Chapter 2
Computational Thinking—More Than a Variant of Scientific Inquiry!

H. Ulrich Hoppe and Sören Werneburg

Abstract The essence of Computational Thinking (CT) lies in the creation of "logical artifacts" that externalize and reify human ideas in a form that can be interpreted and "run" on computers. Various approaches to scientific inquiry (learning) also make use of models that are construed as logical artifacts, but here the main focus is on the correspondence of such models with natural phenomena that exist prior to these models. To pinpoint the different perspectives on CT, we have analyzed the terminology of articles from different backgrounds and periods. This survey is followed by a discussion of aspects that are specifically relevant to a computer science perspective. Abstraction in terms of data and process structures is a core feature in this context. As compared to a "free choice" of computational abstractions based on expressive and powerful formal languages, models used in scientific inquiry learning typically have limited "representational flexibility" within the boundaries of a predetermined computational approach. For the progress of CT and CT education, it is important to underline the nature of logical artifacts as the primary concern. As an example from our own work, we elaborate on "reactive rule-based programming" as an entry point that enables learners to start with situational specifications of action that can be further expanded into more standard block-based iterative programs and thus allows for a transition between different computational approaches. As an outlook beyond current practice, we finally envisage the potential of meta-level programming and program analysis techniques as a computational counterpart of metacognitive strategies.

Keywords Computational thinking · Computational abstraction · Computer science education

H. Ulrich Hoppe (✉) · S. Werneburg
Department of Computer Science and Applied Cognitive Science,
University of Duisburg-Essen, Lotharstraße 63, 47057 Duisburg, Germany
e-mail: hoppe@collide.info

S.-C. Kong and H. Abelson (eds.), *Computational Thinking Education*,
https://doi.org/10.1007/978-981-13-6528-7_2

2.1 Introduction

2.1.1 Origins of the Current Debate

Although Papert (1996) had already used the term "Computational Thinking" (CT) ten years earlier, the current discussion of CT can be traced back to Wing (2006). Wing characterized CT by stating that it "involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science". She emphasized that CT is not about thinking like a computer but about how humans solve problems in a way that can be operationalized with and on computers.

Science as well as humanities are influenced by CT since "computational concepts provide a new language for describing hypotheses and theories" in these fields of interest (Bundy, 2007). In addition, Barr and Stevenson (2011) formulated CT-related challenges for K-12 education in computer science, math, science, language and social studies.

Wing (2008) refined her first statement pointing out that the essence and key of CT is abstraction in a way that is more complex than in mathematics. If someone builds a computational model and wants to include all properties seen in the real environment, s/he cannot enjoy "the clean, elegant or easy definable algebraic properties of mathematical abstraction".

2.1.2 Computational Thinking for K-12

To bring CT to K-12, the International Society for Technology in Education and the Computer Science Teacher Association (ISTE and CSTA 2011) presented a definition of CT for K-12 Education

Computational Thinking is a problem-solving process that includes (*but is not limited to*) *the following characteristics*:

- *Formulating problems in a way that enables us to use a computer and other tools to help solve them*
- *Logically organizing and analyzing data*
- *Representing data through abstractions, such as models and simulations*
- *Automating solutions through algorithmic thinking (a series of ordered steps)*
- *Identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources*
- *Generalizing and transferring this problem-solving process to a wide variety of problems.*

The first three bullet points highlight the importance of computational modeling as part of the problem-solving process. General approaches to computational modeling are typically facilitated by programming languages. However, we will see that the creation of computational or formal models related to given problems can also make use of other abstract tools such as automata or grammars.

As part of the same proposal, it is claimed that CT problem-solving should be connected to realistic applications and challenges in science and humanities. This would link the learning experience and the ensuing creations to relevant real-world problems. However, realism in this sense can also lead to a "complexity overkill" that obstructs the pure discovery of important basic building blocks (the nature of which will be elaborated later). On the other hand, if models are simplified too much they lose fidelity and ultimately credibility. Many computer science concepts do not require an application to real complex environment in their basic version. Interactive game environments, for example, do not necessarily require an accurate modeling of physics but they can still promote the learning of CT concepts.

2.1.3 Model Progression: The Use-Modify-Create Scheme

CT activities typically result in the creation of logical artifacts that can be run, tested against the original intentions, and can be refined accordingly. The creation of an initial executable artifact can be very challenging for learners. Lee et al. (2011) presented a model with a low threshold for novices and promoted it as the "Use-Modify-Create Progression" (see Fig. 2.1).

