of these [latter] systems are not concerned with how well users can translate knowledge ... to a standard programming language. Instead, they focus on trying to create languages and methods of programming that allow people to build as much as possible.” Here they identify systems which support new methods of specifying the program logic, improvements in programming languages, and applications in entertainment and education in other domains of knowledge.

Another, more narrow, survey of tools supporting programming education, published at roughly same time, is the work of Gómez-Albarrán (2005). It focuses on 20 important tools operating in four different domains: reducing the complexity of the development environment; providing examples [to guide/of] programming; visualizing program/algorithm execution; and providing a simulated world where the programmer can control activities. These fall in the domain of teaching systems in the categorization of Kelleher and Pausch.

While this top level categorization is helpful for comparing the goals of the system, it does not provide much granularity about the instructional and learning properties in prior work. Moreover, there has been more than a decade of additional work since its publication. The rest of this chapter will attempt to provide some of this increased granularity, with sections discussing some of the most important types of tools that have been designed to support different aspects of learning computing.

A broad area could be described as tools for *scaffolding learning.* First, there are tools and environments which support *planning, designing, analyzing and constructing programs* (section 5.1). These tools provide a simplified environment, which hides some of the complexity of programming concepts and processes as compared to working directly with professional tools. For example, block-based programming languages and environments, like Scratch or Blockly (developers.google.com/blockly) efficiently remove the requirement for students to engage with fine details of syntax.

A second category of scaffolding tools covers different types of *feedback* for students on their work. As an example, *automatic assessment tools* (section 5.4) can provide feedback on students' programs in terms of their correctness, style, structure, and efficiency. There are also tools for analysing formal design specifications. Such tools are naturally valuable for teachers, too, because they can reduce their workload by analyzing and grading student work automatically. Teachers can then use their time more for guidance and giving feedback on such aspects of student work that are hard to analyze automatically, such as design choices or use of algorithms.

Third, computing concepts are frequently abstract and invisible. Therefore scaffolding student learning with *visualization* and *animation* (section 5.5) is a broadly explored area of tools research. Some such tools provide opportunities for various types of interaction with the visualization, such as responding to questions, browsing or zooming in and out in the dynamic presentation, or providing input values and thus *simulating* a system or operation.

In all of the above categories, one important aspect of tool-based scaffolding is the pace: getting immediate feedback on one's solution or work is valuable because it supports personal reflection on learning much better than getting teachers' feedback several days or even weeks later. Many tools also allow students to re-submit a revised solution or explore interactively how a particular system or concept works, although some impose a delay to avoid very fine-grained interaction.

The fourth area of scaffolding concerns support for students' motivation. This is an area where various forms of *educational games* and *gamified approaches* work (section 5.2). Finally, there is research carried out on *e-books* which form comprehensive interactive resources for learning programming (section 5.3).

We discuss each of these areas in more depth below.

5.1 Supporting the writing of programs

Since educational programming languages and environments first came into existence, there has always been a tension between providing specific tools to decrease barriers to entry and engage learners with with professional tools as their abilities grow.

Languages designed, or adapted, for use in education have long been integrated with development environments tailored to that purpose, from BASIC in the 60s through Turbo Pascal in the 80s and on to graphical IDEs (e.g. Scratch) in the 21st century.

In the early stages of their development, although these IDEs were designed to be used by professional software developers, they were simple enough to be used by beginners without significant learning overheads, but as they matured beyond the simple edit-compile-run cycle to include the rapidly expanding software development toolchain, they became less and less accessible. For their target professional audience, this is not a major problem as they are expected to use the tools every day and be prepared to invest in learning their increasingly complex interfaces, but for beginners, the barrier to entry became increasingly high. This led to a number of attempts from around 2000 to subset existing and emerging IDEs (Eclipse for Education, Visual Studio Express), or provide plugins specific to educational settings (the NetBeans BlueJ plugin). These efforts have largely been abandoned due to the cost of maintenance, or subsumed into a more general "freemium" licensing model where they have lost any specific educational focus.

A more successful approach has focussed on the creation of IDEs specifically targeting learners and existing languages: Dr Scheme for Scheme (Findler et al., 2002) which later became DrRacket, BlueJ for Java (Kölling, 2003) and DrJava (Allen et al., 2002). As well as providing features for learners (such as BlueJ's direct object manipulation and Dr Scheme's Read Evaluate Print Loop), these tools typically included a subset of the functionality of professional IDEs, which over time expanded to include access to other components of the modern software development tool chain such as static code analysis (Cardell-Oliver & Wu, 2011) and revision control systems (Fisker et al., 2008). The latter has particularly opened up research opportunities using students' commits of evolving source code as a research tool (Spacco et al., 2006)

Despite the existence of this wide variety of IDEs for education, the use of command line tools alone for teaching introductory programming is remarkably persistent, with Davies et al. (2011)

reporting that 15% of their sample of 367 US institutions used the command line alone in their CS1 course. A similar proportion was reported for Australian institutions in 2010 by Mason et al. (2010), although they report a decline from 45% in the previous 10 years citing as the major reason a change of perception from IDEs representing an increase in cognitive load ("learning to use the IDE") to a reduction ("reducing the amount that the students had to learn"). With a few exceptions (e.g., Uysal, 2016) these perceptions remain largely untested by research, with such work as has been done largely subsidiary to comparisons of programming languages (e.g. McIver, 2001).

5.2 Games and learning programming

Whereas many of the tools discussed above have played the role of being supportive of learning by simplifying programming languages or providing more supportive code editors or programming environments (Kelleher & Pausch, 2005), some tools have tried to explicitly structure, scaffold, and guide the entire learning experiences. These environments are much more than tools, often offering whole curricula for a set of learning objectives. These environments fall roughly into two categories: *interactive games* and *e-books* (which are similar to tutorials and tutors). Both types of environments offer learning materials, curricula, and some form of sequencing to guide learning but differ in how they motivate and reinforce learning.

