Survey of software visualization systems to teach message-passing concurrency in secondary school

Libert, Cédric; Vanhoof, Wim

Published in:
Highlights of Practical Applications of Cyber-Physical Multi-Agent Systems - International Workshops of PAAMS 2017, Proceedings

DOI:
10.1007/978-3-319-60285-1_33

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Survey of software visualization systems to teach message-passing concurrency in secondary school

Cédric Libert and Wim Vanhoof

University of Namur, Belgium

Abstract. In this paper, we compare 27 software visualization systems according to 8 criteria that are important to create an introduction to programming course based upon message passing concurrency.

Keywords: message passing, concurrency, teaching

Introduction

In Europe, more and more countries want to teach programming to primary and secondary school students. A 2014 report confirms that 15 European countries either already do or were about to do so [8]. The first programming course is an important issue in the IT curriculum. It is the core of many other IT courses, and it improves essential skills such as problem solving, abstraction and decomposition [3, p.41].

This preliminary study aims at finding software visualization systems (SVS) to use in an introductory programming course with 16- to 18-year-old students. This course will have new content, concurrent programming, and a corresponding teaching method, the microworld approach. This method consists in integrating a mini programming language into an SVS where agents show the execution of the code and where the teacher can add new programming concepts progressively. To implement this course, we need to find an SVS that fits with concurrency. The experimentation in situ costs a lot of time, so this preliminary work is a first step. Here we intend to justify, in light of the literature, the criteria used to evaluate SVSs.

In section 1, we justify the benefits of a course based on concurrency and an SVS. We present a particularly suitable concurrency model to teach, namely message passing, and show how it helps to improve programming skills and computational thinking. We then define the concept of an SVS and describe the advantages of such systems for teaching. In particular, we talk about one particular kind of system: microworlds. In section 2 we define some criteria for comparing SVSs with each other, build a comparison table of 27 SVSs and use it to categorize and compare existing systems. In the last section we select two good candidates for our course, and we explain why they are close to what we need and how we will manage to make them fulfil our requirements.

1 Why a concurrency course with a SVS?

The purpose of our research is to evaluate the efficiency, in terms of the improvement of programming skills and computational thinking, of a first programming course based on message passing. We want this course to use the microworld pedagogical approach [41], based on an SVS. In this section, we justify these choices.

1.1 Advantages of the message passing concurrency model

Teaching concurrency in an introductory course is usually seen as harder than teaching sequential programming [49,10,12]. But the students live in a concurrent world, so some authors conjecture that this technique may be more intuitive to them [49,10,12], at least with an appropriate teaching method. Furthermore, for some tasks, concurrency allows an easier decomposition into subproblems [49].

The 2013 ACM Computer Science Curricula report [3, p.44] is in favour of teaching concurrency starting in the first programming course because evolution in three main IT fields (hardware, software and data) requires concurrency. At the hardware level, multicore processors make it possible to physically execute some instructions in parallel and make the most of concurrent programming. In software, reactive user interfaces need concurrency to allow many events to happen and to be dealt with at the same time. Concerning data, thay are so abundant that they need distributed storage and concurrent processing on different computers. Therefore, according to the ACM, teaching needs to support these evolutions, especially with concurrency early in the curriculum.

Message passing, a simple concurrent paradigm We present and compare two main concurrency models: message passing and shared-state concurrency. We focus on the former, because the latter, mostly used in imperative and object-oriented languages, has some drawbacks we hope to avoid. Indeed, shared-state concurrency is based on the concepts of threads, interacting by modifying shared variables and protected by some mutual exclusion mechanism to avoid race conditions and data corruption. This way to deal with concurrency is very low-level, which doesn't appear to be efficient in the learning process. This concurrency model seems harder to understand by students [39] than message passing and is considered to be a pessimistic concurrency control, which requires a lot of effort to organize.

Message passing concurrency is a paradigm encompassing the three concepts of higher order functions, threads and ports (unidirectional communicating channels). A program consists in a set of concurrent agents, each one in its own thread. They communicate through the ports with messages containing arbitrarily complex data. They react to messages according to their behaviour, which is a function. Programming in this paradigm consists mostly in defining concurrent agent behaviours and the messages that they send and receive asynchronously.