In the initial phase of this process, learners *use* or *analyze* a predefined and executable (programming) artifact that typically contains a new construct or new type abstraction. In this phase, the learners will *modify* the artifact, which causes observable changes in the output. At the beginning, these modifications are often confined to *varying* predefined parameters. The consecutive *create* phase is essential for the appropriation of new CT skills, giving learners the opportunity to be creative and express themselves through programming. Iteratively, the students create new computational artifacts, execute these, and evaluate the ensuing outputs. This progression

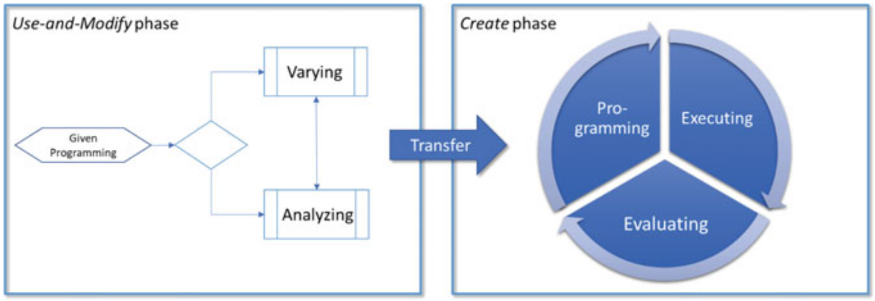


Fig. 2.1 Use-modify-create progression. Adapted from Lee et al. (2011)

incorporates a smooth transition from reusing a predefined new type of artifact to learner generated creative construction.

The specification given by Lee et al. does not provide a clear distinction between the three phases of the cycle. Therefore, it is necessary to refine and rethink this part of the pedagogical model. It is certainly important for novices to have an "interactive starting point" that does not depend on their own original creation and a clear transfer between the *Use-Modify* phase and the *Create* phase is necessary. However, the creative-constructive activities of modeling, specifying and representing problems are also very important ingredients of computational thinking processes.

2.1.4 The CT Terminology

There are few meta-level studies on the existing CT literature. For example, Kaleli-oglu, Gülbahar, and Kukul (2016) unveiled that CT is very frequently related with "problem-solving", whereas Lockwood and Mooney (2017) point out that the definition of CT as well as the embedment of CT in curricula are still emerging, which make a final characterization difficult.

To characterize the discourse on CT considering different phases and perspectives, we have conducted a text analysis on selected corpora of scientific articles. We used text mining tools to extract important concepts connected to CT. First, we defined three and selected categories of articles[1]:

Articles about CT mainly related to …

(i) … computer science education before 2006 (corpus $n = 14$),
(ii) … computer science education 2006 and later (corpus $n = 33$),
(iii) … inquiry-based learning in science (corpus $n = 16$).

The distinction between the first two categories and corpora is based on a separation with respect to the Wing (2006) article. The third category is also limited to articles following up on Wing (2006) but with a focus in K-12 education in science classes and humanities classes. We will show how the concepts related to CT will differ.

To achieve this, we extracted terms from the three corpora based on the standard *tf-idf*-measure. Since the corpora differ in number of documents and volume, the extraction has been normalized to yield about 100 terms per category. The reduction of terms is based on sparsity of the document-term matrices. In the first step, the full paper texts (abstracts included) were loaded. After removing unnecessary symbols and stop words, the texts have been lemmatized using a dictionary of base terms. Relevant composite terms such as "computational thinking" (comp_think), "computer science", "problem solving" (prob_solv) or "computational model" (comp_model) are considered as possible target terms.

[1]The corpora of the three categories can be found in Appendix A on www.collide.info/textminingCT.

The sparsity criterion leads to a selection of terms that appear in at least p documents as a threshold t. The threshold has been adjusted per category to achieve the desired selection of 100 terms approximately. Table 2.1 shows the attributes of each matrix before and after removing of the sparse terms.

The frequency-based word clouds shown in Fig. 2.2 indicate differences in the corresponding portions of discourse for the three categories. Terms such as "learn", "model", "system", and "problem" appear in each category but with different weights. The earlier work in computer science education (first category) frequently refers to "mathematics" and "systems". Category (ii) contains an explicit notion of "computer science", whereas "science" in general and "simulation" are characteristic for category (iii). Category (ii) is also more concerned with special concepts related to computer science, such as "algorithm", "abstraction", and "computing". A special focus in this category appears to be on "games".

In Fig. 2.3, a comparison between selected frequent terms between all three categories is presented. Here, each bar represents the tf-idf-measure normalized to the selected term base.

Figure 2.3 corroborates the impressions gained from the word cloud diagrams. The profiles of categories (ii) and (iii) appear to be quite similar among each other as compared to category (i). "Algorithm" and "abstraction" are more important concepts

Table 2.1 Attributes of the document term matrix before and after removing of sparse terms

	Category (i)	Category (ii)	Category (iii)
Documents	14	33	16
Terms	4329	4411	2143
Average document length	1619.57	834.21	593.81
Sparsity (%)	84	91	88
Terms (with threshold)	88(t = 35%)	98(t = 55%)	104(t = 65%)
Sparsity (%)	21	40	55

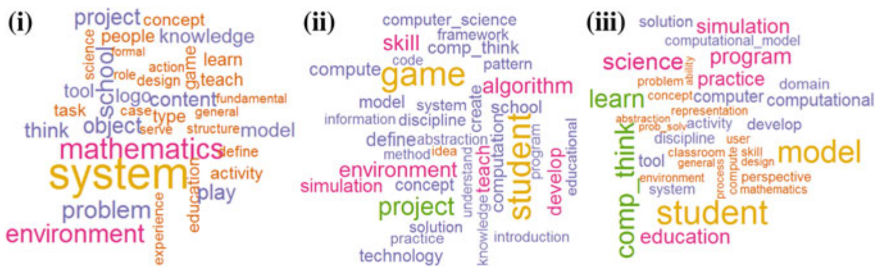


Fig. 2.2 Word Clouds of the terms with the highest tf-idf value of category (i), (ii), and (iii)

