Angeli, C., Voogt, J., Fluck, A., Webb, M., Cox, M., Malyn-Smith, J., & Zagami, J. (2016). A K-6 Computational Thinking Curriculum Framework: Implications for Teacher Knowledge. *Educational Technology & Society, 19* (3), 47–57.

A K-6 Computational Thinking Curriculum Framework: Implications for Teacher Knowledge

Charoula Angeli[1*], Joke Voogt[2], Andrew Fluck[3], Mary Webb[4], Margaret Cox[4], Joyce Malyn-Smith[5] and Jason Zagami[6]

[1]University of Cyprus, Cyprus // [2]University of Amsterdam, The Netherlands // [3]University of Tasmania, Australia // [4]King's College London, UK // [5]Education Development Center, USA // [6]Griffith University, Australia // cangeli@ucy.ac.cy // J.M.Voogt@uva.nl // Andrew.Fluck@utas.edu.au // mary.webb@kcl.ac.uk // mj.cox@kcl.ac.uk // jmsmith@edc.org // j.zagami@griffith.edu.au

[*]Corresponding author

ABSTRACT

Adding computer science as a separate school subject to the core K-6 curriculum is a complex issue with educational challenges. The authors herein address two of these challenges: (1) the design of the curriculum based on a generic computational thinking framework, and (2) the knowledge teachers need to teach the curriculum. The first issue is discussed within a perspective of designing an authentic computational thinking curriculum with a focus on real-world problems. The second issue is addressed within the framework of technological pedagogical content knowledge explicating in detail the body of knowledge that teachers need to have to be able to teach computational thinking in a K-6 environment. An example of how these ideas can be applied in practice is also given. While it is recognized there is a lack of adequate empirical evidence in terms of the effectiveness of the frameworks proposed herein, it is expected that our knowledge and research base will dramatically increase over the next several years, as more countries around the world add computer science as a separate school subject to their K-6 curriculum.

Keywords

Computational thinking curriculum, Pedagogical content knowledge, Technological pedagogical content knowledge, Teacher preparation, K-6

Introduction

In a world in which digital technology plays an important role in carrying out essential daily-life tasks, it is imperative individuals have the education, knowledge, and skills to critically understand the technological systems they use, as well as to be able to troubleshoot and problem solve when things go wrong (Wing, 2006; Czerkawski, 2015; National Research Council, 2010). Czerkawski (2015) argues the knowledge that individuals need to have in order to competently respond to the challenges of the 21st century goes beyond the acquisition of mere skills with immediate application, to knowledge with long-term value that will enable them to understand the basics of computer structures and practices. In essence, the society needs citizens who understand the true affordances of computers in terms of what they can and cannot do, so they themselves become effective authors/creators of computational tools. Wing (2006) broadened the idea of computation, and proposed that computational thinking should be considered as a basic skill taught across the curriculum. She defined computational thinking as the thought process of formulating and solving problems with the use of computers. According to Wing (2006), the teaching of computational thinking, as a basic skill across the school curriculum, will enable K-12 students to learn abstract, algorithmic and logical thinking, and be prepared to solve complex and open-ended problems.

How do we then prepare our students to develop the knowledge they need to survive and effectively cope with the technological challenges of the 21st century? As many educators strongly argued, this educational goal can be achieved by integrating computer science as a distinct discipline and a school subject in the K-12 curriculum (Barr & Stephenson, 2011; Fluck, Webb, Cox, Angeli, Malyn-Smith, Voogt, & Zagami, 2016; Goode, Chapman, & Margolis, 2012; Hazzan, Lapidot, & Ragonis, 2011; Tucker, Deek, Jones, McCowan, Stephenson, & Verno, 2003). Fluck et al. (2016) stated that there is a strong case for integrating computer science in the K-12 curriculum with arguments from both the educational and economic sectors. Succinctly, the educational case asserts that computer science: (a) develops and promotes a unique way of thinking about problems, namely computational thinking, that uses the power of logic, algorithm, abstraction, and precision; (b) empowers individuals to create new artifacts and to move from being consumers of technology to producers of technology; and (c) redefines the way learners think about other disciplines, and, this can have a major impact on teaching practices, such as, for example, interdisciplinary teaching in school. The economic case stresses the critical shortage of applicants in IT-related jobs, especially in Europe,

while at the same time the European Commission predicts that major European countries, such as UK, will need an additional 500,000 IT professionals by 2015 (Husing & Korte, 2010).