One of the first languages to exploit this paradigm was Erlang [7]. This language, used in some highly distributed applications such as WhatsApp [14] and

some parts of GitHub [31], showed the advantages of message passing concurrency over the shared state concurrency model. The encapsulation of code into isolated but communicating agents avoids programming problems due to shared variables, because each agent is the only one that can modify his own state and, since they are higher level entities, the students do not need to understand how the machine works in order to be able to use them.

Message passing to improve programming skills The first programming paradigm taught has a large impact on the mental representation of the learner, and thus on how he will be able to learn programming techniques. Some authors show, for instance, the differences between procedural and object-oriented approaches in tackling a first course. Furthermore, according to White [53], there may even be interferences between two paradigms. For instance, learning object-oriented after procedural programming seems harder for students than the other way round.

The idea of concurrency in the first programming course dates back to the 90's. Feldman and Bachus show novices can learn concurrency [17]. Lynn Andrea Stein thinks we need to code in a different way, shifting from the "computation as calculation" paradigm, where a program is a function, to the "computation as interaction" metaphor [21], where a program is a community of concurrent agents communicating with each other by exchanging messages. This model also corresponds better with high level systems such as operating systems and the internet [48]. Furthermore, in 1997, Moström et al. [35] showed on a small sample of 8 novices that the novices found it easier to use concurrent language than sequential language to solve the problems they submitted. This doesn't show that concurrency is easier, because it is hard to generalize from this experiment, but that concurrency, and message passing in particular, may be easy to use, depending on the problem to be solved.

Message passing to improve computational thinking According to Wing, computational thinking is a skill involving problem solving, system design and human behaviour understanding [55]. Thus it needs important programming concepts. Computational thinking allows people to answer the questions: how hard is a particular problem to solve ? What is the best way to solve it ? Furthermore, computational thinking makes it possible to reformulate a hard problem in order to be able to solve it with reduction, composition, etc. It is also an abstraction skill, i.e. the "replacement of a complex and detailed real-world situation [with] an understandable model within which we can solve a problem" [4].

Some researchers [11,22] and the ISTE (International Society for Technology in Education) [25] formalized computational thinking as the following set of skills: abstracting and generalizing, automating and repeating actions, understanding and using concurrency, decomposing problems, handling conditional structures, using symbols to represent data, processing data in a systematic way, defining algorithms and procedures, knowing the efficiency and the perfor-

mance constraints of a solution, debugging and systematically detecting errors and executing simulations.

Very few papers describe empirical results about expanding computational thinking in terms of these skills with programming. According to most of them, computational thinking skills do not increase with the ability to program [52,23,51,18]. But they tend to focus on imperative, object-oriented or event-based (Scratch) paradigms, without really focusing on concurrency. For each of these skills, message passing concurrency should be able to increase computational thinking.

Indeed, message passing concurrency consists in writing automatic procedures to solve problems. It makes it possible to abstract, generalize and automatize. Thus, the concurrency concept makes it possible to get a good idea of parallelism and decomposition into independent subproblems. Message passing makes the data processing and the information representation obvious. And the conditional structures are used very early to define the agent behaviours. Finally, the use of a software visualization system, that fits completely with concurrency, helps to understand performance, simulation and debug.

1.2 Software visualization systems and teaching

A software visualization system is a pedagogical tool whose purpose is to help students in a programming class to learn by addressing their common difficulties [47]. The SVS is supposed to show different programming concepts dynamically by linking code and its execution to visual events. It makes it easier for the students to understand the notional machine, the abstract computer corresponding to the particular paradigm or language they use. It makes it easily to see the step by step execution of the program in order to trace and check the states.

Sorva et al. created a taxonomy of SVSs with three main categories [47]. We propose to extend this taxonomy with the programming game subcategory. We end up with this taxonomy: program animation (textual language generating visual animations), visual programming (visual language, often made of blocks that the student combine to create the program), algorithm visualization systems (algorithms are written as a flowchart) and programming games (subcategory of program animation consisting in making some actors move or act to go through a predefined game).

Advantages of software vizualisation systems Software visualization systems offer some advantages over textual-only programming languages.

First, the visual nature of these systems makes it easier to understand the programs. Indeed, the human perception requires less translation to represent visual concepts than textual ones [46].