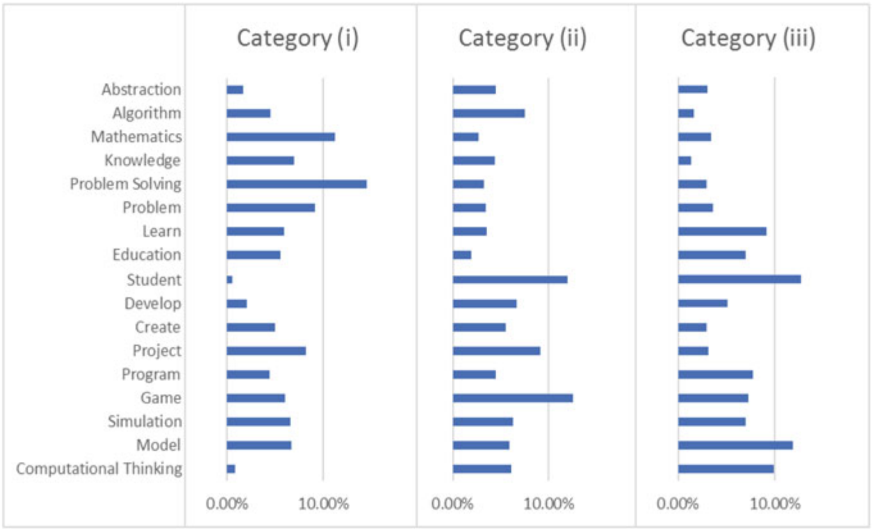


Fig. 2.3 Normalized frequencies of selected terms

in category (ii), whereas "education" is more prominent in category (iii). The latter observation corresponds to the disciplinary background of the contributions which is more often education and less computer science for category (iii) as compared to (ii). However, CT as a theme is even more frequently addressed in category (iii). Several papers in category (ii) use games as examples to illustrate general principles of CT, which explains the prominent role of this concept. For category (i), in addition to the focus on mathematics (cf. "the (Logo) turtle meets Euclid"—Papert 1996), this earlier discourse is much more centered on "knowledge" and "problem (solving[2])" as epistemological categories. In the later discourse for both (ii) and (iii), the epistemological perspective is substituted by instructional and cognitive frameworks.

2.2 Basic Concepts and Building Blocks

2.2.1 "Computational Models" and "Models of Computation"

In his definition of CT, the computer science pioneer Aho (2012) characterizes Computational Thinking as "the thought processes involved in formulating problems so their solution can be represented as computational steps and algorithms". Before

[2]Problem-solving itself is only present in some documents, so the sparsity threshold led to removing this term from the word cloud, even though the td-idf value was high.

starting the problem formulation and construction of a solution, it is necessary to specify an appropriate "model of computation" as a basis. This notion, sometimes also named "computational model" is very specific and important to computer science and very different from the notion of "computational model" that denotes the resulting computational artifact of a constructive CT activity. Aho's "models of computation" are the background and conceptual platforms of computational artifacts; they correspond to "abstract engines" of the type of Turing Machines, Finite State Machines or von Neumann architectures but also possibly Petri Nets, grammars or rewrite system. These models define the basic mechanisms of interpretation of computational artifacts (i.e., "computational models" of the other type).

The grounding of computation on such abstract engines, and accordingly the choice of such foundations, is not very much in the focus of the current more educationally inspired CT discourse, and it might seem that these choices have little to do with CT practice. However, there are relevant examples that explicitly address the choice of basic "models of computation".

The "Kara" microworld (Hartmann, Nievergelt, & Reichert, 2001) uses a programmable ladybug to introduce concepts of computer science and programming. It comes with different versions based on different abstract engines. The original version was based on Finite State Machines, but later versions based on programming languages (JavaKara, RubyKara) were added. That is, the microworld of Kara allows for solving the same of similar problems based on different "models of computation" in the sense of Aho. The FSM version allows for linking the Kara experience to automata theory in general, whereas the Java and Ruby versions may be used as introductions to programming.

Kafura and Tatar (2011) report on a Computational Thinking course for computer science students in which different abstract formalisms or engines (including Petri nets, BNF grammars, lambda expressions) with corresponding tools were employed to construct computational models in response to various problems. This example shows that relying on abstract models of computation can be an alternative to using programming in the construction of computational artifacts.

Curzon and McOwan (2016) describe a specific relation between computational modeling and algorithmic thinking: The algorithm simulates the transformation of an idea contextualized in a virtual or real world into a runnable computational representation, possibly as part of a game environment. However, computational modeling also refers the specification and handling of data and specification process structure. The choice of such structures is not only induced by the "external" problem but is also governed by a partially independent "logic of computation".