Adding computer science as a separate school subject to the core K-12 curriculum is, however, a complex issue that involves many legislative, administrative, political, and educational challenges. The latter are the focal point of this paper. In particular, there are two major educational challenges related to: (a) what computer science content to teach across different educational levels, and (b) what body of knowledge do teachers need to have to be able to teach the computer science curriculum. Over the years, a variety of computer science curricula, representing different views about what is important to teach in computer science and when, have been proposed in the literature and or enacted in different countries, such as UK, USA, Austria, Germany, Mongolia, Israel, Greece, Cyprus, and recently Australia. Well-known efforts in the United States are, amongst others, the *Computer Science Principles*, *Exploring Computer Science*, *Beauty and Joy of Computing*, *Project Lead the Way (PLTW)*, and *Code.org. Computer Science Principles* is part of a larger national effort in the United States, namely the CS 10K Project that aims to develop effective high school computing curricula enacted in 10,000 high schools taught by 10,000 well-prepared teachers by 2016. *Computer Science Principles* constitutes a framework of standards from which high school computer science courses can be built (Astrachan & Briggs, 2012). The framework is specified through a set of six Computational Thinking Practices (i.e., connecting computing, developing computational artifacts, analyzing problems and artifacts, abstracting, communicating, and collaborating), and a set of seven Big Ideas of computer science (i.e., creativity, abstraction, data and information, algorithms, programming, Internet, and global impact), and has been adopted by several high schools in the United States for developing computer science courses, such as the *Beauty and Joy of Computing*, *Code.org*, and *PLTW* (Astrachan & Briggs, 2012; desJardins, 2015). The *Beauty and Joy of Computing* course focuses on the Big Ideas of computing, and its main objective is to expose students to the beauty and joy of programming by engaging them in meaningful projects using the *Snap!* programming language. Similarly, *Code.org* is a high school course with lessons and programming projects around the seven Big Ideas of computing as well, whereas, the *PLTW* uses *Python* as its primary programming environment to expose students to different computational thinking projects.

Analogously, in various other countries similar initiatives have also been undertaken for introducing computer science to high school students (van Diepen, Perrenet, & Zwaneveld, 2011; Micheuz, 2008; Furber, 2012). Undoubtedly, during the last two decades, a lot of work has been done by the computer science education community in promoting computer science as a school subject in secondary education. Unfortunately, the same conclusion cannot be reached about the status of computer science in the elementary school curriculum (grades K-6, approximately from 6 to 12 years old).

A number of computer science education researchers have written about their concerns in regards to teaching computer science in K-6 (e.g., Armoni, 2012). These concerns are primarily linked to the incompatibility between abstraction, an essential process in computer science, and children's weakness to understand abstraction because of their very young age. Armoni (2012) explained that abstraction is an inherent component of computer science that is always encapsulated during the process of thinking about and automating a solution to a problem. From a Piagetian perspective, children before the age of seven cannot really understand concrete logic, whereas children between seven and eleven years old can solve problems that apply to concrete objects, but not problems that apply to abstract concepts or phenomena. Conversely, Gibson (2012) argued that high school is too late for exposing students to computer science for the first time, and stated that early exposure during kindergarten is necessary. In his research, Gibson (2012) found that young children can think abstractly when concrete reference systems are used to situate their thinking.

Recently, there has been much impetus in bringing computer science experiences to elementary school children (Kumar, 2014). Kumar (2014) wrote about the proliferation of app development startup companies that have targeted "*early childhood computing education as the next emerging frontier*" (p. 52), and about formal deliberative initiatives for developing computer science curricula for K-6 students. Succinctly, we acknowledge the effort by Prottsman (2014) who reported on the development of the *Thinkersmith* curriculum in 2011, which introduced a stand-alone set of unplugged activities for K-8 specifically designed to provide students with strong computer science foundations without using computers. Lessons in this curriculum, such as Binary Baubles, used materials found in games and crafts to teach authentic computer science concepts. In 2013, *Code.org* expanded on what *ThinkerSmith* created, and offered a 20-hour unplugged curriculum for grades K-8. After the wide adoption of this curriculum, in 2015 *Code.org* developed further the existing 20-hour unplugged curriculum, which now includes

more than 55 lessons. *CS Unplugged*, another unplugged computer science (CS) approach proposed by Bell, Witten, Fellows, Adams, and McKenzie (2015), is a collection of activities that teach computational thinking through engaging games and puzzles that use cards, string, crayons and lots of physical movement. Students learn about binary numbers and algorithms without using computer programming.

Clearly, early computing education is now at the forefront, and, studies toward this line of research are urgently needed in order to develop an informed body of knowledge about learning and teaching computer science in K-6. Accordingly, the authors propose a curriculum framework with a focus on promoting computational thinking skills for ages 6 to 12. While computational thinking is just one element of computer science, albeit an important one (Fluck et al., 2016), the authors suggest a curriculum for K-6 with an explicit focus on computational thinking, before covering more theoretical and applied concepts of computer science in secondary education. Particularly, this study sought to address the following questions: (a) what computational thinking skills should a curriculum promote in K-6? and (b) what knowledge do teachers need to have to be able to teach a computational thinking curriculum in K-6?

A definition of computational thinking

While the concept of computational thinking in education can be traced back to the work of Seymour Papert (Papert, 1980), Wing's (2006) article has rekindled the interest for promoting computational thinking in K-12. Other efforts aiming at developing a definition for computational thinking include, among others, the National Academy of Sciences workshop (National Research Council, 2010), the initiative undertaken by Furber (2012), and workshops organized by the Computer Science Teachers Association (CSTA) and the International Society for Technology in Education (ISTE).

Succinctly, the 2010 National Research Council's report differentiated computational thinking from computer literacy, computer programming, and computer applications (i.e., games), and broadened the term to include core concepts from the discipline of computer science, such as abstraction, decomposition, pattern generalization, visualization, problem-solving, and algorithmic thinking.