Game-based learning environments motivate learning by creating extrinsic motivation (Gee, 2014) and applying game mechanics and instructional principles to achieve particular learning objectives (Aleven et al., 2010). There have been numerous games that teach aspects of programming. Among the earliest were the Rocky's Boots and Robot Odyssey games (Robinett & Grimm, 1982), which offered a series of increasingly difficult puzzles in which players connected logic gates to achieve particular program outputs. This puzzle-based paradigm is followed in numerous other commercial games, including Lightbot (lightbot.com), CodeCombat (codecombat.com) and Human Resource Machine (tomorrowcorporation.com/humanresourcemachine). These tend to each offer their own simplified programming language, with game-specific commands and operations to manipulate a game world of some kind. Some other games focus on low level programming and/or building hardware, such as Shenzhen IO (www.zachtronics.com/shenzhen-io), Silicon Zeroes (store.steampowered.com/app/684270/Silicon_Zeroes) and TIS 100 (www.zachtronics.com/tis-100). It is also worth noting that while all these games focus on programming or computing concepts, there is variation over whether the focus of the game is mainly educational or entertainment.

Some researchers have explored the paradigm of puzzle-based programming games in more depth. For example, Gidget (Lee et al., 2014) explored a collaborative (rather than competitive) framing of a player's relationship with the computer, framing the computer as a reliable but fallible collaborator incapable of problem-solving; it also focused players on fixing defective programs rather than expecting players to write programs from scratch. Others have explored how to use games to teach more advanced topics, such as concrete debugging strategies

(Miljanovic & Bradbury, 2017), SQL queries (Soflano et al., 2015), test case generation (Tillmann & Bishop, 2014), and software engineering requirements gathering (Connolly & Stansfield, 2006). There is some evidence that programming games are well-liked (Ibrahim et al., 2010), that they quickly shift attitudes about programming to positive (Charters et al., 2014), and can lead to better learning outcomes than environments with similar interactive features that are not framed as games (Lee & Ko, 2015). However, there is little evidence that playing these games leads to transferable knowledge to other programming contexts. Moreover, even if they do, there is some evidence that most players play only a few levels of these games before abandoning them once they encounter difficulties and cannot find help (Yan et al., 2017).

While the focus of the previous tools was teaching programming or some aspects of it, there are also other game related approaches which support building motivation for learning programming. There are many games that allow the user to enhance their capabilities through programming. Kerbal Space Program (kerbalspaceprogram.com/en) is a game where the player designs and flies spaceships and rockets to explore the planetary system in the game. There is an additional component called kOS (ksp-kos.github.io/KOS) which adds a custom programming language to the game enabling players to control the rockets programmatically. Another example is the popular game Minecraft (minecraft.net) and its “redstone” mechanism to create logical operations. Interestingly Minecraft has been used to create some very complex artefacts, like a Basic interpreter (www.youtube.com/watch?v=t4e7PjRygt0) or Atari 2600 Emulator (www.youtube.com/watch?v=jPRkjNDmTlc). While these examples can be considered as their author’s heroic achievements in using such frameworks, there is more to it than this. In gaming communities a highly popular activity is game streaming, where a game player streams the game play with oral comments on the working. The above videos of minecraft belong to this category. Another category is live-streaming, for example in Twitch.tv, where the audience can also interact with the author through chat discussion. Streaming playing an educational game like Shenzhen I/O has a clear educational perspective, and can act as a method of attracting followers to learn more about programming (Haaranen and Duran, 2017). Such streaming has also extended to live programming (Haaranen, 2017). There is very little research currently on these novel approaches to demonstrating CS concepts and programming in this context.

5.3 E-books

Whereas programming games offer extrinsic motivations to learn, interactive e-book environments lack any game-specific premise or extrinsic motive, expecting learners to bring their own motivations to reading and completing the book. Unlike games, however, e-books often provide more explicit instruction and feedback on learning objectives. For example, commercial coding tutorials such as Codecademy (codecademy.com) and Khan Academy (khanacademy.org) offer structured curriculum for programming language basics, OpenDSA (Fouh et al., 2014, opendsa.org/) is an e-book on data structures and algorithms, and CS Principles (Ericson et al., 2016) is an interactive book on Python programming

(www.interactivepython.org/runestone/static/StudentCSP/index.html). All three of these are essentially content, expecting learners to be in a motivating context.

Research environments have focused on providing more detailed, interactive explanations of concepts. For example, some offer interactive program visualizations that supplement natural language explanations of concepts (Rossling & Vellaramkalayil, 2009; Miller & Ranum, 2012; Fouh et al., 2014), providing learners with opportunities to see programs execute. Some of these, such as the PLTutor environment, place program visualizations at the center of the experience, focusing learner attention on the specific effects of a programming language's semantics on a program's control flow and state, using natural language explanations to supplement the visualization (Nelson et al., 2017). Several studies of e-books have shown that use of e-book features varies wildly between different students (Alvarado et al., 2012), and different populations of users such as students and teachers (Ericson et al., 2015; Parker et al., 2017), but that deeper engagement with an e-book's interactive features is generally associated with better learning outcomes (Alvarado et al., 2012). One of the only studies of the causal effect of integrated program visualizations on learning showed that, at least with highly granular visualizations of program behaviour, learning outcomes are significantly higher than with no program visualizations at all (Nelson et al., 2017).

While games, e-books, tutorials, and tutors offer promising opportunities for learning, the body of evidence of their efficacy is still quite shallow, with only a few studies evaluating learning outcomes. Moreover, pedagogical analyses of these genres of learning technology show that all but the most carefully-designed research environments fail to meet even basic learning principles, such as adapting instruction to prior knowledge, providing personalized feedback on practice, and promoting self-regulated learning (Kim & Ko, 2016). There is considerable room for future work to personalize learning, while sustainably engaging learners in the use of these learning environments.