2.2.2 The Notion of "Abstraction"

A central element of CT is abstraction in the specific sense it has in the context of computational principles. Wing (2008) underlines the importance of abstraction for computational thinking

> The essence of computational thinking is abstraction… First, our abstractions do not necessarily enjoy the clean, elegant or easily definable algebraic properties of mathematical abstractions, such as real numbers or sets, of the physical world… In working with layers of abstraction, we necessarily keep in mind the relationship between each pair of layers, be it defined via an abstraction function, a simulation relation, a transformation or a more general kind of mapping. We use these mappings in showing the observable equivalence between an abstract state machine and one of its possible refinements, in proving the correctness of an implementation with respect to a specification and in compiling a program written in a high-level language to more efficient machine code. And so the nuts and bolts in computational thinking are defining abstractions, working with multiple layers of abstraction and understanding the relationships among the different layers. Abstractions are the 'mental' tools of computing.

It is necessary to distinguish between the general understanding of abstraction and the specific nature of computational abstraction, or better "abstractions" (plural). The general idea of abstraction as a conceptual generalization and omission of coincidental details can of course also be found in the computational domain. Abstraction in this general sense if a process that supports the building of categories and thus the structuring of a domain. However, in computer science, abstractions can also be representational or operational constructs (i.e., mental tools in the words of Wing), which is not clear from a common sense point of view. Abstractions of this type include data structures, different concepts of variables depending on underlying models of computation or programming paradigms, procedural, and functional abstraction, recursion, as well lambda expressions in combination with higher order functions.

Hu (2011) elaborates on the relation between CT and mathematical thinking, arguing that, behind a quite different surface, CT skills are indeed quite similar to mathematical thinking. It is true that computational artifacts are governed by mathematical (especially logical) principles. Yet, where mathematical models focus on general structural properties, computational artifacts are operational in the sense that they have inherent behavior. This is reflected in the constructive nature of computational abstractions. CT as a creative activity involves choices and combinations of computational abstractions as mental tools. This calls for "representational flexibility", i.e., the provision of making choices between different representations and abstractions, as part of any kind of teaching and learning targeting CT.

2.2.3 Languages, Representations, and Microworlds

The building of computational artifacts requires a "medium" of representation that affords certain computational mechanisms in way susceptible to human imagination and creation. This idea is reflected in diSessa's notion of computational media as basis for "computational literacies" (diSessa, 2000). The notion of literacy sets the focus on the aspect of familiarity and possibly mastery with respect to the specific medium. The computational medium would include a "model of computation" in Aho's sense and would, on this basis, provide more or less easy access to different types of abstractions. It is a natural choice to use a programming language for this purpose. For programming languages, it is well known that they resonate with computational abstractions (as constructs) in specific ways. For example, the concept of a variable as a storage or memory location is typical for imperative languages. It implies that variables can have mutable values, which is different from the concept of variables in pure functional or logical languages. However, as we have exemplified by referring to the Kara example (Hartmann et al., 2001) and to the CT course designed by Kafura and Tatar (2011), programming languages are not the only choice as a computational medium. Theory-inspired formal approaches provide "abstract engines" that can be used to build computational models, whereas "unplugged" approaches rely on physical or real-world models (e.g., teaching and explaining recursion through role-play).

Computational media for creative and constructive learning are often combined with concrete application domains (corresponding also to learning domains) for which the medium and its representational primitives are particularly designed. This corresponds to the notion of a "microworld", which was already one of the building blocks of the Logo approach. The educational affordances and usage patterns that originate from microworlds are immense and have been widely discussed from an educational technology point of view, see, e.g., (Rieber, 1996). The nature of a microworld as a computational construct and a tool of thought is nicely captured by diSessa (2000)

> A microworld is a type of computational document aimed at embedding important ideas in a form that students can readily explore. The best microworlds have an easy-to-understand set of operations that students can use to engage tasks of value to them, and in doing so, they come to understanding powerful underlying principles. You might come to understand ecology, for example, by building your own little creatures that compete with and are dependent on each other.

From a computer science perspective, microworlds in the sense described by diSessa can be conceived as *domain-specific languages* designed to facilitate constructive learning in certain domains. Compare the general characterization given by van Deursen, Klint, and Visser (2000): "*A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*" This suggests that the principles of designing and implementing DSLs should be taken into account when we develop microworlds as computational media for learning.

2.2.4 CT from the Perspective of Inquiry Learning in Science

Inspired by the process of scientific discovery, Inquiry Learning (IL) is defined as "an approach to learning that involves a process of exploring the natural or material world, and that leads to asking questions, making discoveries, and rigorously testing those discoveries in the search for new understanding" (Ash, 2003). Inquiry learning leads students through various phases (Pedaste et al., 2015), typically starting with an orientation phase followed by a conceptualization with idea generation and the development of hypotheses. During the investigation phase, students engage in experimentation, taking measurements to test their hypotheses. Finally, the results are evaluated and discussed, which may lead to reformulation of hypotheses. This is usually understood as a spiral or cyclic process that allows for repeated revisions and refinements (Minner, Levy, & Century, 2010).

CT and IL practices overlap in the underlying cyclic phase models as learning process structures, as exemplified by the *use-modify-create progression* and various IL-related process models. However, the role of computational artifacts differs between CT and IL: In IL, the artifact or model serves as a proxy for a scientific target scenario (e.g., an ecosystem) and the main point is what the model can tell us about the original. In CT, the computational artifact is of primary interest per se, including the way it is built and its inherent principles. If a learner evaluates the computational artifact (or model) at hand in an IL context, this will typically involve a variation of parameters and possibly redefinition of behaviors. In a CT context, this encompasses the reflection and redesign of the underlying computational model and representation as well.