Similarly, Furber (2012) offered a concise definition of computational thinking as "the process of recognizing aspects of computation in the world that surrounds us, and applying tools and techniques from computer science to understand and reason about both natural and artificial systems and processes" (p. 29).

CSTA and ISTE, in collaboration with leaders from higher education, industry, and K-12 education, developed an operational definition of computational thinking as a problem-solving process that includes, but is not limited to, the following elements: (a) Formulating problems in a way that enables us to use a computer and other tools to help solve them; (b) Logically organizing and analyzing data; (c) Representing data through abstractions, such as, models and simulations; (d) Automating solutions through algorithmic thinking (i.e., a series of ordered steps); (e) Identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources; and (f) Generalizing and transferring this problem-solving process to a wide variety of problems.

Despite the fact that currently there is not one unanimous definition of computational thinking, it seems fair to conclude that, based on the literature reviewed in this study, researchers have come to accept that computational thinking is a thought process that utilizes the elements of abstraction, generalization, decomposition, algorithmic thinking, and debugging (detection and correction of errors). Abstraction is the skill of removing characteristics or attributes from an object or an entity in order to reduce it to a set of fundamental characteristics (Wing, 2011). While abstraction reduces complexity by hiding irrelevant detail, generalization reduces complexity by replacing multiple entities that perform similar functions with a single construct (Thalheim, 2000). Abstraction and generalization are often used together as abstracts are generalized through parameterization to provide greater utility. Decomposition is the skill of breaking complex problems into simpler ones (National Research Council, 2010). Algorithmic thinking is a problem-solving skill related to devising a step-by-step solution to a problem and differs from coding (i.e., the technical skills required to use a programming language) (Selby, 2014). Additionally, algorithmic notions of sequencing (i.e., planning an algorithm, which involves putting actions in the correct sequence), and algorithmic notions of flow of control (i.e., the order in which individual instructions or steps in an algorithm are evaluated) are

also considered important elements of computational thinking (Selby, 2014). Debugging is the skill to recognize when actions do not correspond to instructions, and the skill to fix errors (Selby, 2014).

Table 1 shows the elements of computational thinking as these have been discussed and defined in this section. Accordingly, this conceptual framework is the one that was adopted for developing the computational thinking curriculum framework for K-6 presented in the next section.

Table 1. The elements of computational thinking

Element	Definition
1. Abstraction	The skill to decide what information about an entity/object to keep and what to ignore (Wing, 2011).
2. Generalization	The skill to formulate a solution in generic terms so that it can be applied to different problems (Selby, 2014).
3. Decomposition	The skill to break a complex problem into smaller parts that are easier to understand and solve (National Research Council, 2010; Wing, 2011).
4. Algorithms	The skill to devise a step-by-step set of operations/actions of how to go about solving a problem (Selby, 2014).
a. Sequencing	The skill to put actions in the correct sequence (Selby, 2014).
b. Flow of control	The order in which instructions/actions are executed (Selby, 2014).
5. Debugging	The skill to identify, remove, and fix errors (Selby, 2014).

A computational thinking curriculum framework for K-6

Based on the five computational thinking skills shown in Table 1, a computational thinking curriculum framework is developed and presented in Table 2. Table 2 shows indicators of competence for all five computational thinking skills, namely, abstraction, generalization, decomposition, algorithmic thinking, and debugging, in a progression from simple to complex across the educational levels of K-2, 3-4, and 5-6. Succinctly, the framework aims at engaging children in thinking and problem solving by developing a solution to a problem, automating the solution through algorithmic thinking, and generalizing this solution to new problems when common patterns are identified or recognized. In essence, the framework aims at introducing students of a very young age to the thinking processes of computational thinking so they become competent to learn more advanced theoretical and practical topics of computer science in secondary education. In addition, the framework targets the development of all five computational thinking skills across all K-6 levels, albeit at different levels of competence, through the use of examples and tasks that are within the reach of children either with or without external support (external reference systems). It is noted that the boundaries specified for each level may possibly vary from school to school and from classroom to classroom. By the same token, it is also expected that refinements to the curriculum framework will be ongoing once data become available from pilot offerings of different curricula, aligned with the proposed framework, in diverse contexts.

Table 2. A computational thinking curriculum framework for K-6

Skill	Grade level (age level)		
	K-2 (ages 6 to 8)	3-4 (ages 9 to 10)	5-6 (ages 11 to 12)
Abstraction	• With the use of external reference systems, create a model/representation* to solve a problem (i.e., using specific directional language - forward, left turn, right turn, back - and turns of a given degree (90, 180, 270, 360), children create a path and write instructions to enable others to follow the path, or children design a mat based on a story, and have their Bee-Bot	• Create a model/representation to solve a problem (i.e., create an object and assign properties to it during an activity of digital game design and creation).	• Create a new model/representation to solve a problem (i.e., create a simulation using Scratch).