5.4 Supporting assessment and feedback

Assessment and feedback have always been a fundamental part of programming education, and grading student work in large introductory programming courses is very labour intensive. It is therefore no wonder that developing tools for automatic assessment of students' exercises has always received considerable interest among computing educators. Hollingsworth (1960) reported an early implementation of a mainframe tool for running submitted work against a set of tests and producing a grade from the results. This paper already notes the advantages in terms of cost (although here it's the cost of machine time),

Later work includes systems like ASSESS and AUTOMARK (Redish & Smyth, 1986), ASSYST (Jackson & Usher, 1997), Ceilidh (Burke et al., 1994) and PASS (Thorburn & Rowe, 1997). These systems analyze various aspects of programming assignments, such as program correctness, programming style, efficiency, and code complexity. ASSYST also evaluated students' test coverage and PASS compared student's submission with the teacher's solution

plan, and thus also focused on code design. Later important tools include BOSS (Joy et al., 2005), CourseMaster (Higgins et al., 2002) and Web-CAT (Edwards, 2003, www.web-cat.org). For more information, there are several survey papers, which cover the area well, except for the most recent work (Ala-Mutka, 2005; Douce et al., 2005; Ihantola et al., 2010).

Most work in automatic assessment tools has focussed on analyzing programming submissions, where several different aspects can be assessed. *Program correctness* is the most widely analyzed feature, which, in most cases, is implemented by running a student's program in a safe sandbox against teacher-defined test cases and comparing the results against teachers' model results. Text§-based comparison of output is the most common approach, but is fraught with difficulty in detecting semantically uninteresting variations in formatting, and is gradually being replaced by the use of automated testing frameworks such as JUnit (www.junit.org). *Programming style* is also a frequent measure, covering aspects like code indentation, function lengths, variable names etc. and may include analyzing *code complexity* using well-known software metrics. More recent work with Web-CAT (Edwards et al., 2017) reports using commercial static analysis programs to perform this function, but cautions that their comprehensive nature can result in over-emphasis on superficial "faults". Some more pedagogically-focussed systems trace whether certain *syntactic structures* have been used, either because they were requested in the assignment, or their use was denied. For example Scheme-robo (Saikkonen et al., 2001) compared the actual list structure which student's programs used against a model solution. *Code efficiency* can be measured by running the programs against test cases with different run time expectations. Some tools analyze also *program design/structure* by analyzing the functional structure (Thorburn & Rowe, 1997; Saikkonen et al., 2001).

ASSYST, and more recently Web-CAT, also analyze students' *test cases*. The rationale here is that many students start using automatic systems as kind of debugging tools, focusing only on passing teachers' test cases while the focus should be in their own testing. Edwards (2003) emphasizes test-driven development by analyzing how well students' own test cases have covered the source code (and that the code passes them) and additionally how well the program passes teachers' test cases. The results are combined and it is impossible for students to get high marks with only poor code coverage from their own tests.

While mainstream research has focussed on programming submissions, there are also other applications areas. CourseMaster (Higgins et al., 2002) provides tools to analyze flow-charts, OO diagrams and even electrical circuits. Ali et al. (2007) present a tool for analyzing UML diagrams. Shukur et al. (1999) present a tool for automatic analysis of formal specifications written in the Z language. Dekeyser et al. (2007) describe a tool for analyzing students' skills in SQL queries. Malmi et al. (2004) developed an algorithm simulation system called TRAKLA2 which allowed students to simulate, in terms of GUI manipulation operations, how an algorithm changes a given data structure, for example sorting an array of keys. The simulation sequence log is compared with the sequence generated by a correct implementation of the algorithm to provide feedback for the students. Students can view the model execution as an algorithm

animation, and restart the exercise with new randomized data; thus they can rehearse and refine their solution with no limitations until their final submission.

Automatic assessment tools can be used in different ways. Firstly, they can provide highly valuable *formative feedback* for students, because they can allow students to *resubmit* their work after considering the feedback, and refining their solution. Secondly, they can act as pre-filters for incomplete student submissions (Ala-Mutka et al., 2004; Joy et al., 2005) by checking, for example, that the program passes a number of tests, satisfies code quality requirements etc., before the submission is forwarded for teacher evaluation. The final marking is carried out by the teacher, who has access to the automatic analysis results of the submission. Plagiarism detection may also be included in this filtering phase. Finally, the tools can also be used for *summative assessment* where the results are automatically recorded as (part of) the grade awarded. In this kind of approach, it is important that the student gets information about how the marking was carried out, and, if in doubt, can contact the teacher for clarifying issues with the grading.

There are many challenges with automatic assessment, such as:

- What is the appropriate level and specificity of feedback? Mitrovic et al. (2000) analyzed the impact of 5 different detail levels of feedback by SQL-tutor, ranging from a ***simple report of success/failure to providing complete model solution, and explored the role of feedback level and student success. Best results were reported when students' were given relevant hints where to focus when correcting the erroneous solution, but challenges remain in identifying the "relevant" and delivering it in a timely fashion.
- In tools that allow students to revise their solution and resubmit it to the system, what is the appropriate number of allowed submissions and what kind of policy in general is good to use? As reported above, one problem is that students use the system as a testing tool instead of testing their programs themselves. Edwards (2003) provided one solution to this. Other solutions, if test case evaluation is not available, include limiting the number of submissions, imposing a penalty for excessive submissions, or slowing the process of providing feedback. Karavirta et al. (2006) and Malmi et al. (2005) have analyzed the impact of various resubmission policies in students work in the context of TRAKLA2 algorithm simulation exercises.
- How can students be discouraged from "working to the test" by iteratively creating programs which only pass the tests applied to them on submission? This can be addressed by randomising either the choice of tests, or details of the tests applied, but such randomization is not always trivial, as some cases are more likely to trigger test failures than others.

Chapter 3.3 elaborates on more on challenges in assessment.