Sengupta, Kinnebrew, Basu, Biswas, and Clark (2013) elaborate in detail on relationships between IL and CT concepts using the CTSiM environment ("Computational Thinking in Simulation and Modeling") as a concrete point of reference. The environment as such is based on visual agent-based programming. Their approach is that, from educator's perspective, students learn best when they use design-based learning environments which is also an approach of "science in practice" that involves "engag(ing) students in the process of developing the computational representational practices " (Sengupta, Kinnebrew, Basu, Biswas, & Clark 2013). In this article, certain computational concepts and principles are related to aspects of the IL environment. Abstraction is discussed in relation to the classical definition given by the philosopher Locke as the process in which "ideas taken from particular beings become general representatives of all of the same kind" (Locke, 1700). As we have seen above, this is not sufficient for the understanding of abstraction in a computer science sense. The specified relationships between computer science concepts and structural and operational aspects found in the CTSiM environment are rich and relevant. Yet, we need to distinguish between representational choices made in the design and implementation of the environment and choices that are "handed over" to the learners operating in the environment using the visual agent-based programming interface. These choices are indeed limited.

Perkins and Simmons (1988) showed that novice misconceptions in mathematics, science, and programming exhibit similar patterns in that conceptual difficulties in each of these domains have both domain-specific roots (e.g., challenging concepts) and domain general roots (e.g., difficulties pertaining to conducting inquiry, problem-solving, and epistemological knowledge).

In this perspective, the development of scientific expertise is inseparably intertwined with the development of epistemic and representational practices, e.g., (Giere, 1988; Nersessian, 1992; Lehrer & Schauble, 2006; NRC, 2008). Basu et al. (2012) describe how students use computational primitives to "generate their computational models". The students can see "how their agents operate in the microworld simulation, thus making explicit the emergence of aggregate system behavior" (Basu et al., 2012). Their "computational model" is focused on the description of the system dynamics using computational tools. There is an overlap between inquiry learning, system thinking, and computational thinking. Although the overlap between the areas seems to be evident, CT involves competencies and skills that can be clearly distinguished from the other fields.

The discussion above suggests that although Wing (2006) defined CT as a "thought process", computational thinking becomes evident only in particular forms of epistemic and representational practice that involve the generation and use of external representations (i.e., representations that are external to the mind) by the learners (Sengupta et al., 2013).

2.2.5 Interim Summary

Regarding the structuring of learning processes and the enrichment of such processes with computational media, inquiry learning in science and CT education are quite closely related. However, a discourse that is primarily driven by pedagogical inspirations and interest tends to neglect the importance of genuine computer science concepts and their role in shaping CT. The essence of CT lies in the creation of "logical artifacts" that externalize and reify human ideas in a form that can be interpreted and "run" on computers. The understanding of the principles underlying and constituting such logical artifacts, including "models of computation" in the sense of Aho as well as specific "abstractions as constructs", are of central importance for CT. In contrast, in general scientific inquiry learning, computational models are instrumental for the understanding the domain if interest (e.g., the functioning of ecosystems or certain chemical reactions). Usually, the computational media used in scientific inquiry learning contexts are largely predetermined in terms of data structures and processing mechanisms. In this sense, they are of limited "representational flexibility" regarding the free choice of data representations and algorithmic strategies.

2.3 Specific Approaches and Examples

2.3.1 From Reactive Rule-Based Programming to Block Structures

We have already seen that even for the practical activities CT cannot be reduced to programming only. However, there is no doubt that programming resonates with and requires CT. Programs are the most prominent examples of computational artifacts. Starting the construction and creation of a program is a difficult challenge especially for beginners. Guzdial (2004) discusses different approaches and tools for novices, such as the Logo family, the rule-based family, and traditional programming approaches. These tools provide different underlying models of computation as a basis to create computational artifacts. Although there is a wider variety of options block-based programming tools became a standard for introductory programming (Weintrop & Wilensky, 2015). However, depending on which tool best supports the implementation of an idea, students should be able to choose the way how they represent their ideas as computational artifacts.

We propose a "reactive rule-based programming" tool, in which the user/learner defines program elements as reactions of a programmable agent to situations and events in the learning environment. There is a close interplay between "running" and "teaching" the agent. Whenever the agent finds itself in a situation for which there is no applicable rule the learner will be prompted to enter such a rule. The condition part of the rule is generated by the system in correspondence to the given situational parameters (context). The user then specifies the actions to be applied. Once a suitable rule is entered the system will execute it. If more and more rules are provided the system will be able to execute chains of actions without further user input.

As can be seen in Fig. 2.4, the current context determines the conditions that must be considered for the rule. If none of the already defined rules applies the system requests the student to define actions for the current situation. Then, the defined rule can be executed and the cycle restarts in a new situation or ends in a predefined end. This loop puts the learner in a first person perspective of identifying herself/himself with the agent in the environment.