	follow the path from the narrative).		
Generalization	• Identify common patterns between older and newer problem-solving tasks, and use sequences of instructions previously employed, to solve a new problem (i.e., use a sequence of instructions from an older path, to program the Bee-Bot to follow a new path that includes the older path).	• Remix and reuse (by extending if needed) resources that were previously created.	• Remix and reuse (by extending if needed) resources that were previously created.
Decomposition	• Break a complex task into a series of simpler subtasks (i.e., break a longer path into a series of smaller paths that the Bee-Bot can follow).	• Break a complex task into simpler subtasks. • Develop a solution by assembling together collections of smaller parts.	• Break a complex task into simpler subtasks. • Develop a solution by assembling together collections of smaller parts.
Algorithmic thinking	• Define a series of steps for a solution. • Put instructions in the correct sequence.	• Define a series of steps for a solution. • Put instructions in the correct sequence. • Repeat the sequence several times (iteration).	• Define a series of steps for a solution. • Put instructions in the correct sequence. • Repeat the sequence several times (iteration). • Make decisions based on conditions. • Store, retrieve, and update variables. • Formulate mathematical and logical expressions.
Debugging	• Recognize when instructions do not correspond to actions. • Remove and fix errors.	• Recognize when instructions do not correspond to actions. • Remove and fix errors.	• Recognize when instructions do not correspond to actions. • Remove and fix errors.

Note. *model/representation = can be conceptual, mathematical, mechanical, textual, graphical, etc.

Curriculum design issues: A focus on a holistic design approach

The framework presented in Table 2 constitutes a general framework that can be used to develop various computational thinking programs, courses, or modules in K-6. The curriculum framework is conceptualized in a generic form to allow teachers the freedom and agency to adapt and customize the framework as they see fit for their own classrooms and students. According to van den Akker (2010), this *enactment* perspective, where teachers create their own curriculum realities, is increasingly replacing the *fidelity* perspective on implementation where teachers faithfully follow curricular prescriptions from external sources. Accordingly, this trend "*puts even more emphasis on teachers as key people in curriculum change*" (van den Akker, 2010, p. 185), underlining the utmost importance of relevant teacher preparation. In view of that, the authors herein propose the holistic design approach as one method that teachers can use to enact the computational thinking framework proposed in this paper.

A holistic design approach attempts to "*deal with complexity without losing sight of the separate elements and the interconnections between those elements*" (van Merriënboer & Kirschner, 2007, p. 6). It is the opposite of an atomistic design where complex contents and tasks are reduced to simpler elements, promoting this way content compartmentalization and fragmentation. Compartmentalization and fragmentation support the separation of a whole

into small, distinct, and often isolated parts. For example, teachers teach children to think computationally by teaching them abstraction, then decomposition, followed by generalization, algorithmic thinking, and debugging. It is doubtful if in the end children will have the opportunity to practice the whole complex skill (computational thinking, in this case) in its entirety, and doubtful if they will ever learn to think computationally. On the other hand, a holistic design approach aims at eliminating compartmentalization and fragmentation by focusing on whole complex and authentic learning tasks, without losing sight of the individual elements that make up the complex whole. Thus, with this approach, if implemented correctly by the teacher, children learn to think computationally to solve a problem, and also learn all other constituent and interconnected pieces of knowledge (theoretical and or practical) that are directly related with the computational thinking task. We support the holistic design approach for teaching computational thinking and emphasize here two design steps in the process, namely, (a) the design of problem-solving tasks with a focus on real-life issues, and (b) the sequencing of problem-solving tasks from simple to complex. We do acknowledge that more design steps exist in the literature.

With regard to the first design step, it is argued that the sources of the computational thinking curriculum ought to be problems, issues, and concerns directly related to life itself. A curriculum of this kind will result in usable knowledge - that is, knowledge that can be applied directly in the context of real life, problems and concerns at hand - and not in inert knowledge (Voogt, Fisser, Good, Mishra, & Yadav, 2015; Webb, Fluck, Cox, Angeli-Valanides, Malyn-Smith, Voogt, & Zagami, 2015). Educational researchers have found that a curriculum that is focused on problem solving around real-world problems can result in greater intellectual curiosity, motivation, improved attitude toward schooling, and higher achievement in college (Wolf & Brandt, 1998). Consequently, a curriculum designed around real-life problems can be a way to make computational thinking relevant to students' lives, and, thus, a way to keep them interested in the subject matter. Ultimately, this may end up in increasing substantially the number of students who will eventually pursue computer science as their major field of study later in college.

From an implementation point of view, a curriculum designed around real-life problems demands a wider range of content, simply because authentic real-world problems are usually multidisciplinary in nature. As a consequence, a curriculum from this perspective poses new demands on teaching often requiring close collaboration among teachers with different content expertise. It should be noted that real-life problem-solving tasks constitute challenging design endeavors, and, a curriculum designer may approach the design process through the means of rapid prototyping before designing an entire educational program, course, or module.