A second advantage is that students feel involved in the system [38], so they tend to assimilate concepts easier. There are some positive experiments with SVSs like App Inventor used by college students [45,24], Scratch [32,19] or Alice [36].

A third advantage is that it makes concurrency and step-by-step debugging easier [37] thanks to the visual support. The former is obviously important for us, and some studies show that such graphic systems tend to encourage the use of concurrency [5] and have been used by professionals in order to more easily implement concurrent and parallel system [57]. The latter, the debugger, is useful for two reasons. Some teachers, such as Cross et al. [16], use the debugger as a teaching tool to show step by step execution of a program to students. But students can also use it to develop some useful skills [15,30], as long as they have good debugging strategies. We also saw that debugging is part of the computational thinking aptitudes described in section 1.1.

Microworld pedagogical approach Programming is hard to teach, because it requires many skills and all relevant concepts are intertwined. Furthermore, students usually see this course as boring and hard, especially when the language is not specifically designed for teaching and when it uses an unnatural syntax. The microworld pedagogical approach, based on SVSs, solves these problems.

A microworld is "a subset of reality or a constructed reality whose structure matches that of a given cognitive mechanism so as to provide an environment where the latter can operate effectively" [42, p.204]. This improves students' exploration and understanding of new concepts. In a programming course, a microworld is composed of a programming language and an SVS. Both are usually integrated in a unified interface with a script editor and a visualization window.

The main advantage of this kind of environment is that it deals with many pedagogical problems that arise when using a traditional language. Stelios Xinogalos [56] and Linda Mciver [33] note some of these problems. First, there are too many instructions in programming languages, compared to microworlds where the vocabulary is more limited and progressively enriched. Second, students tend to focus on syntax of full languages, while microworlds usually offer a simpler syntax. Third, the execution of the program is usually hidden, as opposed to the microworld where agents act according to the script. Usually, students seem to understand programming concepts better with the use of microworlds, according to Selios Xinogalos [56].

2 Comparison of software visualization systems

We now define exactly what visualization system we want. We first define and describe comparison criteria and use them to build a comparison table containing 27 SVSs. We then cluster the table into four categories of systems. We finally select the systems employing concurrency, message passing and microworlds.

2.1 Criteria

In order to select the software visualization system that could help us to build a first programming course based on message passing concurrency, we define some comparison criteria based partially on criteria found in the literature [28,13,29] and on the need we described in section 1.

Extrinsic criteria These criteria, such as the release and last update dates, do not concern the nature of the system or the language itself. The release and last update dates are important to us, as we prefer to use a maintained and up-to-date system with modern graphics. We also focus on the license type, because we favour open source software because it does not restrict the installation of software on many computers in a secondary school and at home, and it reduces the cost and is potentially able to be adapted to our needs.

Visual Nature This criterion constitutes the main advantage of a software visualization system over a textual one for learning, as we saw in section 1.2. We identify categories of visualization environments, and evaluate visual functionalities. The categories of SVSs we identify are adapted from the taxonomy developed by Sorva et al. [47] and were explained in section 1.2.

Languages, paradigms and concepts Each programming visualization system helps to teach a particular language in a particular paradigm defined by a set of concepts. The language used may be an already existing language, like Java or Python, or a language designed for this particular system. Many languages that we encountered in visualization systems are a mix of some of these imperative, object-oriented, event-based, functional and logic paradigms.

Some of these paradigms are not suitable for teaching based on concurrency: imperative and object oriented paradigms, as we saw in section 1.1, seem to be less efficient for teaching concurrency because they usually implement the shared state concurrency model; the event-based paradigm, although conceptually appealing, does not allow asynchronous message delivery in the way described in section 1, and sometimes limits the message structure to something very simple, like a string or an atom.

Despite all of this, we also want to know, independently of the paradigm, whether or not each of these languages is concurrent, because this is the core concept we want to teach.

Debugger As seen in section 1.2, easy access to a debugger is a double advantage for the teacher demonstrating the execution of the program and for students learning to diagnose programming problems in a visual way. We thus prefer systems that come with a debugger.