The reactive rule-based programming approach is the basis for the *ctMazeStudio*, a web-based programming environment to develop algorithms steering an agent

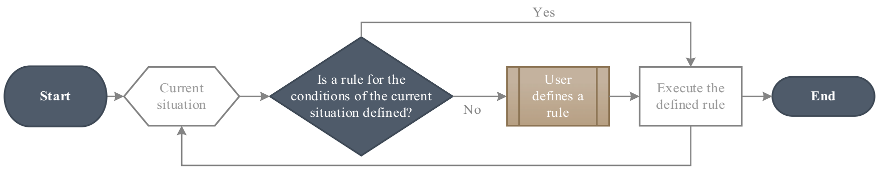
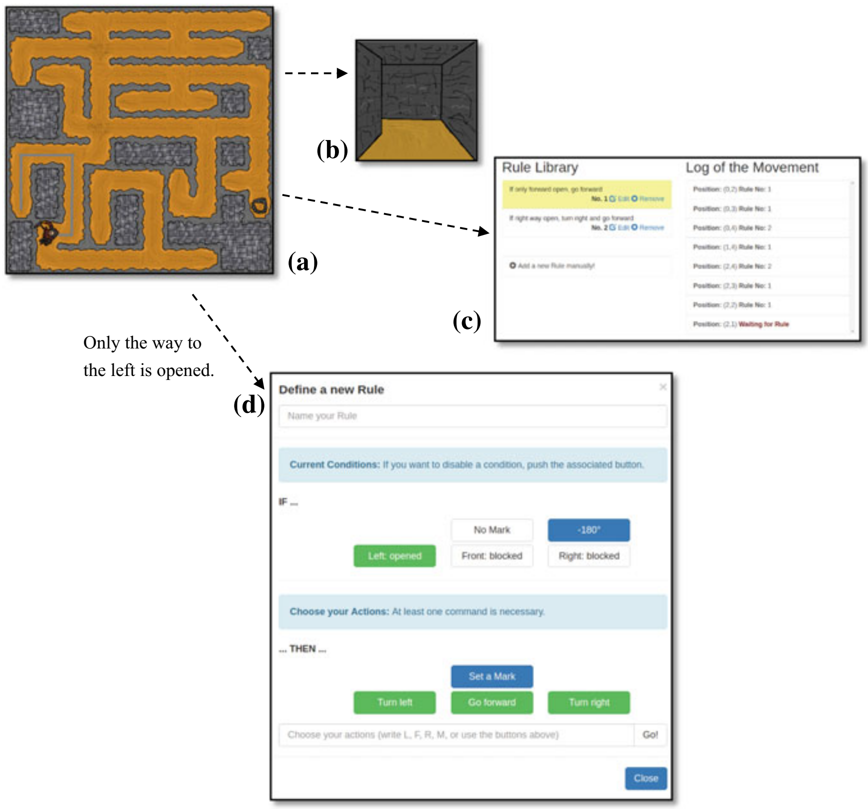


Fig. 2.4 Flow diagram for the reactive rule-based programming paradigm

who tries to find a way out of a maze. The goal for the students is to implement a universal solution that works on any maze. This learning environment contains three components: the rule editor, the behavior stage and a rule library (Fig. 2.5).

As can be seen in Fig. 2.5, the rule editor (d) provides a visual programming interface, which is available when a new situation is encountered. The editor comprises a condition component (IF part) and an action component (THEN part). For the given conditions, the students can select the desired actions for the current and corresponding situations with the same conditions to define a local "reactive" behavior. The users can also delete conditions, which implies that the corresponding rule will be applied more generally (generalization).

The rule library (c) manages the collection of all defined rules. In this user interface, the students can edit or delete already defined rules, directly enter new rules and change the order (and thus priority) of the rules to be checked. In the behavior stage (a), the behavior of the agent is visualized. Depending on the entries in the



If only forward open, go forward
No. 1
If right way open, turn right and go forward
No. 2
Add a new Rule manually!

Position: (0,2) Rule No: 1
Position: (0,3) Rule No: 1
Position: (0,4) Rule No: 2
Position: (1,4) Rule No: 1
Position: (2,4) Rule No: 2
Position: (2,3) Rule No: 1
Position: (2,2) Rule No: 1
Position: (2,1) Waiting for Rule

Current Conditions: If you want to disable a condition, push the associated button.

Choose your Actions: At least one command is necessary.

Choose your actions (write L, F, R, M, or use the buttons above)

Fig. 2.5 Visualization of the reactive approach in *ctMazeStudio*

rule library, the corresponding actions are executed and the specific entry in the rule library is highlighted.

To support CT, it is possible to apply different strategies to improve the programming code. When learners investigate and test their rulesets in consecutive situations, they may revise formerly defined rule sets through generalization (of conditions) or reordering. The challenge is to create a maximally powerful ruleset with a minimum number of rules. This requires a level of understanding that allows for predicting global behavior based on locally specified rules. In the maze example, as one of the first steps, a small set of rules will be created to implement a wall-following strategy. This strategy will be later refined, e.g., to avoid circling around islands.

The *ctMazeStudio* environment can also be programmed through a block-structured interface to represent conditions and loops governed by a global control strategy. In this sense, it provides a choice between different models of computation and thus supports "representational flexibility". Based on these options, we are currently studying transitions from rule-based reactive to block-structured iterative programs.