5.5 Concretising the virtual/revealing the hidden

Software is a complex artifact. Its structure is in principle visible in program code but the length and complexity of the code can obscure the structure and make it difficult to capture even for professionals. The dynamic execution of a program is invisible. We can see only the effect (output) of the program, which could be correct or incorrect, or that the execution ends in an error. While error messages may give information about the proximate cause of the problem, the initial causes of the problem are generally not accessible by looking at the error messages or incorrect results only (Eisenstadt called this the “Cause/Effect Chasm” (1997). Therefore, both program code and the actual execution process both need to be considered. *Software visualization* is an area of research which seeks to address these challenges. It covers both educational and professional applications to understand software, its development and execution. Good overviews of the field include Diehl (2007) and Stasko (1998).

For educational purposes there are two main areas of software visualization: *program visualization (PV)* and *algorithm visualization (AV).* Program Visualization is “the visualization of actual program code or data structures in either static or dynamic form” (Stasko, 1998). It is a field which aims to make the code level execution of a program visible for a programmer. Some well-known examples are: Jeliot (Ben-Ari et al., 2011), jGrasp (www.jgrasp.org) and Online Python tutor (Guo et al., 2013). Sorva et al. presented a fairly recent survey of the whole field (2013). Most PVs are targeted to novices and are restricted to small, illustrative, programs since step-by-step execution of large software systems to investigate their logic is beyond the scope of reasonable work.

One way to view the goal of PV is to help learners understand a *notional machine* (Du Boulay, 1986), which is an abstract, most often simplified, model of what happens in program memory. Most PV systems target helping understanding of program execution on a notional machine level by showing how the program execution proceeds statement by statement, and visualizing variable values, simple data structures, stack frames and heap structures. Visualization is not popular on hardware (processor) level, because modern computers and processors are so complex that exact understanding of all details is frequently out of scope for even advanced practitioners. However, moderate understanding of, for example, what happens during the execution of Java or C programs is essential for becoming a competent programmer.

It is worth emphasizing that a notional machine is not a strictly defined concept; the level of abstraction can vary depending on what level of details are relevant for the stage and goals of education. The PLTutor system (Nelson et al., 2017) targeted a precise and comprehension level of understanding about JavaScript, providing detailed pedagogy about each individual operation at the instruction-level, rather than the line or sub-expression level of a program. The system was based on a theory of programming language semantics knowledge that argues that learning a language involves learning a mapping between syntax and the causal effects of syntax defined by the language’s semantics. To teach this mapping, the tutor interleaves conceptual instruction about language semantics with explanations of the side effects of individual components or language constructs. In one of the few studies of PV effects on

learning outcomes, this low-level of semantic granularity led to significantly better learning outcomes than coding tutorials that involve writing simple programs with these constructs.

Algorithm visualization (AV) is a genre of learning technologies that aim at visualizing program and data at a more abstract level, focusing more on data structures (typically depicted as graphical entities) and how a program manipulates those structures. While algorithm code is frequently included, the visualization focuses on what happens with data and not so much on code level details. The boundary between program and algorithm visualization is not strict and many systems include features on both levels. Some famous systems include Sorting-out-sorting video (Backer & Sherman, 1983), Tango (Stasko, 1990), jHAVÉ (Naps et al., 2000), TRAKLA2 (Malmi et al., 2004), ALVIE (Crescenzi, 2010), Animal (Rößling & Freisleben, 2002) and jFLAP (Rodger, 2006). A somewhat dated but useful survey of the field can be found in (Shaffer, 2010). A collection of algorithm visualization systems and resources can be found in www.algoviz.org.

Most program and algorithm visualization systems focus on *animation* of the execution, where the user can browse the dynamic execution stepwise, possibly choosing the granularity of steps, while executions is carried out by the computer. An alternative approach is *program simulation* or *algorithm simulation,* where the user acts as the processor and executes the program or algorithm using available graphical interaction tools. The system can evaluate the correctness of the steps and provide feedback for the student. An example of program simulation systems is UUhistle (Sorva and SIrkiä, 2010) and examples of algorithm simulation systems include TRAKLA2 (Malmi et al., 2004) and jSAV (Karavirta & Shaffer, 2013).

The literature in this area also uses other terms. *Code visualization* and *code animation* overlap with program visualization but focus solely on code level details. *Program animation* is sometimes used as a synonym of PV. *Static program visualization* refers to (static) visualization of program structures rather than (dynamic) execution. *Visual debugging* refers to debuggers which include visualizations of program and data structures instead of textual program code only. *Visual programming* is, on the other hand, an entirely separate genre of programming language and environments, which focuses on constructing programs using visual rather than textual entities. These visual languages have nothing to do with visualization tools.

There has been considerably empirical research on PV and AV systems, especially the latter. Research has mainly focused on engagement and presentation. A milestone in the research was the meta-study of empirical evaluations of AV systems by Hundhausen et al. (2002). The results indicated that the main difference between studies where better learning results of using AV systems in education were reported or not reported, was *student engagement.* That is, if students were actively working with the visualizations vs. merely viewing them, they learned better. An important follow-up work was carried out in an ITiCSE working group (Naps et al., 2002) who defined the *engagement taxonomy* for differentiating between various forms of engaging activities while using AV systems. This widely cited framework has guided much of the consequent research. In its original form it identified six different modes. *No viewing* (nothing)

and *viewing* were the lowest levels with no engaging activity involved. In *responding* mode learners are presented with questions related to the visualization. *Changing* level allows them to modify the visualization, for example, by varying the input data set. In *construction* mode they can create their own visualization of a program or an algorithm and finally in *presenting* mode they present visualizations to others for feedback and discussion.

