Survey of Software Visualization Systems to Teach Message-Passing Concurrency in Secondary School

Cédric Libert(✉) and Wim Vanhoof

University of Namur, Namur, Belgium
cedric.libert@unamur.be

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

1 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 Sect. 2, 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 Sect. 3 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.

J. Bajo et al. (Eds.): PAAMS 2017 Workshops, CCIS 722, pp. 386–397, 2017.
DOI: 10.1007/978-3-319-60285-1_33

2 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.

2.1 Advantages of the Message Passing Concurrency Model

Teaching concurrency in an introductory course is usually seen as harder than teaching sequential programming [10,12,48]. But the students live in a concurrent world, so some authors conjecture that this technique may be more intuitive to them [10,12,48], at least with an appropriate teaching method. Furthermore, for some tasks, concurrency allows an easier decomposition into subproblems [48].

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 and Sivitanides [52], 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 [47]. Furthermore, in 1997, Moström and Carr [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 [54]. 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 performance 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 [18, 23,50,51]. 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.

2.2 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 [46]. 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 [46]. 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 [45].

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 [24,44], Scratch [19,32] 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 [56]. 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 Sect. 2.1.

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" [41, 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. Xinogalos [55] and 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 Xinogalos [55].

3 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.

3.1 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 [13,28,29] and on the need we described in Sect. 2.