2.3.2 "Computational Metacognition"

CT is based on the principle of externalizing and "reifying" mental constructs or ideas in a computational medium. Accordingly, there is a close correspondence between the mental and the computational model. However, not all mental or cognitive constructs may lend themselves to such a mapping. How far can this approach be extended? In the sequel, we discuss the potential of extending this correspondence towards second order, "reflexive" constructs:

From a pedagogical point of view, thinking skills or cognitive skills comprise metacognition and reflection as important ingredients (see, e.g., Schraw, 1998). This raises the question if such a mapping from the mental to computational realm is also possible for metacognition. Indeed, advanced abstraction techniques in programming and program analysis allow for such mappings.

A first example of adding "computational metacognition" to CT environments is the application of analytic (computational) techniques to learner generated computational artifacts, especially to programs. We have to distinguish between static analyses based on the source code (as is the case for certain types of software metrics) and dynamic analyses based on the run-time behavior and results of programs (including "testing" approaches such as JUnit for Java).

Matsuzawa et al. (2017) used coding metrics to analyze characteristics of programming exercises and visualized these through a dashboard. The provision of this visualization could improve the teaching and understanding of classroom exercises in introductory programming. Manske and Hoppe (2014) have used static and dynamic analysis techniques to assess "creativity" in student programs created as solutions to so-called "Euler Problems" (see https://projecteuler.net/). Usually, it is easy to provide a brute force solution but additional insight is needed to make it more elegant and efficient. An example challenge is finding the sum of all even Fibonacci numbers with values below 4 million (Problem 2). These Euler problems define challenges of both mathematical and computational nature. Manske and Hoppe introduced several metrics to capture different features of the programming solutions. Static and dynamic code metrics (including lines of code, cyclomatic complexity, frequency of certain abstractions, and test results) cover structural quality and compliance, while similarity based metrics address originality. In a supervised machine learning approach, classifiers have been generated based on these features together with creativity scores from expert judgments. The main problem encountered in this study was the divergence of human classification related to creativity in programming.

Logic programming in combination with so-called meta-interpreters allows for dynamically reflecting deviations of actual program behavior from intended results or specifications (Lloyd, 1987). The method of deductive diagnosis (Hoppe, 1994) uses this technique in the context of a mathematics learning environment to identify and pinpoint specific mistakes without having to provide an error or bug library. From a CT point view, these meta-level processing techniques are relevant extensions of the formal repertoire. In the spirit of "open student modeling" (Bull & Kay, 2010), not only the results but also the basic functioning of such meta-level analyses could be made available to the learners to improve reflection on their computational artifacts and to extend their understanding of computational principles.

Of course, metacognition has also been addressed and discussed in several approaches to inquiry learning (White, Shimoda, & Frederiksen, 1999; Manlove, Lazonder, & Jong, 2006; Wichmann & Leutner, 2009). In these approaches, metacognition is conceived as an element of human learning strategies, possibly supported by technology but not simulated on a computational level. The examples above show that "second order" computational reflection techniques can be applied to "first order" computational artifacts in such way as to reveal diagnostic information related to the underlying human problem-solving and construction processes. In this sense, we can identify computational equivalents of metacognitive strategies. Making these second order computations susceptible to human learning as tools of self-reflection is a big challenge, but certainly of interest for CT education.

2.4 Conclusion

The current scientific discourse centered around the notion of "Computational Thinking" is multi-disciplinary with contributions from computer science, cognitive science, and education. In addition, the curricular application contexts of CT are manifold. Still, it is important to conceive the basic computational concepts and principles in such a way as to keep up with the level of understanding developed in modern computer science. This is especially the case for the notion of "abstraction".

Our comparison of the perspectives on CT from computer science education and Inquiry Learning in science has brought forth the following main points:

1. The essence of Computational Thinking (CT) lies in the creation of "logical artifacts" that externalize and reify human ideas in a form that can be interpreted and "run" on computers. Accordingly, CT sets a focus on computational abstractions and representations—i.e., the computational artifact and how it is constituted is of interest as such and not only as a model of some scientific phenomenon.
2. Beyond the common sense understanding of "abstraction", computational abstractions (plural!) are constructive mind tools. The available abstractions (for data representation and processing) form a repertoire of possible choices for the creation of computational artifacts.
3. Inquiry learning uses computational artifacts and/or systems as models of natural phenomena, often in the form of (programmable) simulations. Here, the choice of the computational representation is usually predetermined and not in the focus of the learners' own creative contributions.
4. Inquiry learning as well as CT-related learning activities both exhibit and rely on cyclic patterns of model progression (cycle of inquiry steps - creation/revision cycle).

There is a huge potential for a further productive co-development of CT-centric educational environments and scenarios from multiple perspectives. "Representational flexibility" as the handing over of choices related to data structuring and other abstractions to the learners is desirable from a computer science point of view. This does not rule out the meaningful and productive use of more fixed computational models in other learning contexts. Yet, this future co-development of CT should benefit from taking up and exploring new types of abstractions and models of computation (including, e.g., different abstract engines or meta-level reasoning techniques) to enrich the learning space of CT. This may also reconnect the discourse to epistemological principles.

Acknowledgements This article is dedicated to the memory of Sören Werneburg who made important and essential contributions to this work, including the full responsibility for the design and implementation of the *ctMazeStudio* and *ctGameStudio* environments.