Many studies have been carried out to evaluate whether the assumption holds that higher engagement levels would lead to higher learning results. While many positive results have been found, there is also critique that the taxonomy is insufficient, and should be revised or augmented. Myller et al. (2009) extended the taxonomy with finer variations in the interaction with visualization tool and used it to analyze collaborative learning process. Sorva et al. (2013) presented another extension by adding a second dimension concerning the ownership of the task at hand. That is, the engagement is also related to whether learners only manipulate given content or provide their own input cases, modify the visualization software, or even create their own software. Furthermore, few studies actually investigate whether program visualization tools cause better learning. The most recent evaluation of PLTutor (Nelson et al., 2017) involved a controlled experiment that holistically demonstrated improved learning outcomes, but this study did not separate the effect of the visualization from other aspects of its surrounding instruction.

While the engagement taxonomy has had a significant impact on research, there are no compelling results to confirm that the original taxonomy or its extensions reflect accurately the relation of engagement and learning outcomes. These remain open questions. Moreover, the taxonomy also does not consider the role of how information is presented in the visualization and what kind of textual or audio information supports the visualization.

6 Discussion

6.1 Incentives for tools research

Tool contributions in computing education research are likely to be of continued importance. First, digitalization and massification are transforming education at all levels. While the emergence of massive open online courses (MOOCs) a few years ago did not cause the revolution many teachers in computing education feared (Eckerdal et al, 2014; Sheard et al. 2014), they have become a part of mainstream education and have a significant role both in tertiary education and, particularly, Continuing Professional Development. Due to the sheer number of participants, often many thousands, MOOCs cannot be operated without software tools. With the increased number of students pursuing computing education in higher education, many of these same challenges of teaching at scale have reached traditional institutions of education, thus creating an incentive for development and deployment of automatic assessment and feedback tools. Moreover, because MOOCs cannot generally build on pedagogical foundations that require student-teacher interaction, there is an obvious need for tools supporting self-study of computing topics. Peer review and group work are also widely used in

MOOCs and so there is also an incentive to develop more computing-specific tools supporting these methods.

The influence of MOOCs is also apparent in *blended learning*, which has become a mainstream method for organizing and managing traditional face-to-face teaching and learning in the face of rising enrollments, providing similar incentives for tool development and use. Furthermore, pedagogical practices are moving towards student-centered approaches that generally activate students more compared with the teacher-centered tradition. While practical exercises using either professional or educational tools have always been a central part of computing education, the shift towards even more active methods further drives efforts on tool development and research.

The second important factor promoting tools is the demographic change in students. This is visible in two ways. First, students of this generation have used digital tools for their whole life and often expect that digital tools are a natural part of their studying. To keep students motivated, learning environments have to develop to include more (appropriate) digital content and opportunities for student interaction. Secondly, institutions have a large and growing number of non-traditional students, who study from distance either temporarily or permanently and who thus need more digital content and services than traditional students. They are blended learners by necessity.

The use of learning analytics (see Chapter 3.14) is a third factor. They are increasingly applied in education at all levels to analyze student behavior. Compared with traditional methods, digital content and tools can log student activities in great detail and provide huge opportunities to monitor students' progress either in course or at the program level. The gathered data can also be used to generate feedback for students themselves, for example, on their study practices (Auvinen et al., 2015), and feedback for curriculum developers about hot and cold spots in their instructional design as well as cohort-level feedback about difficulties. For example, Code.org iterates on its curriculum based on both qualitative feedback, but also instrumentation of the use of it's online tools. However, applying these logging facilities needs careful ethical consideration in research. For example, the granularity of logging (e.g., submission level logging, key-level logging, even accessing information about other activities students are doing while working on their tasks) should be treated with care to avoid ethical conflicts.

6.2 Challenges in computing education tools research

Conducting research on learning technologies for computing education poses some unique challenges.

In many cases, tool development requires significant effort. Usability and efficiency considerations are crucial if the tool is to be used at scale in real course environments, possibly with large numbers of students. These is even true for running carefully controlled laboratory

studies, as there are a range of confounding factors in tool design that can mask the benefits of the tool. Ko et al. (2015) present a detailed guide for evaluating programming tools..

Publishing details of the functionality of new tools, supported only by small single-cohort satisfaction analyses, is becoming increasingly difficult with many conferences and journals, which require more rigorous evaluation studies. Thus the time (and effort) between deployment of a new tool and dissemination of its features and advantages has increased significantly. Papers describing such systems *per se* may be publishable in other venues (e.g. Software Engineering, Programming, or CHI conferences and workshops) but for that they often need to make a significant, novel, technical contribution in the domain of the publication, which is not always clearly the case. A similar problem arises when an existing tool is re-implemented for use in a different programming language, such as DrJava (Allen et al., 2002), derived from DrScheme which was itself originally reported in a symposium on programming languages (Findler et al., 1997).

Sustaining tool research is another challenge. Tools are often designed and implemented as a part of a fixed-term research project, or a PhD student's work, and there is no one to continue maintenance and development after the project funding finishes or the thesis is submitted and the student has moved on. Numerous tools have suffered such a fate, and are therefore inaccessible to a wider audience or for continuing research. Obviously for any teacher or institution, adopting a tool that has unclear prospects of long-term support is a significant risk. Bugs might not get fixed, hosted services might terminate at short notice, and support may not be available. Some tools lack features which are required for widespread use, such as comprehensive support for internationalisation (e.g. character set and interface translations) or accessibility features for students with disabilities. The latter is a particular issue for tools with custom graphical interfaces, which take them beyond the reach of the facilities provided as standard by the underlying platform. Although it's not often considered in an educational setting, there may also be issues around ownership or licensing of intellectual property in work created using a tool, especially in cases where the tool is inextricably linked with students' work (e.g. Scratch or Greenfoot).

A problem for tool sustainability is that technologies evolve rapidly. This applies both to the technologies the tools use, as well as those they teach. While ten years ago many tools were delivered as Java applets or Flash applications, current web browsers no longer support these technologies and so the tools are inaccessible. The same problems apply to tools implemented as plugins for professional tools (e.g. Eclipse or NetBeans) with rapidly changing APIs. Old versions may simply cease working when platforms change, and therefore constant updating or even re-implementations are needed. This is work which has little scientific value, even though from practical point of view it may be essential. Tools which aim to teach rapidly evolving topics or languages such as Java or Python, or libraries like JUnit are also at the mercy of changes in their infrastructure.