With regard to the sequencing of the problem-solving tasks, a sequence from simple whole tasks to more complex whole tasks is recommended. It is made clear that each problem-solving task, irrespective of complexity, engages the learner in whole-task problem-solving experiences. In the context of computational thinking, this means that each learning task, simple or complex, confronts the learner with all or almost all of the constituent computational thinking skills for a real-life computational thinking experience. All tasks are meaningful, authentic, and relevant to children's life. A sequence of tasks constitutes the backbone of the computational thinking curriculum. It is also evident that children may need guidance and support as they start working on more challenging tasks. Support may be provided in the form of external reference systems to help students gradually develop abstractions. Students may also need guidance with the problem-solving process itself.

The knowledge that teachers need to teach the curriculum

As Gal-Ezer and Stephenson (2010) stated, having a curriculum is important, but preparing teachers to teach the curriculum is also critical. Amongst computer science teacher educators, the framework of pedagogical content knowledge (PCK) has been highly regarded as an appropriate framework for defining the knowledge teachers need to have to be able to teach computer science (e.g., Hubwieser, Magenheim, Mühling, & Ruf, 2013; Saeli, 2012). Succinctly, PCK refers to a body of knowledge, which is highly context sensitive, cannot be conceptualized in isolation from teachers' classroom and teaching experiences, and is beyond and above a simple synthesis of knowledge of subject matter and pedagogy (Shulman, 1986; Shulman, 1987). PCK is an amalgam of knowledge that "*embodies the aspects of content most germane to its teachability*" (Shulman, 1986, p. 9), and refers to the transformation of content into forms that are understandable to learners. According to van Driel and Berry (2012), having a good PCK means that teachers have several representations of the most commonly taught topics within a certain subject. The more representations teachers have at their disposal and the better they recognize learning difficulties, the more effectively they can deploy their PCK (van Driel & Berry, 2012).

Within the domain of computer science, a number of computer science education researchers attempted to define PCK for computer science, either in general ways (Hubwieser et al., 2013; Saeli, Perrenet, Jochems, & Zwaneveld, 2011; Stephenson, Gal-Ezer, Haberman, & Verno, 2005) or more specific ways (Saeli, 2012). Saeli et al. (2011) concentrated on the teaching of programming in secondary education, and provided a general conceptualization of PCK for the domain of programming in terms of its constituent elements (i.e., what to teach about computer programming, how to teach programming, and what are learners' difficulties in programming). In a following study, Saeli (2012) was able to provide a more specific conceptualization of PCK for the domain of programming in the context of secondary education, which included details about each constituent knowledge base. In terms of the content to be taught, she mentioned loops, data structures, arrays, problem-solving skills, decomposition, parameters, and algorithms amongst others. Regarding teachers' pedagogical knowledge she mentioned offering a simple programming language to better facilitate students' effort to learn the syntax of the language, and choosing several worthy problems to solve. Lastly, she identified learners' difficulties about different programming concepts, such as loops, arrays, variables, and general problem-solving skills.

In the early 2000s, though, a number of educational researchers undertook systematic efforts for extending and enriching the concept of PCK by adding Technology Knowledge as another essential category of teachers' knowledge base (Angeli & Valanides, 2005; Koehler & Mishra, 2008; Niess, 2005). From this perspective, the introduction of Technology Knowledge in the existing framework of PCK successfully expanded PCK to TPCK - that is, Technological Pedagogical Content Knowledge (Angeli & Valanides, 2005; Angeli & Valanides, 2009; Koehler & Mishra, 2008; Niess, 2005). A conceptualization of the framework of TPCK is proposed by Angeli and Valanides (2005; 2009) as shown in Figure 1. According to Figure 1, TPCK is conceptualized as a unique body of knowledge that is formed by the contribution of five distinct knowledge bases, namely, content knowledge, pedagogical knowledge, knowledge of learners, knowledge of the educational context, and technology knowledge (Angeli & Valanides, 2005; Angeli & Valanides, 2009). This body of knowledge grows when teachers are engaged systematically in useful educational practices, either in their own classrooms or teacher professional development programs.

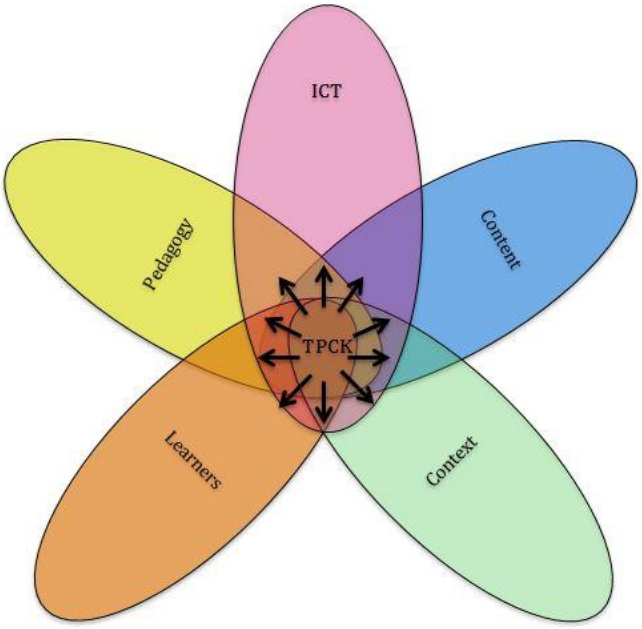

Figure 1. Technological Pedagogical Content Knowledge (adopted from Angeli & Valanides, 2005)

TPCK is an important body of knowledge for the field of computer science, because technology is at the center of the computer science domain, either, as a means in itself (i.e., to learn to use the technology as a goal), or as a means for achieving or teaching something else (i.e., to use technology in order to solve a problem or to teach a computer science concept). For the purposes of this study, the authors provide a conceptualization of TPCK for the construct of computational thinking, as it is defined in Table 1, in order to better explain what teachers need to know to be able to teach a computational thinking course aligned with the framework proposed in Table 2.