References

Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal, 55*(7), 832–835.

Ash, D. (2003). Inquiry Thoughts, Views and Strategies for the K-5 Classroom. In *Foundations: A monograph for professionals in science, mathematics and technology education*.

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.

Basu, S., Kinnebrew, J., Dickes, A., Farris, A., Sengupta, P., Winger, J., … Biswas, G. (2012). A science learning environment using a computational thinking approach. In *Proceedings of the 20th International Conference on Computers in Education*.

Bull, S., & Kay, J. (2010). Open Learner Models. In *Advances in intelligent tutoring systems* (pp. 301–322): Springer.

Bundy, A. (2007). Computational thinking is pervasive. *Journal of Scientific and Practical Computing, 1*(2), 67–69.

Curzon, P., & McOwan, P. W. (2016). *The power of computational thinking: Games, magic and puzzles to help you become a computational thinker*: World Scientific

diSessa, A. (2000). Changing minds: Computers. *Learning and Literacy*.

Giere, R. (1988). Laws, theories, and generalizations.

Guzdial, M. (2004). Programming environments for novices. *Computer Science Education Research, 2004,* 127–154.

Hartmann, W., Nievergelt, J., & Reichert, R. (2001). Kara, finite state machines, and the case for programming as part of general education. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments*.

Hoppe, H. U. (1994). Deductive error diagnosis and inductive error generalization for intelligent tutoring systems. *Journal of Interactive Learning Research, 5*(1), 27.

Hu, C. (2011). Computational thinking: What it might mean and what we might do about it. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*.

ISTE, & CSTA. (2011). *Computational thinking in K–12 education leadership toolkit*.

Kafura, D., & Tatar, D. (2011). Initial experience with a computational thinking course for computer science students. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*.

Kalelioglu, F., Gülbahar, Y., & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing, 4*(3), 583.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., et al. (2011). Computational thinking for youth in practice. *ACM Inroads, 2*(1), 32–37.

Lehrer, R., & Schauble, L. (2006). Scientific thinking and science literacy. In *Handbook of Child Psychology*.

Lloyd, J. W. (1987). Declarative error diagnosis. *New Generation Computing, 5*(2), 133–154.

Locke, J. (1700). *An essay concerning human understanding*.

Lockwood, J., & Mooney, A. (2017). Computational thinking in education: Where does it fit? *A Systematic Literary Review*.

Manlove, S., Lazonder, A. W., & Jong, T. D. (2006). Regulative support for collaborative scientific inquiry learning. *Journal of Computer Assisted Learning, 22*(2), 87–98.

Manske, S., & Hoppe, H. U. (2014). Automated indicators to assess the creativity of solutions to programming exercises. In *2014 IEEE 14th International Conference on Advanced Learning Technologies (ICALT)*.

Matsuzawa, Y., Tanaka, Y., Kitani, T., & Sakai, S. (2017). A demonstration of evidence-based action research using information dashboard in introductory programming education. In *IFIP World Conference on Computers in Education*.

Minner, D. D., Levy, A. J., & Century, J. (2010). Inquiry-based science instruction—what is it and does it matter? Results from a research synthesis years 1984 to 2002. *Journal of Research in Science Teaching, 47*(4), 474–496.

Nersessian, N. J. (1992). How do scientists think? Capturing the dynamics of conceptual change in science. *Cognitive Models of Science, 15,* 3–44.

NRC. (2008). *Public participation in environmental assessment and decision making*: National Academies Press.

Papert, S. (1996). An Exploration in the Space of Mathematics Educations. *International Journal of Computers for Mathematical Learning, 1*(1), 95–123.

Pedaste, M., Mäeots, M., Siiman, L. A., De Jong, T., Van Riesen, S. A., Kamp, E. T., et al. (2015). Phases of inquiry-based learning: Definitions and the inquiry cycle. *Educational Research Review, 14,* 47–61.

Perkins, D. N., & Simmons, R. (1988). Patterns of misunderstanding: An integrative model for science, math, and programming. *Review of Educational Research, 58*(3), 303–326.

Rieber, L. P. (1996). Microworlds. In *Handbook of research for educational communications and technology* (Vol. 2, pp. 583–603).

Schraw, G. (1998). Promoting general metacognitive awareness. *Instructional Science, 26*(1–2), 113–125.

Sengupta, P., Kinnebrew, J. S., Basu, S., Biswas, G., & Clark, D. (2013). Integrating computational thinking with K-12 science education using agent-based computation: A theoretical framework. *Education and Information Technologies, 18*(2), 351–380.

van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices, 35*(6), 26–36.

Weintrop, D., & Wilensky, U. (2015). To block or not to block, that is the question: Students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*.

White, B. Y., Shimoda, T. A., & Frederiksen, J. R. (1999). Enabling students to construct theories of collaborative inquiry and reflective learning: Computer support for metacognitive development. *International Journal of Artificial Intelligence in Education (IJAIED), 10,* 151–182.

Wichmann, A., & Leutner, D. (2009). Inquiry learning: Multilevel support with respect to inquiry, explanations and regulation during an inquiry cycle. *Zeitschrift für Pädagogische Psychologie, 23*(2), 117–127.

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

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 366*(1881), 3717–3725.