Only a few tools have been able to reach a state where there is an established research group, or a dedicated individual, or external funding, which can ensure their long-term future. Examples include BlueJ, Web-CAT, jGRASP, DrRacket, Scratch, and Alice. Sometimes the development team has been able to build a business which develops and supports the tool by, for example, charging for enhanced support or back-end services. Sometimes a large enough developer community has emerged, providing the resources to maintain and further develop the tool. Some tools have also formed successful user communities which provide valuable feedback for developers, as well as showcases and discussion forums which can serve as peer and developer support for users (Roque at al., 2012).

Finally, research on tools often posses some common weaknesses. Many papers motivate the tool development by a teacher's observations, and these observations may or may not reflect the students actual learning problems. It would be valuable to demonstrate evidence of learning or studying problems, generate a hypothesis how the problem can be addressed, and build the tool as a test machine for the hypothesis. Secondly, another related aspect is that tools research is frequently disconnected to any educational or psychological theories about students' learning or behavior. Such theories could direct future research, by providing more explicit arguments for the observed learning problems, generated hypotheses and interpretation of empirical results. The field needs theories specific to programming and computing to help build a more robust understanding of how tools mediate learning.

References

Allen, E., Cartwright, R. and Stoler, B., 2002, February. DrJava: A lightweight pedagogic environment for Java. In *ACM SIGCSE Bulletin* 34(1), pp. 137-141. ACM.

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

Ala-Mutka, K. and Jarvinen, H.M., 2004, August. Assessment process for programming assignments. In *Proceedings. IEEE International Conference on Advanced Learning Technologies, 2004.* (pp. 181-185). IEEE.

Aleven, V., Myers, E., Easterday, M. and Ogan, A., 2010, April. Toward a framework for the analysis and design of educational games. In *Digital Game and Intelligent Toy Enhanced Learning (DIGITEL), 2010 Third IEEE International Conference on* (pp. 69-76). IEEE.

Ali, N.H., Shukur, Z. and Idris, S., 2007. Assessment system for UML class diagram using notations extraction. *International Journal on Computer Science Network Security*, *7*, pp.181-187.

Alvarado, C., Morrison, B.B., Ericson, B., Guzdial, M., Miller, B. and Ranum, D.L., 2012. Performance and use evaluation of an electronic book for introductory Python programming.

Auvinen, T., Hakulinen, L. and Malmi, L., 2015. Increasing students' awareness of their behavior in online learning environments with visualizations and achievement badges. *IEEE Transactions on Learning Technologies*, *8*(3), pp.261-273.

Baecker, R.M. and Sherman, D., 1981. Sorting Out Sorting. narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81 and excerpted in ACM SIGGRAPH Video Review #7, 1983.

Benford, S., Burke, E., Foxley, E., Gutteridge, N. and Zin, A.M., 1994. Ceilidh as a course management support system. *Journal of Educational Technology Systems*, *22*(3), pp.235-250.

Ben-Ari, M., Bednarik, R., Levy, R.B.B., Ebel, G., Moreno, A., Myller, N. and Sutinen, E., 2011. A decade of research and development on program animation: The Jeliot experience. *Journal of Visual Languages & Computing*, *22*(5), pp.375-384. www.cs.joensuu.fi/jeliot

Brown, M. H., & Sedgewick, R., 198). A system for algorithm animation. In *SIGGRAPH Computer Graphics* 18(3), pp. 177-186. ACM.

Bruce, K.B., Danyluk, A. and Murtagh, T., 2001, February. A library to support a graphics-based object-first approach to CS 1. In *ACM SIGCSE Bulletin,* 33(1), pp. 6-10. ACM.

Brusilovsky, P., Edwards, S., Kumar, A., Malmi, L., Benotti, L., Buck, D., Ihantola, P., Prince, R., Sirkiä, T., Sosnovsky, S. and Urquiza, J., 2014, June. 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). ACM.

Cardell-Oliver, R. and Doran Wu, P., 2011, June. UWA Java tools: harnessing software metrics to support novice programmers. In *Proceedings of the 16th annual joint conference on Innovation & Technology in Computer Science Education* (pp. 341-341). ACM.

Charters, P., Lee, M. J., Ko, A. J., & Loksa, D. 2014, March. Challenging stereotypes and changing attitudes: the effect of a brief programming encounter on adults' attitudes toward programming. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 653-658). ACM.

Connolly, T.M., & Stansfield, M.H. 2006. Enhancing eLearning: Using Computer Games to Teach Requirements Collection and Analysis. WG HCI & UE of the Austrian Computer Society.

Cooper, S., Dann, W. and Pausch, R., 2000, April. Alice: a 3-D tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, 15(5), pp. 107-116. Consortium for Computing Sciences in Colleges.

Cooper, S., Dann, W., & Pausch, R. 2003, February. Teaching objects-first in introductory computer science. In *ACM SIGCSE Bulletin*, 35(1), pp. 191-195. ACM.

Crescenzi, P., 2010. Alvie 3.0. https://sites.google.com/site/alviehomepage/alvie3

Davies, S., Polack-Wahl, J.A. and Anewalt, K., 2011, March. A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 625-630). ACM.

Dekeyser, S., de Raadt, M., & Lee, T. Y., 2007, March. Computer assisted assessment of SQL query skills. In *Proceedings of the eighteenth conference on Australasian database-Volume 63* (pp. 53-62). Australian Computer Society, Inc..

Diehl, S., 2007. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media.

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

Eckerdal, A., Kinnunen, P., Thota, N., Nylén, A., Sheard, J., & Malmi, L., 2014, June. Teaching and learning with MOOCs: computing academics' perspectives and engagement. In *Proceedings of the 2014 conference on Innovation & Technology in Computer Science Education* (pp. 9-14). ACM.