Table 1. A comparison of 27 software visualization systems in four categories: concurrent visual programming (CVP), sequential visual programming (SVP), concurrent program animation (CPA) and sequential program animation (SPA)

	Name	Visual Nature	Concurrency	Paradigm	Debugger	Language	License	Creation	Update
CVP	App Inventor	visual programming language	yes	Event-based	yes	Blockly	open source	2010	2017
	Snap	visual programming language	yes	Event-based, functional	yes	Snap	open source	2011	2015
	Starlogo nova	visual programming language	yes	Event-based,object-oriented, agents	yes	Starlogo nova	open source	1996	2014
	Kedama	visual programming language	yes	Object-oriented	yes	Squeak/Smalltalk	open source	2005	?
	Scratch	visual programming language	yes	Event-based	no	Scratch	open source	2006	2015
	Blockly	visual programming language	yes	Imperative	no	Blockly	open source	2012	2017
	Alice 3	visual programming language	yes	Object-oriented	no	Java	open source	1998	2016
	ToonTalk	visual programming language	yes	Concurrent constraint programming	yes	Toontalk	proprietary	1995	2009
	AgentSheets	visual programming language	yes	Active objects	no	Visual AgenTalk	proprietary	1989	2012
SVP	Etoys	visual programming language	no	Object-oriented	yes	Squeak/Smalltalk	open source	1996	2012
	Amici	visual programming language	no	Imperative	?	Amici	open source	2011	2013
	Raptor	algorithm visualization system	no	Imperative	no	pseudocode	proprietary	2015	2016
	LARP	algorithm visualization system	no	Imperative	no	pseudocode	proprietary	2004	2008
CPA	Robot Code	programming game	yes	Object-oriented	yes	Java	open source	2001	2017
	Karel J Robot	program animation system	yes	Object-oriented	yes	Java	open source	2005	2016
	Processing	program animation system	yes	Object-oriented	yes	Java	open source	2001	2016
	Greenfoot	program animation system	yes	Object-oriented	oui	Java	open source	2004	2015
	NetLogo	program animation system	yes	Agent-based	no	Logo + agents	open source	1999	2015
	Multilogo	program animation system	yes	Object-oriented and concurrent	?	Logo+LEGO	open source	1990	?
	Microworlds Ex	program animation system	yes	?	?	Microworlds	proprietary	2003	2007
SPA	Guido van Robot	program animation system	no	Imperative	yes	Python	open source	2009	2010
	Karel++	program animation system	no	Object-oriented	yes	Karel++	open source	1997	1998
	Karel the Robot	program animation system	no	Imperative	no	Karel	open source	1981	2000
	Löve	program animation system	no	Imperative	no	Lua	open source	2008	2016
	Kojo	program animation system	no	Object-oriented, functional	no	Scala	open source	2010	2015
	Logo	program animation system	no	Imperative	?	Logo	open source	1967	2002
	Jeroo	program animation system	no	Object-oriented	yes	Jeroo (like Java)	proprietary	2004	2014

2.2 Categories

Table 1, contains the result of applying the defined criteria to 27 software visualization systems. We group them in categories based on two main criteria:

- Do users need to write textual code? This refers to a dichotomy among program animation system, where students need to write code themselves, and visual programming, where they only move code blocks;
- Can users use them to write concurrent programs or only sequential programs? Since we want to teach concurrency, we prefer that they can.

Sequential (no concurrency) program animation systems (SPA) This category gathers some of the oldest systems for teaching programming, like Logo and Karel. These systems aren't up-to-date compared to other categories. Most them are imperative, but some are object-oriented. Kojo [40], although clearly in this category, seems to be an outsider for two reasons. Firstly, although it isn't a visual programming language, the user doesn't have to write all the code himself, because she can click on predefined instructions to add them into the editor. Secondly, since it is based on Scala, it is also a functional language (a rare paradigm in this kind of visual system). Since Scala includes the Akka [9] library for concurrent actors, this system could become very interesting at some point for teaching concurrency if Akka was integrated into Kojo and the graphical interface. But there have been few experiments on the use of Kojo (and, generally, Scala) as a first language. The challenges of this language, as Regnell et al. write [43], are the error messages that are hard to understand and the type system (supposedly harder for slower learner).