Analytically, content knowledge (CK) is defined as knowledge about computational thinking (CK_{CT}). This includes knowledge and understanding about the skills of abstraction (including modeling), denoted as $CK_{CT(A)}$, generalization, denoted as $CK_{CT(G)}$, decomposition, designated as $CK_{CT(D)}$, algorithmic thinking, designated as $CK_{CT(Algo)}$, and debugging, denoted as $CK_{CT(Debug)}$. $CK_{CT(Algo)}$ includes knowledge of several computational thinking concepts, such as, data, processing, information, sequencing, loops, parallel processing, events, conditions, operators, variables, and dataflow of control.

Learner knowledge for computational thinking (LK_{CT}) includes knowledge about learners' difficulties in (a) developing abstractions that are beyond of any particular programming language or tool, denoted as $LK_{CT(A)}$, (b) generalizing from one solution to another by identifying common patterns, denoted as $LK_{CT(G)}$, (c) decomposing complex problems to simpler ones, designated as $LK_{CT(D)}$, (d) thinking algorithmically to solve a problem (including difficulties in understanding relevant concepts, such as sequencing, loops, flow of control, conditions, etc.), denoted as $LK_{CT(Algo)}$, and (e) debugging, denoted as $_{LK}CT(Debug)$.

Pedagogical knowledge for computational thinking (PK_{CT}) includes the general pedagogical knowledge applicable to all other content domains (i.e., the use of questions to promote understanding, use of examples, explanation, demonstration), in addition to knowledge about subject-specific pedagogical practices pertinent to computational thinking. PK_{CT} is defined in terms of the following teaching tactics: (a) model how to problem solve or think about a problem in iterative and incremental ways, (b) present or explain a solution to a problem in terms of a series of steps, (c) model decision making based on conditions, (d) do something based on (and expanding) what others or you have done (reuse and remix), (e) show how a complex problem can be decomposed into simpler problems and develop a solution in increments, (f) show how to design a model before writing a computer program for solving the problem, and (g) try things out as you go and make revisions based on what happens.

Technology knowledge for computational thinking (TK_{CT}) includes knowledge and skills about how to (a) operate/use a variety of technologies, (b) invent new technologies/tools, (c) solve a task using technical processes, methods, and tools, and (d) learn and adapt to new technologies.

Context knowledge for computational thinking (CX_{CT}) is defined from the point of view explicated by Porras-Hernández and Salinas-Amescua (2013) who proposed to regard context knowledge along two important dimensions, namely (a) scope (macro, mezzo, and micro level context) and (b) actor (students' and teachers' inner and external context). Macro context is defined by social, political, technological, and economic conditions at a global level that influence the value and worth of adding computer science and computational thinking to the school curriculum. Mezzo context is defined by the social, cultural, political, organizational, and economic conditions settled in the local community and the educational institution about the value of computational thinking in children's lives. Finally, micro context is the level that deals with in-class conditions for learning (e.g., available resources for computational thinking, available technologies, norms and policies, beliefs, expectations, teachers' and students' goals about computational thinking). In addition, Porras-Hernández and Salinas-Amescua (2013) argued that in order to comprehend teachers' uses of technology, it is important to consider teachers' and students' (actors') unique characteristics, as they are brought in the context as separate objects of knowledge with internal (e.g., students' needs, preferences, misconceptions, learning difficulties, prior knowledge, teachers' self-efficacy, pedagogical beliefs) and external contexts (e.g., ethnicity, culture, community, and socioeconomic background).

Lastly, TPCK for computational thinking ($TPCK_{CT}$) is defined as knowing how to: (1) Identify a range of creative and authentic computational thinking projects; (2) Identify a range of technologies with an appropriate set of affordances in terms of providing the necessary technological means for practicing/teaching the whole range of computational thinking skills with each project; and (3) Use the affordances of technology to transform CK_{CT} and PK_{CT} using representations that make the overall computational thinking experience comprehensible for all learners.

The question that naturally arises at this point is: "What form should teacher preparation take, so that teachers develop their $TPCK_{CT}$ competencies adequately?" In the next section, we provide preliminary research evidence from a teacher education course on preparing teachers how to teach computational thinking.

Teacher preparation in developing TPCK competencies for computational thinking

In the fall of 2015, fifteen elementary school teachers pursuing a master's degree in instructional technology were enrolled in a course on learning how to teach computational thinking in their K-6 classrooms. All teachers were unfamiliar with computational thinking and had no prior experiences with computer programming. The teachers participated in 13 three-hour weekly meetings. The participants were engaged in hands-on design activities with the Scratch computer programming environment. The learning-by-design approach, which has been shown to be effective in contemporary teacher development studies (McKenney, Kali, Markauskaite, & Voogt, 2015), was used in the course to engage teachers in designing models of different problem situations before constructing computer programs for solving the problems.