Edwards, S.H., 2003, October. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 148-155). ACM.

Edwards, S.H., Kandru, N. and Rajagopal, M., 2017, August. Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 65-73). ACM.

Eisenstadt, M., 1997. My hairiest bug war stories. *Communications of the ACM*, *40*(4), pp.30-37.

Ericson, B. J., Guzdial, M. J., & Morrison, B. B., 2015, July. Analysis of interactive features designed to enhance learning in an ebook. In *Proceedings of the eleventh annual International Conference on International Computing Education Research* (pp. 169-178). ACM.

Ericson, B. J., Rogers, K., Parker, M., Morrison, B., & Guzdial, M., 2016, August. 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). ACM.

Felleisen, M., Findler, R.B., Flatt, M. and Krishnamurthi, S., 1998. The DrScheme project: an overview. *ACM Sigplan Notices*, 33(6), pp.17-23.

Findler, R.B., Flanagan, C., Flatt, M., Krishnamurthi, S. and Felleisen, M., 1997, September. DrScheme: A pedagogic programming environment for Scheme. In *International Symposium on Programming Language Implementation and Logic Programming* (pp. 369-388). Springer, Berlin, Heidelberg.

Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. and Felleisen, M., 2002. DrScheme: A programming environment for Scheme. *Journal of functional programming*, *12*(2), pp.159-182.

Fisker, K., McCall, D., Kölling, M. and Quig, B., 2008, June. Group work support for the BlueJ IDE. In *ACM SIGCSE Bulletin*, 40(3), pp. 163-168. ACM.

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*, pp. 22-40.

Higgins, C., Symeonidis, P., & Tsintsifas, A., 2002. The marking system for CourseMaster. *ACM SIGCSE Bulletin*, *34*(3), pp. 46-50.

Gee, J.P., 2014. What video games have to teach us about learning and literacy. Macmillan.

Gómez-Albarrán, M., 2005. The teaching and learning of programming: a survey of supporting software tools. *The Computer Journal*, 48(2), pp. 130-144.

Guo, P. J., 2013, March. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 579-584). ACM. www.pythontutor.com

Haaranen, L., 2017. Programming as a Performance: Live-streaming and Its Implications for Computer Science Education. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (ITiCSE '17). ACM, New York, NY, USA, pp. 353-358.

Haaranen, L., Duran, R., 2017. Link Between Gaming Communities in YouTube and Computer Science. In *Proceedings of the 9th International Conference on Computer Supported Education* (CSEDU), pp. 17-24.

Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L., 2012, September. 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). ACM.

Hollingsworth, J., 1960. Automatic graders for programming classes. *Commun. ACM,* 3(10), pp. 528-529.

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

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). ACM.

Jackson, D., & Usher, M., 1997, March. Grading student programs using ASSYST. In *ACM SIGCSE Bulletin*, 29(1), pp. 335-339. ACM.

Joy, M., Griffiths, N., & Boyatt, R., 2005. The boss online submission and assessment system. *Journal on Educational Resources in Computing (JERIC)*, *5*(3), p. 2.

Karavirta, V., Korhonen, A., & Malmi, L., 2006. On the use of resubmissions in automatic assessment systems. *Computer science education*, *16*(3), pp. 229-240.

Karavirta, V., & Shaffer, C. A., 2013, July. JSAV: the JavaScript algorithm visualization library. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (pp. 159-164). ACM.

Kelleher, C. and Pausch, R., 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, *37*(2), pp.83-137.

Kemeny, John G.; Kurtz, Thomas E., 1964. *Basic: a manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System* (1st ed.). Hanover, N.H.: Dartmouth College Computation Center.

Kim, A.S. and Ko, A.J., 2017, March. A pedagogical analysis of online coding tutorials. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 321-326). ACM.

Ko, A.J., Latoza, T.D. and Burnett, M.M., 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, *20*(1), pp.110-141.

Kölling, M., Quig, B., Patterson, A. and Rosenberg, J., 2003. The BlueJ system and its pedagogy. *Computer Science Education*, *13*(4), pp.249-268.

Lancaster, T., & Culwin, F., 2004. A comparison of source code plagiarism detection engines. *Computer Science Education*, *14*(2), pp. 101-112.

Lee, M.J., Bahmani, F., Kwan, I., LaFerte, J., Charters, P., Horvath, A., Luor, F., Cao, J., Law, C., Beswetherick, M. and Long, S., 2014, July. Principles of a debugging-first puzzle game for computing education. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on* (pp. 57-64). IEEE.

Lee, M. J., & Ko, A. J., 2015, July. Comparing the effectiveness of online learning approaches on CS1 learning outcomes. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research,* (pp. 237-246). ACM.

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), pp. 267.

Malmi, L., Karavirta, V., Korhonen, A., & Nikander, J., 2005. Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *Journal on Educational Resources in Computing (JERIC)*, *5*(3), p. 7.

Mason, R., Cooper, G. and de Raadt, M., 2012, January. Trends in introductory programming courses in Australian universities: languages, environments and pedagogy. In *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123* (pp. 33-42). Australian Computer Society, Inc..

McIver, L., 2002, June. Evaluating languages and environments for novice programmers. In *Fourteenth Annual Workshop of the Psychology of Programming Interest Group (PPIG 2002), Brunel University, Middlesex, UK*.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M., 2013. Learning computer science concepts with scratch. *Computer Science Education*, *23*(3), pp. 239-264.

Miller, B.N. and Ranum, D.L., 2012, July. Beyond PDF and ePub: toward an interactive textbook. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (pp. 150-155). ACM.

Miljanovic, M.A. and Bradbury, J.S., 2017, August. RoboBUG: A Serious Game for Learning Debugging Techniques. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 93-100). ACM.