Concurrent program animation systems (CPA) Theses systems are mostly based on Java and Logo, and most are object-oriented. One exception is Netlogo [50], where the agent-based paradigm prevails. This language adds concurrency to the old Logo language and has been used to model some complex systems such as patterns emerging in nature [54] or, more lately, street robbery [6]. Unfortunately, there are two drawbacks to this system. First of all, syntax seems a bit hard to learn for novices, because there are many keywords they need to know from the beginning in order to command agents. The second drawback concerns these agents: it seems impossible to define their behaviour, or to define new messages. There are four implemented agent types [1] and they can only react to some predefined "ask" instructions.

Sequential visual programming languages (SVP) This category contains three imperative and one object-oriented language based on Squeak, a SmallTalk dialect. We prefer not to use object-oriented and imperative languages because they make it harder to add a concurrency model based on message passing.

Concurrent visual programming languages (CVP) This category contains the most up-to-date systems and representatives of the most diverse paradigms. Though many of them look interesting, here we focus on the open source systems offering a debugger and implementing an inherently concurrent paradigm. The concurrent constraint programming language **ToonTalk** [26] allows users to handle not only concurrency concepts like actor spawning and termination and sending and receiving messages through unidirectional channels but also constraint/logic programming concepts such as clause, guards and body [27], in a 3D playful city microworld. Some experiments with ToonTalk were conducted with kindergarten children [34] but the author did not mention any statistical result. This language is a good candidate for us, but its target audience, as they mention on their website, is "children", so it might not be suitable for secondary school students. The event-based **App Inventor** for Android [2], unlike ToonTalk, has been successfully used by high school students [45,24] and university students [20]. This language aims at coding Android applications easily. Unfortunately, this language is not really close to the microworld approach we want to use, since it does not imply any agent acting in an environment, but handles sensors and panels of the phone directly. **Star Logo Nova** [44] is similar to NetLogo, but with blocks. Finally, **Snap!** is derived from Scratch, with agents whose behaviour has to be defined according to some event occurring (it has also influenced Scratch, whose procedure blocks derive from Snap!). Scratch was extended by adding higher order functions concept in order to allow students to become more familiar with functional programming. Unfortunately, as in Scratch, the "messages" that agents can send are limited to a simple label, and can only be broadcasted, which seems too limited for what we intend to do. Furthermore, since the reception of a message is an event, the agent stops the task he is currently executing when he receives a new one. There is no "mailbox" that allows agents to receive messages asynchronously.

Conclusion: which SVS to choose?

In this paper, we conjecture the importance of teaching concurrency properly in order to improve programming skills and computational thinking. We described software visualization systems because the microworld pedagogical approach is convenient to teach concurrency, and because a microworld is an SVS. We finally compared 27 SVSs with respect to six defined criteria and created a table in which we identified four categories, of which the most interesting for our needs seems to be the category of visual programming languages allowing concurrency. This category includes languages whose code is not textual but consists of blocks that the user has to put together to create the program, and where the languages have concurrency concepts. Two SVSs where identified as the best in this category, because they implement interesting paradigms and have a debugger: ToonTalk and Snap!. But are these systems able to address our needs?

Snap! is event-based. This means that its message transfer is not asynchronous. It also does not use a unidirectional channel from one agent to another,

but rather a broadcast system. Finally, messages may only contain labels, which is a great limitation because we want to be able to transfer complex data structures between agents in the message passing concurrency model.

ToonTalk seems good with its constraint based concurrency paradigm, including unidirectional channels, complex messages and concurrent actors. But since the target audience is children, secondary students might consider it too childish. So far, no experiment has used this system with teenagers.

The next steps after this preliminary work are to precisely evaluate how difficult it would be to add a proper message passing concurrency model to Snap! and to collect teenage students' perception of ToonTalk. Then we will be able to choose the language that fulfils the requirements of a secondary level introduction to programming course based on message passing concurrency and microworlds.