The course instructor initially engaged the teachers in authentic problem solving by asking them to think about the city/town they were living and identify ways of how people's lives in those places could be improved. The teachers explained their thinking about possible improvements and then the instructor asked them to think about how computers could be used for solving some of the problems they identified. A brainstorming activity resulted in ten different ideas that constituted the real-life tasks that the course instructor used to teach the teachers about computational thinking. The ten tasks were sequenced from simple to complex based on the involvedness of the solution.

For each problem, teachers were taught how to create a model first before writing a computer program for solving the problem. Creating a model proved to be extremely difficult for the teachers and often times they asked their instructor for help. Early attempts in creating models resulted in models containing lots of unnecessary information, but, gradually teachers, with the help of the course instructor learned that models are abstractions of something free from inessential detail. The teachers were taught how to create models through a process that was explicitly taught to them and involved identification of the important entities of the model, their characteristics (parameters in the model), and relationships, either quantitative or qualitative, between the parameters of the entities. The teachers showed commitment in developing the best models they could possibly create, and, often times they exhibited lots of creative ideas of how to make them better.

In regards to teaching teachers computer programming, the course instructor used systematically the following pedagogical strategies: (a) decide what sprites are needed for your project, (b) decide what scripts are needed for your project, (c) organize the scripts in meaningful ways for you and others, (d) develop some code, try it out, then develop some more, (e) test and debug, and (f) build or extend on existing projects or ideas. During the programming tasks, computational concepts such as, data, processing, information, sequencing, loops, parallel processing, events, conditions, operators, variables, and dataflow of control, were explicitly explained and illustrated with lots of programming examples. Teachers had no difficulties with understanding programming concepts, even though they found the concepts of variables and conditional logic more challenging than the others.

Concluding remarks

In conclusion, the authors in this paper presented a computational thinking curriculum framework for designing a curriculum for K-6, an area of research that is still in its infancy, described design guidelines for enacting the curriculum framework, and defined $TPCK_{CT}$ as the body of knowledge that teachers need to have to be able to teach the curriculum in K-6. In addition, the authors provided an example of a teacher preparation course that was specifically designed to promote teachers' $TPCK_{CT}$. It is recognized that more empirical evidence in the form of rich educational cases is needed in terms of further investigating the effectiveness of the frameworks proposed herein in a variety of contexts. It is expected that with the gradual adoption of computer science as a distinct school subject in the K-6 curriculum of countries around the world, our knowledge and research base regarding the issues discussed in this paper will dramatically expand over the next several years.

References

Angeli, C., & Valanides, N. (2005). Preservice teachers as ICT designers: An Instructional design model based on an expanded view of pedagogical content knowledge. *Journal of Computer-Assisted Learning*, *21*(4), 292-302.

Angeli, C., & Valanides, N. (2009). Epistemological and methodological issues for the conceptualization, development, and assessment of ICT-TPCK: Advances in technological pedagogical content knowledge (TPCK). *Computers & Education*, *52*, 154-168.

Armoni, M. (2012). Teaching CS in kindergarten: How early can the pipeline begin? *ACM Inroads*, *3*(4), 18-19.

Astrachan, O., & Briggs, A. (2012). The CS principles project. *ACM Inroads*, *3*(2), 38-42.

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *ACM Inroads*, *2*(1), 48-54.

Bell, T. C., Witten, I. H., Fellows, M. R., Adams, R., & McKenzie, J. (2015). *CS Unplugged: An Enrichment and extension programme for primary-aged students*. Retrieved from http://csunplugged.org/wp-content/uploads/2015/03/CSUnplugged_OS_2015_v3.1.pdf

Czerkawski, B. (2015). Computational thinking in virtual learning environments. In *Proceedings of E-Learn: World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2015* (pp. 993-997). Chesapeake, VA: Association for the Advancement of Computing in Education (AACE).

desJardins, M. (2015). Creating AP® CS principles: Let many flowers bloom. *ACM Inroads*, *6*(4), 60-66.

Fluck, A., Webb, M., Cox, M., Angeli, C., Malyn-Smith, J., Voogt, J., & Zagami, J. (2016). Arguing for computer science in the school curriculum. *Educational Technology and Society*, *19*(3), 38-46.

Furber, S. (2012). *Shut down or restart? The way forward for computing in UK schools*. London, UK: The Royal Society.

Gal-Ezer, J., & Stephenson, C. (2010). Computer science teacher preparation is critical. *ACM Inroads*, *1*(1), 61-66.

Gibson, J. P. (2012, July). Teaching graph algorithms to children of all ages. In *Proceedings of the 17th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)* (pp. 34–39). New York, NY: ACM.

Goode, J., Chapman, G., & Margolis, J. (2012). Beyond curriculum: The Exploring computer science program. *ACM Inroads*, *3*(2), 47-53.

Hazzan, O., Lapidot, T., & Ragonis, N. (2011). *Guide to teaching computer science: An Activity-based approach*. London, UK: Springer.