Mitrovic, A., Martin, B., & Mayo, M., 2002. Using evaluation to shape ITS design: Results and experiences with SQL-Tutor. *User Modeling and User-Adapted Interaction*, *12*(2), pp. 243-279.

Naps, T. L., Eagan, J. R., & Norton, L. L., 2000, May. JHAVÉ—an environment to actively engage students in Web-based algorithm visualizations. In *ACM SIGCSE Bulletin*, 32(1), pp. 109-113. ACM.

Naps, T.L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Velázquez-Iturbide, J.Á., 2002, June. Exploring the role of visualization and engagement in computer science education. In *ACM Sigcse Bulletin,* 35(2), pp. 131-152. ACM.

Nelson, G. L., 2017, August. Comprehension-First Pedagogy and Adaptive, Intrinsically Motivated Tutorials. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 287-288). ACM.

Nelson, G. L., Xie, B., & Ko, A. J., 2017, August. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 2-11). ACM.

Parker, M. C., Rogers, K., Ericson, B. J., & Guzdial, M., 2017, August. Students and Teachers Use An Online AP CS Principles EBook Differently: Teacher Behavior Consistent with Expert Learners. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 101-109). ACM.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. and Kafai, Y., 2009. Scratch: programming for all. *Communications of the ACM*, *52*(11), pp.60-67.

Redish, K. A., & Smyth, W. F., 1986. Program style analysis: A natural by-product of program compilation. *Communications of the ACM*, *29*(2), pp. 126-133.

Robinett, W. and Grimm, L., 1982. Rocky's Boots/Robot Odyssey. *The Learning Company*.

Rodger, S. H., 2002, June. Using hands-on visualizations to teach computer science from beginning courses to advanced courses. In *Second Program Visualization Workshop* (pp. 103-112).

Rodger, S. H., & Finley, T. W., 2006. *JFLAP: an interactive formal languages and automata package*. Jones & Bartlett Learning.

Rosales, F., García, A., Rodríguez, S., Pedraza, J. L., Méndez, R., & Nieto, M. M., 2008. Detection of plagiarism in programming assignments. *IEEE Transactions on Education*, *51*(2), pp. 174-183.

Ibrahim, R., Yusoff, R.C.M., Omar, H.M. and Jaafar, A., 2010. Students perceptions of using educational games to learn introductory programming. *Computer and Information Science*, *4*(1), p.205.

Rößling, G. and Vellaramkalayil, T., 2009. A visualization-based computer science hypertextbook prototype. *ACM Transactions on Computing Education (TOCE)*, *9*(2), p.11.

Rößling, G., & Freisleben, B., 2002. ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages & Computing*, *13*(3), pp. 341-354.

Roque, R., Kafai, Y. and Fields, D., 2012, June. From tools to communities: Designs to support online creative collaboration in Scratch. In *Proceedings of the 11th International Conference on Interaction Design and Children* (pp. 220-223). ACM.

Saikkonen, R., Malmi, L., & Korhonen, A., 2001, June. Fully automatic assessment of programming exercises. In *ACM Sigcse Bulletin*, 33(3), pp. 133-136. ACM.

Sheard, J., Eckerdal, A., Kinnunen, P., Malmi, L., Nylén, A., & Thota, N., 2014, November. MOOCs and their impact on academics. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (pp. 137-145). ACM.

Sheard, J., Simon, S., Hamilton, M., & Lönnberg, J., 2009, August. Analysis of research into the teaching and learning of programming. In *Proceedings of the fifth international workshop on Computing education research workshop* (pp. 93-104). ACM.

Shukur, Z., Burke, E., & Foxley, E., 1999. The automatic assessment of formal specification coursework. *Journal of Computing in Higher Education*, *11*(1), pp. 86-119.

Simon, A., 2007. classification of recent Australasian computing education publications. *Computer Science Education*, *17*(3), pp. 155-169.

Simon, S., 2009. Informatics in Education and Koli Calling: a Comparative Analysis. *Informatics in Education*, *8*(1), p. 101.

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.

Sirkiä, T. and Haaranen, L., 2017. Improving online learning activity interoperability with Acos server. *Software: Practice and Experience*, *47*(11), pp.1657-1676.

Soflano, M., Connolly, T.M. and Hainey, T., 2015. An application of adaptive games-based learning based on learning style to teach SQL. *Computers & Education*, *86*, pp.192-211.

Sorva, J., & Sirkiä, T., 2010, October. UUhistle: a software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 49-54). ACM.

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), p. 15.

Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J.K. and Padua-Perez, N., 2006. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin*, *38*(3), pp.13-17.

Stasko, J.T., 1990. Tango: A framework and system for algorithm animation. *Computer*, *23*(9), pp.27-39.

Stasko, J. ed., 1998. *Software visualization: Programming as a multimedia experience*. MIT press.

Thorburn, G., & Rowe, G., 1997. PASS: An automated system for program assessment. *Computers & Education*, *29*(4), pp. 195-206.

Tillmann, N., Bishop, J., Horspool, N., Perelman, D. and Xie, T., 2014, June. Code hunt: searching for secret code for fun. In *Proceedings of the 7th International Workshop on Search-Based Software Testing* (pp. 23-26). ACM.

Uysal, M.P., 2016. Evaluation of learning environments for object-oriented programming: measuring cognitive load with a novel measurement technique. *Interactive Learning Environments*, *24*(7), pp.1590-1609.

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

Wirth, N., 1973, July. *The Programming Language Pascal (Revised Report)*. ETH Zürich. https://doi.org/10.3929/ethz-a-000814158

Wolz, U. and Koffman, E., 1999, June. simpleIO: a Java package for novice interactive and graphics programming. In *ACM SIGCSE Bulletin*, 31(3), pp. 139-142. ACM.

Yan, A., Lee, M.J. and Ko, A.J.,2017. Predicting Abandonment in Online Coding Tutorials. *IEEE Symposium on Visual Languages and Human-Centered Computing (VL/HCC)*, pp. 191-199.