References

1. Netlogo user manual version 6.0. https://ccl.northwestern.edu/netlogo/docs/.
2. Abelson, H.: App inventor for android. Google Research Blog (2009)
3. ACM/IEEE-CS Joint Task Force on Computing Curricula: Computer science curricula 2013. Tech. rep., ACM Press and IEEE Computer Society Press (2013)
4. Aho, A.V., Ullman, J.D.: Foundations of Computer Science, C Edition. Computer Science Press / W. H. Freeman (1992)
5. Aivaloglou, E., Hermans, F.: How kids code and how we know: An exploratory study on the scratch repository. In: Proceedings of the 2016 ACM Conference on International Computing Education Research. pp. 53–61. ICER '16, ACM (2016)
6. Amrutha, S., Idicula, S.M.: Agent based simulation of street robbery. Department of computer science, Royal college of engineering and technology Thrissur, India (2014)
7. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
8. Balanskat, A., Engelhardt, K.: Computing Our Future: Computer Programming and Coding-Priorities, School Curricula and Initiatives Across Europe (2014)
9. Bonér, J., Klang, V., Kuhn, R., et al.: Akka library. http://akka.io
10. Brabrand, C.: Constructive Alignment for Teaching Model-Based Design for Concurrency, pp. 1–18. Springer Berlin Heidelberg (2008)
11. Brennan, K., Resnick, M.: New frameworks for studying and assessing the development of computational thinking. In: Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada. pp. 1–25 (2012)
12. Carro, M., Herranz, A., Mariño, J.: A model-driven approach to teaching concurrency. Trans. Comput. Educ. 13(1), 5:1–5:19 (2013)
13. Castillo-Barrera, F.E., Arjona-Villicana, P.D., Ramirez-Gamez, C.A., Hernandez-Castro, F.E., Sadjadi, S.M.: Turtles, robots, sheep, cats, languages what is next to teach programming? a future developer's crisis? In: Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer EngineeringMTDL. p. 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (2013)
14. Chechina, N., Hernandez, M.M., Trinder, P.: A scalable reliable instant messenger using the sd erlang libraries. In: Proceedings of the 15th International Workshop on Erlang. pp. 33–41. Erlang 2016, ACM (2016)

15. Chmiel, R., Loui, M.C.: Debugging: from novice to expert. ACM SIGCSE Bulletin 36(1), 17–21 (2004)
16. Cross, J., Hendrix, T.D., Barowski, L.A.: Using the debugger as an integral part of teaching cs1. In: Frontiers in Education. vol. 2, pp. F1G–F1G. IEEE (2002)
17. Feldman, M.B., Bachus, B.D.: Concurrent programming can be introduced into the lower-level undergraduate curriculum. SIGCSE Bull. 29(3), 77–79 (1997)
18. Fox, R.W., Farmer, M.E.: The effect of computer programming education on the reasoning skills of high school students. Frontiers in Education: Computer Science and Computer Engineering (FECS'11) (2011)
19. Franklin, D., Conrad, P., Boe, B., Nilsen, K., Hill, C., Len, M., Dreschler, G., Aldana, G., Almeida-Tanaka, P., Kiefer, B., Laird, C., Lopez, F., Pham, C., Suarez, J., Waite, R.: Assessment of computer science learning in a scratch-based outreach program. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education. pp. 371–376. SIGCSE '13, ACM (2013)
20. Gestwicki, P., Ahmad, K.: App inventor for android with studio-based learning. Journal of Computing Sciences in Colleges 27(1), 55–63 (2011)
21. Goldin, D., Wegner, P.: The interactive nature of computing: Refuting the strong church–turing thesis. Minds and Machines 18(1), 17–38 (2008)
22. Grover, S., Pea, R.: Computational thinking in k–12. a review of the state of the field. Educational Researcher 42(1), 38–43 (2013)
23. Gülbahar, Y., Kalelioğlu, F., et al.: The effects of teaching programming via scratch on problem solving skills: A discussion from learners' perspective. Informatics in Education-An International Journal (Vol13_1), 33–50 (2014)
24. Honig, W.L.: Teaching and assessing programming fundamentals for non majors with visual programming. In: Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education. pp. 40–45. ITiCSE '13, ACM (2013)
25. ISTE, CSTA: Nsf. computational thinking teacher resources, 2011
26. Kahn, K.: Toontalk tm—an animated programming environment for children. Journal of Visual Languages & Computing 7(2), 197–217 (1996)
27. Kahn, K.M.: From prolog and zelta to toontalk. In: ICLP. pp. 67–78 (1999)
28. KIPER, J.D., HOWARD, E., AMES, C.: Criteria for evaluation of visual programming languages. Journal of Visual Languages & Computing 8(2), 175 – 192 (1997)
29. Li, F.W., Watson, C.: Game-based concept visualization for learning programming. In: Proceedings of the Third International ACM Workshop on Multimedia Technologies for Distance Learning. pp. 37–42. MTDL '11, ACM (2011)
30. Lister, R.: Objectives and objective assessment in cs1. In: ACM SIGCSE Bulletin. vol. 33, pp. 292–296. ACM (2001)
31. Lutz, M.J.: The erlang approach to concurrent system development. In: Frontiers in Education Conference, 2013 IEEE. pp. 12–13. IEEE (2013)
32. Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M., Rusk, N.: Programming by choice: Urban youth learning programming with scratch. SIGCSE Bull. 40(1), 367–371 (2008)
33. McIver, L.: The effect of programming language on error rates of novice programmers. In: 12th Annual Workshop of Psychology of Programmers Interest Group (PPIG). pp. 181–192 (2000)
34. Morgado, L., Cruz, M.G.B., Kahn, K.: Working in toontalk with 4-and 5-year olds. In: International Association for Development of the Information Society-IADIS International Conference e-Society 2003. p. 988 (2003)
35. Moström, J.E., Carr, D.: Programming paradigms and program comprehension by novices. Luleå tekniska universitet (1997)