Hubwieser, P., Magenheim, J., Mühling, A., & Ruf, A. (2013, August). Towards a conceptualization of pedagogical content knowledge for computer science. In *Proceedings of the ninth annual international ACM conference on International computing education research* (pp. 1-8). New York, NY: ACM.

Husing, T., & Korte, W. B. (2010). *Evaluation of the implementation of the Communication of the European Commission: E-skills for the 21st century*. Bonn, Germany: Empirica. Retrieved from http://hdl.voced.edu.au/10707/323186

Koehler, M. J., & Mishra P. (2008). Introducing TPCK. In AACTE Committee on Innovation and Technology (Eds.), *Handbook of Technological Pedagogical Content Knowledge (TPCK) for educators* (pp. 3–29). New York, NY: Routledge.

Kumar, D. (2014). Digital playgrounds for early computing education. *ACM Inroads*, *5*(1), 20-21.

McKenney, S., Kali, Y., Markauskaite, L., & Voogt, J. (2015). Teacher design knowledge for technology enhanced learning: An Ecological framework for investigating assets and needs. *Instructional science*, *43*(2), 181-202.

Micheuz, P. (2008). Some findings on informatics education in Austrian academic secondary schools. *Informatics in Education*, *7*(2), 221-236.

National Research Council. (2010). *Committee for the workshops on computational thinking: Report of a workshop on the scope and nature of computational thinking*. Washington, DC: National Academy Press. doi:10.17226/12840

Niess M. L. (2005). Preparing teachers to teach science and mathematics with technology: Developing a technology pedagogical content knowledge. *Teaching and Teacher Education*, *21*, 509–523.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books, Inc.

Porras-Hernández, L. H., & Salinas-Amescua, B. (2013). Strengthening TPACK: A Broader notion of context and the use of teacher's narratives to reveal knowledge construction. *Journal of Educational Computing Research*, *48*(2), 223-244.

Prottsman, K. (2014). Computer science for the elementary classroom. *ACM Inroads*, *5*(4), 60-63.

Saeli, M. (2012). *Teaching programming for secondary school: A Pedagogical content knowledge based approach* (Unpublished doctoral dissertation). Technische Universiteit Eindhoven, Netherlands.

Saeli, M., Perrenet, J., Jochems, W. M., & Zwaneveld, B. (2011). Teaching programming in secondary school: A Pedagogical content knowledge perspective. *Informatics in Education*, *10*(1), 73-88.

Selby, C. C. (2014). *How can the teaching of programming be used to enhance computational thinking skills?* (Unpublished doctoral dissertation). University of Southampton, Southampton, UK.

Shulman, L. S. (1986). Those who understand: Knowledge growth in teaching. *Educational Researcher*, *15*, 4–14.

Shulman, L. S. (1987). Knowledge and teaching: Foundations of the new reform. *Harvard Educational Review*, *57*, 1-23.

Stephenson, C., Gal-Ezer, J., Haberman, B., & Verno, A. (2005). *The New educational imperative: Improving high school computer science education.* New York, NY: Computer Science Teachers Association (CSTA).

Thalheim, B. (2000). *Fundamentals of entity-relationship modeling*. New York, NY: Springer.

Tucker, A. B., Deek, F., Jones, J., McCowan, D., Stephenson, C., & Verno, A. (2003). *A Model curriculum for K-12 computer science.* New York, NY: ACM/Computer Science Teachers Association.

van Diepen, N., Perrenet, J., & Zwaneveld, B. (2011). Which way with informatics in high schools in the Netherlands? The Dutch dilemma. *Informatics in Education*, *10*(1), 123-148.

van Driel, J. H., & Berry, A. (2012). Teacher professional development focusing on pedagogical content knowledge. *Educational Researcher*, *41*(1), 26-28.

van den Akker, J. (2010). Building bridges: How research may improve curriculum policies and classroom practices. In *Beyond Lisbon 2010: Perspectives from research and development for education policy in Europe* (pp. 177-195). Aarau, Switzerland: CIDREE.

van Merriënboer, J. V., & Kirschner, P. A. (2007). *Ten steps to complex learning: A Systematic approach to four-component instructional design*. Mahwah, NJ: Lawrence Erlbaum.

Voogt, J., Fisser, P., Good, J., Mishra, P., & Yadav, A. (2015). Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies*, *20*(4), 715-728.

Webb, M., Fluck, A., Cox, M., Angeli-Valanides, C., Malyn-Smith, J., Voogt, J., & Zagami, J. (2015). Curriculum: Advancing understanding of the roles of computer science/informatics in the curriculum. In K-W Lai (Ed.), *EDUsummIT 2015 Summary Report* (pp. 60-68). Retrieved from http://www.curtin.edu.au/edusummit/local/docs/edusummit2015-ebook.pdf

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

Wing, J. M. (2011, March). *Computational thinking.* Retrieved from https://csta.acm.org/Curriculum/sub/CurrFiles/WingCTPrez.pdf

Wolf, P., & Brandt, R. (1998). What do we know from brain research? *Educational Leadership, 56*(3), 8-13.