36. Mullins, P., Whitfield, D., Conlon, M.: Using alice 2.0 as a first language. J. Comput. Small Coll. 24(3), 136–143 (2009)
37. Myers, B.A.: Taxonomies of visual programming and program visualization. Journal of Visual Languages & Computing 1(1), 97–123 (1990)
38. Naps, T.L.: Jhavé: Supporting algorithm visualization. IEEE Computer Graphics and Applications 25(5), 49–55 (2005)
39. Ortiz, A.: Teaching concurrency-oriented programming with erlang. In: Proceedings of the 42nd ACM technical symposium on Computer science education. pp. 195–200. ACM (2011)
40. Pant, L., Pant, V., Pant, N.: Kojo homepage. http://www.kogics.net/kojo.
41. Papert, S.: Computer-based microworlds as incubators for powerful ideas. In: The Computer in the school : tutor, tool, tutee. Teachers College Press (1980)
42. Papert, S.: Computer-based microworlds as incubators for powerful ideas. The computer in the school: Tutor, tool, tutee pp. 203–210 (1980)
43. Regnell, B., Pant, L., Kogics, D.: Teaching programming to young learners using scala and kojo. LTHs Pedagogiska Inspirationskonferens 8, 4 (2014)
44. Resnick, M.: Starlogo: An environment for decentralized modeling and decentralized thinking. In: Conference companion on Human factors in computing systems. pp. 11–12. ACM (1996)
45. Roy, K.: App inventor for android: Report from a summer camp. In: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education. pp. 283–288. SIGCSE '12, ACM (2012)
46. Smith, D.C.: Pygmalion: a creative programming environment. Tech. rep., DTIC Document (1975)
47. Sorva, J., Karavirta, V., Malmi, L.: A review of generic program visualization systems for introductory programming education. Trans. Comput. Educ. 13(4), 15:1–15:64 (2013)
48. Stein, L.A.: Challenging the computational metaphor: Implications for how we think (1999)
49. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30(3) (2005)
50. Tisue, S., Wilensky, U.: Netlogo: Design and implementation of a multi-agent modeling environment. In: Proceedings of the Agent2004 Conference (2004)
51. Unuakhalu, M.F.: Enhancing problem-solving capabilities using object-oriented programming language. Journal of Educational Technology Systems 37(2), 121–137 (2008)
52. VanLengen, C., Maddux, C.: Does instruction in computer programming improve problem solving ability ? Journal of IS Education 12 (1990)
53. White, G., Sivitanides, M.: Cognitive differences between procedural programming and object oriented programming. Inf. Technol. and Management 6(4), 333–350 (2005)
54. Wilensky, U.: Modeling nature's emergent patterns with multi-agent languages. Citeseer
55. Wing, J.M.: Computational thinking. Commun. ACM 49(3), 33–35 (2006)
56. Xinogalos, S.: An evaluation of knowledge transfer from microworld programming to conventional programming. Journal of Educational Computing Research 47(3), 251–277 (2012)
57. ZHANG, K., HINTZ, T., MA, X.: The role of graphics in parallel program development. Journal of Visual Languages & Computing 10(3), 215 – 243 (1999)